Comprehensive Implementation Report for a Network-Enabled Commercial Freezer Controller

Section 1: System Architecture for a Connected Controller

This section establishes the high-level firmware and network architecture required to develop a robust, reliable, and remotely accessible freezer controller. The design choices detailed herein are foundational, prioritizing system stability and deterministic control while enabling a modern, responsive user interface. The architecture is specifically tailored to the capabilities of the Waveshare ESP32-S3 4.3inch Capacitive Touch Display Development Board (Type B) and integrates the principles of fault tolerance and state management from prior system analyses.

1.1 The Dual-Core Paradigm: Assigning Responsibilities to Core 0 (Control) and Core 1 (Interface)

The ESP32-S3's dual-core processor is not merely a performance enhancement but a critical architectural component for ensuring the reliability of a commercial refrigeration system. The core control logic of the freezer, which manages time-sensitive operations such as anti-short-cycle delays, defrost timers, and probe readings, demands deterministic, uninterrupted execution. As established in the fault-tolerance analysis, any blocking function call within the main control task can starve the scheduler, prevent the system's watchdog timer from being reset, and lead to a catastrophic cycle of hangs and reboots. A web server, by its very nature, involves handling network requests that can block for unpredictable durations, posing an unacceptable risk to the core refrigeration functions.

To mitigate this risk entirely, a strict separation of concerns must be enforced at the hardware level by dedicating each processor core to a specific role. This architectural decision creates a "firewall" between the time-critical control logic and the non-deterministic interface tasks, ensuring that network latency or a slow client can never compromise the freezer's operation.

- **Core 0 (The "Control Core"):** This core will be exclusively dedicated to running a single, high-priority FreeRTOS task, herein referred to as control_logic_task. This task is the heart of the freezer. It will execute the primary operational state machine, manage all timers (e.g., AC, idF, MdF), perform all sensor readings using the non-blocking, asynchronous methods detailed in the fault-tolerance report, and directly command the state of the relays. Crucially, this task is also solely responsible for periodically resetting the Task Watchdog Timer (TWDT). This guarantees that the freezer's safety and core functionality are preserved under all circumstances. The FreeRTOS API call xTaskCreatePinnedToCore will be used to permanently assign this task to Core 0, preventing the scheduler from ever moving it.
- **Core 1 (The "Interface Core"):** This core will be responsible for all other system functions, particularly those that are non-deterministic or user-facing. This includes the

main Arduino loop() function, which will service the LVGL graphics library's timer handler (lv_timer_handler) for the physical HMI, ensuring a smooth and responsive touch screen experience. The entire network stack will also run on this core. This encompasses the Wi-Fi connection management, the ESPAsyncWebServer instance, the WebSocket server, and any clients used for sending remote notifications (e.g., SMTP or Pushover clients). By offloading these responsibilities to a separate core, the user interface—both local and remote—remains fluid and interactive, regardless of the state or workload of the control logic.

This dual-core paradigm is a non-negotiable architectural foundation for a commercial-grade product. A simpler, single-threaded approach would inevitably lead to scenarios where, for example, a blocking web request could prevent the timely termination of a defrost cycle, creating a potential fire hazard or causing damage to the freezer's contents and the refrigerated product. This separation ensures that the controller is both feature-rich and fundamentally safe.

1.2 Thread-Safe Data Exchange: Implementing Mutexes for Shared State and Parameter Structures

The segregation of tasks across two cores introduces a classic software engineering challenge: concurrent access to shared resources. The control_logic_task on Core 0 needs to read from the ParameterMap structure (containing settings like St, Hy, tdF, etc.) and continuously write to the ControllerState structure (updating values like temp_p1, relay_state, etc.). Concurrently, the network task on Core 1 must read from the ControllerState to provide real-time updates to the web interface and write to the ParameterMap whenever a user submits a new setting.

Unprotected access to these shared data structures from different cores is a direct path to data corruption via race conditions. For instance, the web server on Core 1 might begin reading the ParameterMap for display at the exact moment the user saves a new set of values, resulting in a partially updated, inconsistent state being sent to the web application. A more dangerous scenario involves the control task on Core 0 reading a partially updated setpoint (St) value, causing the compressor to cycle to a dangerously incorrect temperature. These types of intermittent, difficult-to-reproduce "Heisenbugs" are unacceptable in a reliable system.

The standard and robust solution to this problem in a FreeRTOS environment is the use of mutual exclusion semaphores, or **mutexes**. A mutex acts as a key for a shared resource, ensuring that only one task can access that resource at any given time.

To ensure data integrity, the firmware will implement two distinct mutexes:

- SemaphoreHandle_t stateMutex;
- SemaphoreHandle_t paramsMutex;

These mutexes will be used to protect their corresponding data structures. Any block of code, on either core, that needs to read from or write to the global ControllerState object must first acquire the stateMutex using xSemaphoreTake(stateMutex, portMAX_DELAY) and must release it immediately afterward using xSemaphoreGive(stateMutex). Likewise, all access to the ParameterMap object will be bracketed by calls to take and give the paramsMutex. This disciplined approach prevents race conditions by serializing access to the shared memory, transforming a potential source of chaos into a predictable and reliable system.

1.3 The Technology Stack: Justifying the Selection of ESPAsyncWebServer, WebSockets, and LittleFS

To deliver a modern, responsive, and maintainable web interface, a specific combination of libraries is required. The selection of this technology stack is driven by the need for high performance and ease of development.

- **ESPAsyncWebServer:** Standard Arduino WebServer libraries operate on a synchronous, blocking model, which is inefficient and unsuitable for this application. The ESPAsyncWebServer library, by contrast, is built on AsyncTCP and handles network requests asynchronously. This means it can manage multiple simultaneous client connections without blocking the main program loop, which is essential for a device that must remain responsive while serving a web application.
- **WebSockets:** A traditional HTTP polling approach, where the web browser repeatedly asks the server for updates, is inefficient and introduces latency. The WebSocket protocol provides a superior alternative by establishing a persistent, full-duplex communication channel between the browser and the ESP32. Once this connection is established, the ESP32 can *push* data (such as temperature changes, relay status, or alarm notifications) to the web interface in real-time, without waiting for a request. This is the key technology that enables an "instantaneous" user experience, where UI elements update the moment a state changes on the device.
- **LittleFS:** Embedding HTML, CSS, and JavaScript code directly into C++ source code as giant strings is a common but deeply flawed practice. It makes the frontend code extremely difficult to write, read, and maintain. A far more professional approach is to store the entire web application—index.html, style.css, and script.js files—on the ESP32's onboard flash memory using a filesystem. LittleFS is a modern, wear-leveling filesystem designed specifically for microcontrollers and is the recommended choice for this project. This approach completely decouples the frontend web development from the backend C++ firmware development, allowing for independent updates and a much cleaner project structure.

This combination of ESPAsyncWebServer, WebSockets, and LittleFS is not merely a suggestion but the industry-standard methodology for creating high-performance, maintainable embedded web applications on the ESP32 platform.

1.4 WiFi Credential Management: Designing a WiFi Manager for On-Site Network Configuration

A commercial product cannot be shipped with hard-coded WiFi credentials. The device must be easily configurable by a technician at the installation site without requiring a firmware re-flash. The established solution for this is a **WiFi Manager**. This feature is fundamental to the "out-of-box experience" and transforms the controller from a hobbyist prototype into a field-deployable product.

The implementation will follow a well-defined sequence:

1. **Boot Sequence:** Upon startup, the controller will attempt to load saved WiFi credentials from its Non-Volatile Storage (NVS).
2. **Connection Attempt:** It will try to connect to the saved network in Station (STA) mode.
3. **Fallback to Access Point (AP) Mode:** If no credentials are found, or if the connection fails after a short timeout, the controller will cease trying to connect and will instead create its own WiFi network by entering Access Point (AP) mode. This network will have a recognizable SSID, such as Freezer_Controller_Setup.
4. **Configuration Portal:** A technician can then use a smartphone or laptop to connect to

this temporary network. Upon connection, they can navigate to a fixed IP address (typically 192.168.4.1) in their web browser.

5. **Credential Submission:** The ESP32 will serve a simple HTML form page that allows the technician to scan for local networks, select the correct one, and enter its password.
6. **Save and Reboot:** When the form is submitted, the ESP32 saves the new SSID and password securely to NVS and immediately reboots. Upon restarting, it will successfully connect to the local network using the newly provided credentials.

This self-configuration capability is a critical non-functional requirement that directly impacts the product's usability and professionalism.

Section 2: Backend Implementation: The ESP32 Web Server

This section provides the detailed C++ firmware blueprint for the server-side logic that will run on the ESP32's "Interface Core" (Core 1). These code structures will handle all network interactions, serving the web application and managing the real-time data flow required for remote control and monitoring.

2.1 Initializing the Server and Filesystem

The initialization process within the setup() function must follow a specific order to ensure all components are ready before accepting connections.
First, the LittleFS filesystem must be mounted. This makes the web application files (index.html, etc.) available to the server. The code will use LittleFS.begin() and include error-checking to handle potential mounting failures, which could indicate a corrupted filesystem.
Next, the core network objects are instantiated. An AsyncWebServer object is created to listen for HTTP requests on the standard web port, port 80. Concurrently, an AsyncWebSocket object is created and configured to listen on a specific path, such as /ws. A dedicated event handler function, onWsEvent, will be registered with the WebSocket object to manage its lifecycle events (connection, disconnection, and data reception).
The following code skeleton illustrates this initialization sequence:

```
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <LittleFS.h>
#include "shared_state.h" // Contains shared structs and mutexes

// Instantiate server objects
AsyncWebServer server(80);
AsyncWebSocket ws("/ws");

// Forward declaration of the WebSocket event handler
void onWsEvent(AsyncWebSocket *server, AsyncWebSocketClient *client,
AwsEventType type, void *arg, uint8_t *data, size_t len);

void setup_network_services() {
    // Mount the LittleFS filesystem
    if (!LittleFS.begin()) {
```

```
        Serial.println("An Error has occurred while mounting
LittleFS");
        // Handle error: perhaps enter a fault state
        return;
    }
    Serial.println("LittleFS mounted successfully.");

    // Attach the WebSocket event handler
    ws.onEvent(onWsEvent);
    server.addHandler(&ws);

    // Define routes for serving files (detailed in next section)
    //...

    // Start the server
    server.begin();
    Serial.println("HTTP and WebSocket server started.");
}
```

2.2 Serving the Frontend Application from LittleFS

To decouple the frontend from the firmware, the web server's primary role is to serve the static files (index.html, style.css, script.js) that comprise the web application. The ESPAsyncWebServer library provides an efficient mechanism for this.
Instead of defining a separate route for each file, a more robust and scalable solution is to implement a "catch-all" handler using server.onNotFound(). This function is executed whenever the server receives a request for a URL that does not match any other predefined route. Inside this handler, the code will interpret the requested path as a file path within the LittleFS filesystem.
The handler will perform the following steps:

1. Construct the full file path (e.g., /index.html, /style.css).
2. Check if the file exists using LittleFS.exists(path).
3. If the file exists, determine its MIME type based on the file extension.
4. Serve the file to the client using the request->send(LittleFS, path, mimeType) method. This function efficiently streams the file from the filesystem directly to the client.
5. If the file does not exist, respond with a standard 404 Not Found error.

This approach ensures that any file placed in the project's data directory is automatically available to be served by the web server, making frontend updates as simple as uploading a new filesystem image.

```
// This function will be called from within setup_network_services()
void setup_file_server_routes() {
    // Route for the root URL, serves index.html
    server.on("/", HTTP_GET,(AsyncWebServerRequest *request){
        request->send(LittleFS, "/index.html", "text/html");
    });

    // Route for serving CSS
```

```
    server.on("/style.css", HTTP_GET,(AsyncWebServerRequest *request){
        request->send(LittleFS, "/style.css", "text/css");
    });

    // Route for serving JavaScript
    server.on("/script.js", HTTP_GET,(AsyncWebServerRequest *request){
        request->send(LittleFS, "/script.js",
"application/javascript");
    });

    // Optional: Add routes for favicon or other assets
    //...

    // Catch-all for any other request
    server.onNotFound((AsyncWebServerRequest *request){
        request->send(404, "text/plain", "Not found");
    });
}
```

2.3 WebSocket Server: Handling Connections and Real-Time Data Flow

The WebSocket server is the communication backbone for the real-time interface. Its event handler, onWsEvent, is the central point for managing all client interactions.

- **WS_EVT_CONNECT:** When a new client (a browser tab) establishes a WebSocket connection, the server must immediately bring the new UI up to date. Upon this event, the server will be programmed to serialize the *entire* current state of the controller—including all probe temperatures, relay statuses, and the complete ParameterMap—into a single JSON packet. This "full state" packet is then sent directly to the newly connected client. This ensures that a user opening the web page sees the correct, current data instantly.
- **WS_EVT_DISCONNECT:** When a client disconnects, this event can be used to log the event for debugging purposes and to manage system resources.
- **WS_EVT_DATA:** This event is triggered when the server receives a message from a client. The message will be a JSON string formatted according to the API protocol defined in the next section. The data is passed to a central processing function, handleWebSocketMessage(), which will parse the JSON, validate the command, and take the appropriate action (e.g., updating a parameter in the shared ParameterMap struct).

To keep all connected clients synchronized, a dedicated function, broadcastStateUpdate(), will be implemented. This function will be called periodically (e.g., once per second) by the network task and also immediately after any significant state change (like an alarm being triggered). It will serialize the latest ControllerState into a compact JSON "update" packet and broadcast it to *all* connected clients using ws.textAll(). This is the mechanism that ensures every user sees temperature changes and relay activations in real-time.

2.4 The JSON API: A Definitive Protocol for Commands and State

Synchronization

A clearly defined, unambiguous Application Programming Interface (API) is essential for decoupling the backend firmware from the frontend web application. It serves as a formal contract, allowing for parallel development and simplifying debugging. JSON (JavaScript Object Notation) is the de facto standard for such APIs due to its human-readable format and ease of parsing in both C++ (with libraries like ArduinoJson) and JavaScript.
The following table formalizes the WebSocket JSON API protocol for all communication between the ESP32 server and the web client. This structure is critical for enabling modular, AI-assisted code generation, as it provides a precise specification to build against.

Table 1: WebSocket JSON API Protocol

Message Type	Direction	JSON Structure	Description
getFullState	Client → Server	{"cmd": "getState"}	A client requests a complete snapshot of the system's state and all parameters. Typically sent upon initial connection.
fullState	Server → Client	{"type": "fullState", "payload": {"temps":{...}, "relays":{...}, "params":{...}}}	The server's response to getFullState. Contains all current probe temperatures, relay statuses, and the complete set of configurable parameters from the ParameterMap.
updateState	Server → Client	{"type": "update", "payload": {"temps":{...}, "relays":{...}, "alarms":{...}}}	A periodic, partial update sent by the server to all clients, containing only the real-time data that changes frequently.
setParam	Client → Server	{"cmd": "setParam", "payload": {"param": "St", "value": -20.0}}	A client requests to change the value of a single configurable parameter. param is the label (e.g., "St", "Hy", "idF"), and value is the new setting.
paramConfirm	Server → Client	{"type": "paramConfirm", "payload": {"param": "St", "value": -20.0}}	The server confirms to the originating client that a parameter change was successfully received and applied.
directControl	Client → Server	{"cmd": "directControl",	A client on the

Message Type	Direction	JSON Structure	Description
		"payload": {"relay": "K1", "state": 1}}	authenticated "Direct Control" page requests to manually override the state of a specific relay (K1-K4). state is 1 for ON, 0 for OFF.
alarm	Server → Client	{"type": "alarm", "payload": {"code": "HA", "active": true, "msg": "High Temp"}}	The server pushes a new alarm status to all clients. code is the alarm code (e.g., "HA", "P1F"), and active is true or false.

Section 3: Frontend Implementation: The Web Application

This section details the construction of the client-side web application. It will be built using standard, universally supported web technologies—HTML, CSS, and JavaScript—to ensure maximum compatibility across browsers and devices. The entire application will be self-contained in three files (index.html, style.css, script.js) designed to be served from the ESP32's LittleFS filesystem.

3.1 HTML Structure: Building the User Interface Layout

The index.html file serves as the structural foundation of the web application. It defines the layout and provides the containers that will be dynamically populated with data by JavaScript. A single-page application (SPA) architecture is employed, where all content is loaded once, and UI updates are handled programmatically without full page reloads, providing a fluid, app-like experience.
The HTML body will be semantically structured with div elements, each assigned a unique ID for easy targeting by JavaScript:

- **id="connection-status":** A small, persistent element at the top or bottom of the page to provide visual feedback on the WebSocket connection status (e.g., "Connected", "Reconnecting...").
- **id="dashboard":** The main view, displaying the most critical real-time information: current cabin temperature, setpoint, and the status of key components like the compressor and fans.
- **id="parameters":** A comprehensive container that will house the full list of configurable parameters. This section will be dynamically generated by JavaScript based on the fullState data received from the ESP32. It will contain input fields for every parameter defined in the master dictionary.
- **id="direct-control":** The administrative panel for manual actuator control. This div will be hidden by default and will only become visible after successful authentication. It will contain toggle switches for each relay (Compressor, Fan, Defrost, Liquid Line Solenoid).
- **id="alarm-banner":** An initially hidden container used to display active alarm messages (e.g., "HIGH TEMPERATURE ALARM").

This modular HTML structure allows for a clean separation of concerns and facilitates the dynamic rendering and updating of the user interface.

3.2 CSS Styling: Implementing a Responsive, Professional Design

The visual presentation of the interface is defined in the style.css file. The primary goal is to create a clean, professional, and responsive design that is equally usable on a large desktop monitor in an office and a small smartphone screen in the field. This aligns with the "pro-sumer" design philosophy, which blends consumer-grade aesthetics with industrial clarity.
Key CSS techniques will be employed to achieve this:

- **Responsive Layout:** Modern CSS layout models like Flexbox and Grid will be used to structure the main page elements. This ensures that content blocks (like the parameter list) naturally reflow and stack vertically on narrow screens, preventing horizontal scrolling and maintaining readability.
- **Media Queries:** CSS media queries will be used to apply different styles based on the screen width. For example, on screens wider than 800 pixels, the parameter list might be displayed in two columns, while on screens narrower than 800 pixels, it will switch to a single-column layout. Font sizes and button padding will also be adjusted for optimal legibility and touch-friendliness on mobile devices.
- **Visual Hierarchy:** Styling will be used to enforce the visual hierarchy established in the HMI design. The current temperature will have the largest font size. Parameter labels will be clear and distinct from their values. Buttons will have clear hover and active states to provide user feedback. The color palette from the HMI specification will be used consistently, with restrictive use of alarm colors like red.

3.3 JavaScript Core: WebSocket Client and Dynamic UI Updates

The script.js file is the engine of the web application. It is responsible for all client-side logic, including communication with the ESP32 and manipulation of the HTML Document Object Model (DOM).
The script's functionality can be broken down into three main areas:

1. **WebSocket Management:**
    - On page load, the script will immediately attempt to establish a WebSocket connection to the ESP32 server at the address ws://{CURRENT_HOST}/ws. Using window.location.hostname ensures the script works without modification, whether accessed via a local IP or a remote ngrok URL.
    - Event handlers will be defined for the four key WebSocket events:
        - onopen: Fired when the connection is successful. The script will update the #connection-status indicator and send a {"cmd": "getState"} message to request the initial data dump.
        - onclose: Fired when the connection is lost. The script will update the status indicator and trigger a timer to automatically attempt reconnection every few seconds, ensuring a resilient user experience.
        - onerror: To log any connection errors for debugging.
        - onmessage: This is the most critical handler. It is fired every time a message is received from the ESP32.
2. **Incoming Data Processing:**
    - The onmessage handler will parse the incoming JSON data packet. A switch

statement will be used to route the logic based on the type field of the JSON object (as defined in Table 1).
    - If type is fullState, the script will execute a function to populate the entire UI. This includes iterating through the params object and dynamically creating the HTML input elements and labels for the #parameters section.
    - If type is updateState, the script will update only the specific elements in the #dashboard that display real-time data (e.g., document.getElementById('current-temp').innerText = data.payload.temps.p1;). This is far more efficient than redrawing the entire page.
    - If type is alarm, the script will show or hide the #alarm-banner and update its text content.
3. **Outgoing Command Handling:**
    - Event listeners will be attached to every interactive element (input fields, buttons, switches).
    - When a user changes a parameter value in an input field, the change event will trigger a function that constructs a {"cmd": "setParam", "payload": {...}} JSON packet and sends it to the ESP32 using websocket.send().
    - Clicking a button on the "Direct Control" panel will similarly construct and send a {"cmd": "directControl", "payload": {...}} packet.

This event-driven, data-centric JavaScript architecture ensures that the UI is always a direct reflection of the freezer's state, providing a seamless and reliable control experience for the user.

Section 4: Enabling Secure Remote Access and Control

A core requirement of the project is to provide secure access to the controller's web interface from outside the local network. This capability is essential for remote management, monitoring, and diagnostics. This section analyzes various access strategies and presents a robust, secure, and user-friendly implementation plan.

4.1 Analysis of Remote Access Strategies

Exposing an embedded device on a local network to the public internet can be achieved through several methods, each with significant implications for security and ease of use.

- **Port Forwarding:** This traditional method involves manually configuring the local network's router to forward incoming traffic on a specific port (e.g., port 80) to the ESP32's internal IP address. While functional, this approach is fraught with problems. It is complex for non-technical users to configure, as it requires accessing router administration settings. More importantly, it is highly insecure. It directly exposes the ESP32 to the public internet, making it a target for automated scans, vulnerability exploits, and denial-of-service attacks.
- **Home VPN Server:** Setting up a VPN server on the local network (e.g., on a Raspberry Pi or a capable router) provides a secure, encrypted tunnel into the network. A remote user would connect to the VPN and then be able to access the ESP32 as if they were on the local network. While this is a very secure method, its setup and maintenance are significantly more complex than port forwarding and are generally beyond the scope of a

typical end-user.

- **Tunneling Service (ngrok):** Services like ngrok provide a modern, secure alternative that bypasses the complexities and security risks of the other methods. The ESP32's web server remains entirely on the local network, never directly exposed. Instead, a small, lightweight ngrok agent running on a computer within the same network initiates a secure *outbound* connection to the ngrok cloud service. The service then provides a public, encrypted (HTTPS) URL that tunnels traffic back to the local ngrok agent, which in turn forwards it to the ESP32. This model is inherently more secure because no inbound ports need to be opened on the local firewall.

The following table provides a clear comparison, justifying the selection of a tunneling service as the recommended approach for this project.

Table 2: Remote Access Method Comparison

Method	Security	Setup Complexity	Reliability	Cost
Port Forwarding	**Very Low.** Directly exposes the device to the public internet, requiring robust on-device security.	**High.** Requires manual router configuration, which is error-prone and varies by manufacturer.	**Moderate.** Dependent on router stability and dynamic DNS services if the public IP changes.	Free (monetary).
Home VPN Server	**High.** Creates a secure, encrypted private network.	**Very High.** Requires setting up and maintaining a dedicated VPN server on the network.	**High.** Dependent on the stability of the VPN server hardware/software.	Low to moderate (for hardware).
ngrok Tunneling	**Very High.** The device is never directly exposed. Traffic is encrypted via a public URL. Authentication can be added at the tunnel level.	**Low.** Requires running a single command on one local computer. No router configuration needed.	**Very High.** Managed by a professional cloud service with high uptime.	Free tier available; paid plans for custom domains and advanced features.

4.2 Implementation Guide: Using ngrok for Secure, Zero-Configuration Tunneling

Based on the analysis, ngrok is the recommended solution. The setup process is straightforward and can be performed by a system administrator or a technically proficient user on-site.

1. **Account and Agent Setup:**
    - Create a free account at the ngrok dashboard to obtain an authentication token.
    - Download the ngrok agent appropriate for the operating system of a computer that will be always-on and connected to the same local network as the freezer controller (e.g., a back-office PC or a dedicated Raspberry Pi).
    - Configure the agent with the authentication token by running the command provided on the ngrok dashboard. This is a one-time setup step.

2. **Starting the Tunnel:**
    - First, determine the local IP address of the ESP32 controller. This can be found via the device's serial monitor output during boot or from the WiFi Manager interface.
    - On the computer running the ngrok agent, execute the following command in a terminal, replacing <ESP32_IP_ADDRESS> with the actual address:
      `ngrok http <ESP32_IP_ADDRESS>:80`
    - The ngrok agent will connect to the cloud service and display a public URL in the terminal (e.g., https://random-string.ngrok.io).
3. **Remote Access:** This public URL is now the secure gateway to the freezer controller's web interface. It can be accessed from any device with an internet connection worldwide. For a permanent installation, a paid ngrok plan offers the ability to use a stable, custom subdomain (e.g., https://myfreezer.ngrok.io), which is highly recommended for a professional product.

4.3 Authentication for the "Direct Control" Panel

The "Direct Control" panel is a powerful administrative tool that must be protected from unauthorized access. While authentication can be implemented on the ESP32 itself, a more secure approach is to offload this responsibility to the ngrok tunnel.

- **On-Device Authentication (Alternative):** It is possible to protect specific routes on the ESPAsyncWebServer using HTTP Basic Authentication. The library provides a request->authenticate(user, pass) method that, when applied to a route like /admin, would trigger a browser login prompt. However, this places the burden of correctly and securely implementing the authentication logic on the ESP32 firmware.
- **Offloaded Authentication (Recommended):** A superior and simpler method is to use ngrok's built-in authentication features. By adding a single flag to the ngrok command, a robust authentication layer can be applied to the entire tunnel. This means an unauthorized user cannot even send a single packet of data to the ESP32.
  `ngrok http <ESP32_IP_ADDRESS>:80 --basic-auth="admin:SecurePassword123"`
  This command instructs ngrok to require a username and password before allowing any traffic through the tunnel. This moves the security perimeter from the resource-constrained microcontroller to a professionally managed, robust cloud service, which is an inherently more secure architecture. Paid ngrok plans offer even more advanced options, such as OAuth integration with providers like Google.

4.4 Advanced Security: On-Device Hardware Security Features

For a production-grade commercial device, software-level security should be complemented by hardware-level protections available on the ESP32-S3 SoC. Enabling these features provides defense-in-depth against sophisticated attacks.

- **Secure Boot:** This feature uses cryptographic signatures to ensure that the device will only execute authentic software. During manufacturing, a signing key is generated and its hash is permanently burned into the ESP32's eFuses. All subsequent firmware, including OTA updates, must be signed with the corresponding private key. The bootloader will refuse to load any firmware with an invalid signature, preventing unauthorized modifications or malicious firmware from being run on the device.

- **Flash Encryption:** This feature encrypts the entire contents of the external flash memory using a key stored within the ESP32. Once enabled, the firmware, the LittleFS filesystem containing the web application, and any saved credentials in NVS are unreadable to anyone who physically desolders and attempts to read the flash chip. This protects the application's intellectual property and sensitive user data from physical extraction.

Enabling these one-time programmable eFuse features is a critical step in hardening the device for commercial deployment, providing a strong foundation of trust and confidentiality.

Section 5: Implementing the Remote Notification System

A critical function of the controller is to proactively alert users to alarm conditions, such as high/low temperatures or probe failures. A timely notification can prevent the spoilage of thousands of dollars worth of product. This section details two robust methods for implementing this remote alerting capability.

5.1 A Comparison of Notification Channels: Email (SMTP) vs. Push Notifications (Pushover)

The choice of notification channel depends on the desired urgency, reliability, and ease of setup. The two leading candidates for an ESP32-based system are direct email via SMTP and a dedicated push notification service like Pushover.

- **Email (SMTP):** Sending an email is a universally understood notification method. Using the ESP-Mail-Client library, the ESP32 can connect directly to an SMTP server (like Gmail's) and send a formatted HTML email. While reliable, email is not designed for immediate, high-priority alerting. Emails can be delayed, filtered into spam, or simply lost in a cluttered inbox.
- **Pushover:** Pushover is a service designed specifically for sending real-time push notifications to iOS and Android devices, as well as desktop browsers. It excels at delivering critical alerts. Notifications can be assigned different priority levels, from silent to an emergency-level alert that bypasses the user's quiet hours and requires explicit acknowledgment. It also supports custom alert sounds, allowing for distinct notifications for different types of alarms.

The following table compares these two methods to guide the selection process.

Table 3: Notification Method Comparison

Method	Setup Effort	Alert Immediacy	Customization (Priority/Sounds)	Ideal Use Case
Email (SMTP)	**Moderate.** Requires setting up a dedicated email account and generating an App Password, which can be a complex process for users unfamiliar with it.	**Low to Moderate.** Delivery can be delayed by mail servers. Alerts can be easily missed in a busy inbox.	**Low.** All alerts are treated as standard emails.	General status updates, non-critical logs, or notifications where immediate action is not required.

Method	Setup Effort	Alert Immediacy	Customization (Priority/Sounds)	Ideal Use Case
Pushover	**Low.** Requires creating an account and an "application" on the Pushover website to get a user key and API token. The process is straightforward.	**Very High.** Notifications are delivered instantly via native push services.	**High.** Supports five priority levels, including emergency alerts that repeat until acknowledged. Supports custom sound choices per application.	**Critical Alerts.** High/low temperature alarms, probe failures, or any condition requiring immediate user attention.

For a commercial freezer application where a rapid response to an alarm is paramount, **Pushover is the strongly recommended solution** due to its immediacy and advanced priority features. Email remains a viable secondary option for less critical reports.

5.2 Implementation Guide: Sending Formatted Alerts with Pushover

Implementing Pushover notifications involves making a secure HTTPS POST request to the Pushover API.

1. **Pushover Account Setup:**
    - Create an account on the Pushover website.
    - Note your **User Key**, which is displayed on the main dashboard.
    - Create a new "Application" to get a unique **API Token**. This allows you to customize the name and icon of the notifications sent from the freezer.
2. **Firmware Implementation:**
    - A dedicated function, sendPushoverNotification(String message, int priority), will be created in the firmware.
    - This function will use the HTTPClient and WiFiClientSecure libraries to perform the HTTPS request.
    - To validate the secure connection, the Pushover API's root CA certificate must be included in the code.
    - The function will construct a JSON payload containing the mandatory token and user keys, along with the message, a title (e.g., "Freezer Alarm"), and the desired priority level (e.g., 2 for emergency).
    - This function will be integrated into the control_logic_task. When the controller's state machine detects a confirmed alarm condition (e.g., the high-temperature alarm HA is active), it will call this function with a descriptive message like "High Temperature Alarm: Cabinet is -5.2°C" and a high priority.

```
#include <HTTPClient.h>
#include <WiFiClientSecure.h>
#include <ArduinoJson.h>

// Pushover API details
const char* PUSHOVER_API_TOKEN = "YOUR_APP_API_TOKEN";
const char* PUSHOVER_USER_KEY = "YOUR_USER_KEY";
const char* PUSHOVER_ROOT_CA = "-----BEGIN
```

```
CERTIFICATE-----\n...\n-----END CERTIFICATE-----\n"; // Pushover's
root CA

void sendPushoverNotification(String message, int priority) {
    if (WiFi.status() == WL_CONNECTED) {
        WiFiClientSecure client;
        client.setCACert(PUSHOVER_ROOT_CA);

        HTTPClient https;
        if (https.begin(client,
"https://api.pushover.net/1/messages.json")) {
            https.addHeader("Content-Type", "application/json");

            StaticJsonDocument<256> doc;
            doc["token"] = PUSHOVER_API_TOKEN;
            doc["user"] = PUSHOVER_USER_KEY;
            doc["title"] = "Freezer Controller Alert";
            doc["message"] = message;
            doc["priority"] = priority;

            String jsonPayload;
            serializeJson(doc, jsonPayload);

            int httpCode = https.POST(jsonPayload);
            if (httpCode > 0) {
                if (httpCode == HTTP_CODE_OK) {
                    Serial.println("Pushover notification sent
successfully.");
                }
            } else {
                Serial.printf(" POST... failed, error: %s\n",
https.errorToString(httpCode).c_str());
            }
            https.end();
        }
    }
}
```

5.3 Alternative Implementation: Sending Alerts via Email (SMTP)

For users who prefer email, or for sending less critical daily/weekly status reports, an SMTP-based implementation will be provided as a complete alternative.

1. **Email Account Setup:**
    - The most critical and often overlooked step is preparing the sender's email account. The guide will provide explicit instructions to create a **new, dedicated Gmail account** for the controller.
    - It will then walk through the process of enabling 2-Step Verification and generating

a 16-digit **App Password**. Using a standard account password will fail; the App Password is required by Google for programmatic access.

2. **Firmware Implementation:**
    - A function sendEmailAlert(String subject, String htmlContent) will be created using the ESP-Mail-Client library.
    - This function will instantiate and configure an SMTPSession object with the SMTP server details (smtp.gmail.com, port 465), the sender's full email address, and the generated App Password.
    - It will then create an SMTP_Message object, setting the recipient's email address, the subject line, and the message body. The library supports sending richly formatted HTML content, allowing for professional-looking alert emails.
    - This function can be called by the control logic in the same way as the Pushover function to send alerts when fault conditions are detected.

Section 6: Final Integration and Commissioning Blueprint

This final section consolidates all the preceding architectural components, code blueprints, and implementation strategies into a unified project structure. It provides a clear path for building, deploying, and commissioning the controller, as well as a roadmap for future enhancements.

6.1 The Complete Code Structure: A Unified Blueprint for the Project

To promote maintainability, scalability, and clarity, the project's firmware should be organized into a modular file structure. This approach moves beyond a single, monolithic .ino file and separates code based on functionality.
The recommended platformio.ini configuration file will declare all necessary library dependencies and build flags:

```
[env:waveshare_esp32_s3_touch_lcd_4_3b]
platform = espressif32
board = esp32s3box
framework = arduino
monitor_speed = 115200

; Enable Octal PSRAM and set Flash speed/mode for performance
board_upload.flash_size = 16MB
board_build.arduino.memory_type = qio_opi
board_build.f_flash = 80000000L
board_build.flash_mode = qio
board_build.filesystem = littlefs

; Global Build Flags for Libraries
build_flags =
    -D BOARD_HAS_PSRAM
    -D LV_CONF_INCLUDE_SIMPLE
    -D LV_TICK_PERIOD_MS=5
```

```
; Library Dependencies
lib_deps =
    lvgl/lvgl@~8.3.11
    espressif/ESP32_Display_Panel
    espressif/ESP32_IO_Expander
    bblanchon/ArduinoJson@^6.0
    ottowinter/ESPAsyncWebServer-esphome@^3.0.0
    mobizt/ESP Mail Client@^3.0
```

The source code will be organized into the following file structure within the PlatformIO project:

- src/
    - main.cpp: The main entry point. Handles initial setup of serial, peripherals, and NVS. Creates and starts the FreeRTOS tasks for control and networking. The loop() function will be minimal, primarily yielding to the RTOS.
    - control_task.cpp: Contains the control_logic_task function and all related state machine logic, sensor reading, and relay control. This code runs exclusively on Core 0.
    - network_task.cpp: Contains the network_interface_task function. This includes the WiFi Manager logic, ESPAsyncWebServer and WebSocket setup, and all handlers for serving files and processing API requests. This code runs on Core 1.
    - notifications.cpp: Contains the implementation for sendPushoverNotification() and/or sendEmailAlert().
    - shared_state.h: A critical header file included by all tasks. It defines the ControllerState and ParameterMap structs, and declares the global instances of these structs and their corresponding mutexes (extern SemaphoreHandle_t...).
- data/
    - index.html: The main HTML file for the single-page application.
    - style.css: The CSS stylesheet for the web application.
    - script.js: The client-side JavaScript for handling WebSocket communication and dynamic UI updates.

This clean, decoupled structure is the final "building block" representation, making the system easy for a development team to work on and for an AI to parse and extend.

6.2 Deployment Workflow: A Step-by-Step Checklist for Flashing and Testing

A structured deployment process is essential for ensuring all components work together correctly.

1. **Install Filesystem Uploader:** Ensure the appropriate filesystem uploader plugin for the chosen IDE (Arduino IDE or PlatformIO/VSCode) is installed. For Arduino IDE, this is the "ESP32 Sketch Data Upload" tool. For PlatformIO, this is handled automatically.
2. **Populate Data Directory:** Place the final index.html, style.css, and script.js files into the project's data folder.
3. **Upload Filesystem Image:** Use the IDE's tool to build and upload the LittleFS filesystem image to the ESP32. This must be done before uploading the main firmware.
4. **Upload Firmware:** Compile and upload the main C++ firmware to the ESP32 via its USB-C port.

5. **Initial Configuration:** Open a serial monitor to observe the boot process. On the first boot, the device should enter AP mode. Use a phone or laptop to connect to the "Freezer_Controller_Setup" network and navigate to 192.168.4.1 to configure the local WiFi credentials.
6. **Local Network Test:** After the device reboots and connects to the local WiFi, find its new IP address from the serial monitor. Access this IP address from a browser on the same network to fully test the web interface, including parameter changes and direct control (after authentication).
7. **Remote Access Setup:** On a separate, always-on computer on the same network, install and run the ngrok agent, pointing it to the ESP32's local IP address and enabling authentication.
8. **Remote Test:** Use the public ngrok URL from a device on a completely different network (e.g., a smartphone on a cellular connection) to verify remote access and control.
9. **Alarm Test:** Manually trigger an alarm condition (e.g., by setting the high-temperature alarm threshold ALU to a very low value) and verify that a notification is correctly received via Pushover or email.

6.3 Roadmap for Future Enhancements: Beyond the Current Scope

The architecture detailed in this report provides a powerful and scalable foundation. To ensure the product remains competitive and continues to add value, a strategic roadmap for future enhancements is recommended.

- **Phase 2 (True Cloud Integration):** While ngrok provides excellent secure remote access, a more scalable solution for managing a fleet of devices involves a true IoT cloud platform. The next logical step would be to transition the communication protocol from WebSockets to MQTT. The ESP32 would act as an MQTT client, connecting to a central broker (e.g., a managed service like AWS IoT Core). This would enable centralized fleet management, secure OTA updates, and a much more robust architecture for data aggregation and remote control, eliminating the need for an on-site ngrok agent.
- **Phase 3 (Data Logging and Compliance):** The commercial food industry often requires strict temperature monitoring for compliance with regulations like HACCP (Hazard Analysis and Critical Control Points). The firmware can be enhanced to log temperature and alarm history at regular intervals. This data could be stored locally on the LittleFS filesystem for later download or, more effectively, published via MQTT to a cloud database for long-term storage, analysis, and automated compliance reporting.
- **Phase 4 (Intelligent and Adaptive Control):** With a robust data pipeline in place, the controller can evolve from a reactive to a proactive system. The most significant efficiency improvement would be the implementation of an adaptive or "on-demand" defrost cycle, as suggested in the initial system report. By analyzing the thermal relationship between the cabin (P1) and evaporator (P2) probes against compressor run times, the controller can intelligently infer the buildup of performance-degrading frost. This would allow it to initiate defrost cycles only when truly necessary, rather than on a fixed timer, leading to dramatic reductions in energy consumption and thermal stress on the refrigerated products—a powerful competitive differentiator.

Works cited

1. ESP32 as a Webserver - NextPCB, https://www.nextpcb.com/blog/esp32-as-a-webserver 2.

ESP32 Web Server - Arduino IDE | Random Nerd Tutorials, https://randomnerdtutorials.com/esp32-web-server-arduino-ide/ 3. Building an ESP32 Web Server: The Complete Guide for Beginners, https://randomnerdtutorials.com/esp32-web-server-beginners-guide/ 4. FreeRTOS - ESP32 - — ESP-IDF Programming Guide v4.2 documentation, https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-reference/system/freertos.html 5. How to use mutex - Programming - Arduino Forum, https://forum.arduino.cc/t/how-to-use-mutex/964924 6. Semaphore or Mutex? Still confused on which to use : r/esp32 - Reddit, https://www.reddit.com/r/esp32/comments/sxfzqp/semaphore_or_mutex_still_confused_on_which_to_use/ 7. ESP32 WebSocket Server: Control Outputs (Arduino IDE) | Random ..., https://randomnerdtutorials.com/esp32-websocket-server-arduino/ 8. Led Control With ESP Webserver : 5 Steps - Instructables, https://www.instructables.com/Led-Control-With-ESP-Webserver/ 9. ESP32 Asynchronous web server - MyHomeThings, https://myhomethings.eu/en/esp32-asynchronous-web-server/ 10. [Full Tutorial] ESP32 Remote Control with WebSocket - RNT Lab, https://rntlab.com/question/full-tutorial-esp32-remote-control-with-websocket/ 11. ESP32 Web Server Using Bootstrap 4 and WebSockets - Hackster.io, https://www.hackster.io/donowak/esp32-web-server-using-bootstrap-4-and-websockets-0bf950 12. littleFS - Random Nerd Tutorials, https://randomnerdtutorials.com/?s=littleFS 13. ESP32: Upload Files to LittleFS using Arduino IDE - Random Nerd Tutorials, https://randomnerdtutorials.com/esp32-littlefs-arduino-ide/ 14. ESP32 Web Server: Display Sensor Readings in Gauges - Random Nerd Tutorials, https://randomnerdtutorials.com/esp32-web-server-gauges/ 15. ESP32: Create a Wi-Fi Manager (AsyncWebServer library) | Random Nerd Tutorials, https://randomnerdtutorials.com/esp32-wi-fi-manager-asyncwebserver/ 16. IoT WiFi Credential Manager With ESP32 : 7 Steps - Instructables, https://www.instructables.com/IoT-WiFi-Credential-Manager-With-ESP32/ 17. WiFiManager with ESP32 - Stop Hard-coding WiFi Credentials! - YouTube, https://www.youtube.com/watch?v=VnfX9YJbaU8 18. How to Set an ESP32 Access Point (AP) for Web Server - Random Nerd Tutorials, https://randomnerdtutorials.com/esp32-access-point-ap-web-server/ 19. Different way to add local wifi credentials? : r/esp32 - Reddit, https://www.reddit.com/r/esp32/comments/1h4wbx4/different_way_to_add_local_wifi_credentials/ 20. Putting the ESP32 microcontroller on the Internet - Ngrok, https://ngrok.com/blog-post/putting-the-esp32-microcontroller-on-the-internet 21. ESP32/ESP8266 Web Server HTTP Authentication (Username and Password Protected), https://randomnerdtutorials.com/esp32-esp8266-web-server-http-authentication/ 22. ESP32 ESP8266 Web Server HTTP Authentication: Password Protection, https://microcontrollerslab.com/esp32-esp8266-http-authentication-web-server-password-protected/ 23. Security Overview - ESP32 - — ESP-IDF Programming Guide v5.5 documentation, https://docs.espressif.com/projects/esp-idf/en/stable/esp32/security/security.html 24. ESP32 Email Alert Based on Temperature Threshold (change values on web server), https://randomnerdtutorials.com/esp32-email-alert-temperature-threshold/ 25. ESP32 Send Emails using SMTP Server: HTML, Text, Attachments ..., https://randomnerdtutorials.com/esp32-send-email-smtp-server-arduino-ide/ 26. ESP32: Send Pushover Notifications (Arduino IDE) | Random Nerd ..., https://randomnerdtutorials.com/esp32-pushover-notifications-arduino/ 27. ESP32 MQTT

Publish Subscribe with Arduino IDE | Random Nerd ..., https://randomnerdtutorials.com/esp32-mqtt-publish-subscribe-arduino-ide/ 28. ESP32 and AWS IoT Tutorial - AgileVision.io, https://agilevision.io/blog/esp32-and-aws-iot-tutorial/ 29. Building an AWS IoT Core device using AWS Serverless and an ESP32, https://aws.amazon.com/blogs/compute/building-an-aws-iot-core-device-using-aws-serverless-and-an-esp32/