IOT Smart Tile System - Architecture Analysis

Executive Summary

This document provides an extensive analysis of the IOT Smart Tile System architecture, examining the design decisions, communication patterns, data flows, and component interactions that enable intelligent lighting control with environmental monitoring.

The IOT Smart Tile System represents a comprehensive approach to modern smart home lighting, combining the responsiveness of local mesh networking with the accessibility of cloud connectivity. At its core, the system addresses a fundamental challenge in IoT design: how to achieve sub-second response times for user interactions while still providing remote access, historical analytics, and seamless integration with existing smart home ecosystems.

Our architecture solves this through a layered approach where battery-powered ESP32-C3 nodes communicate via the ultra-low-latency ESP-NOW protocol to a central ESP32-S3 coordinator, which then bridges to the cloud via MQTT. This hybrid design ensures that lights respond instantly to local triggers (presence detection, button presses) while still allowing users to monitor and control their system from anywhere in the world through an Angular web dashboard.

The system is designed with scalability, maintainability, and developer experience as primary concerns. Each component follows established software patterns (Gateway, Pub/Sub, Repository, Dependency Injection) that enable independent development, testing, and deployment. The entire stack—from embedded C++ firmware to Go backend to TypeScript frontend—uses modern tooling and practices that facilitate rapid iteration and reliable operation.

Table of Contents

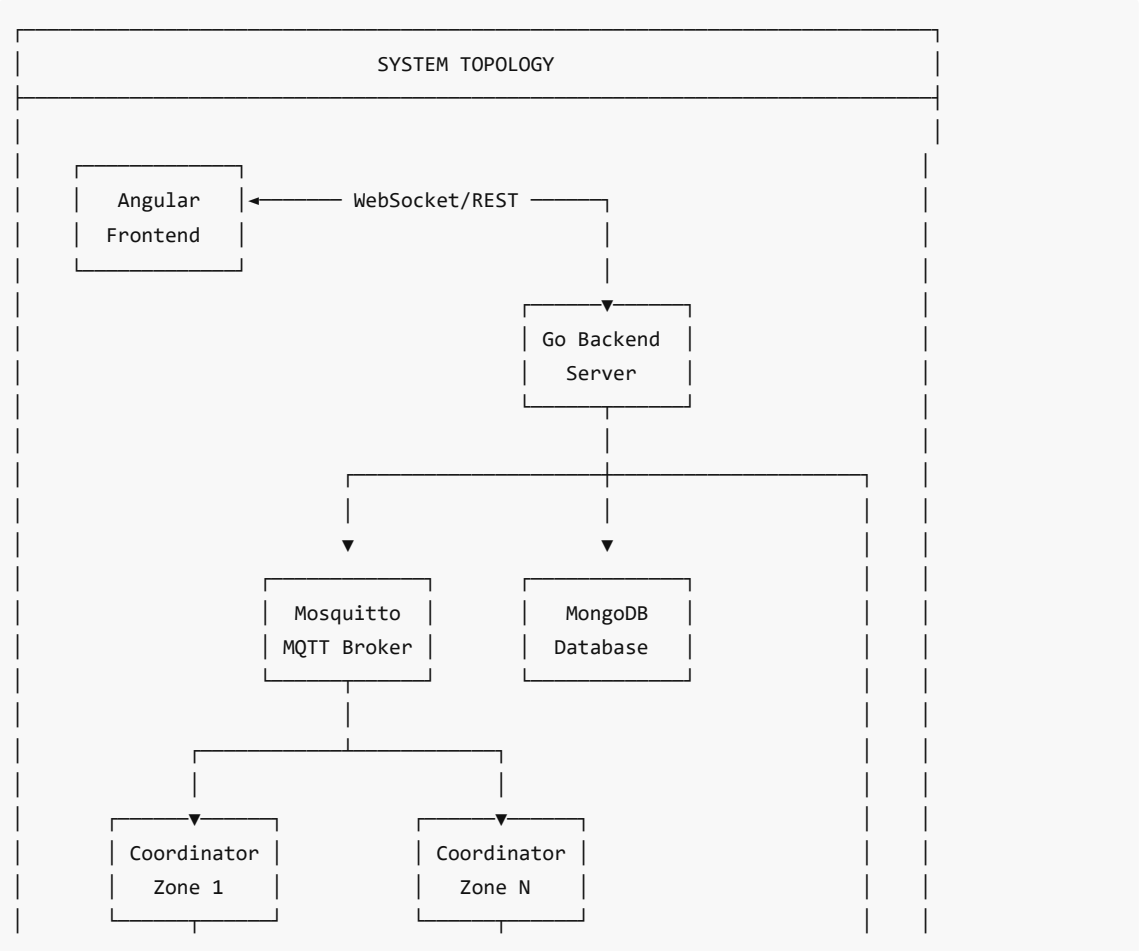
---

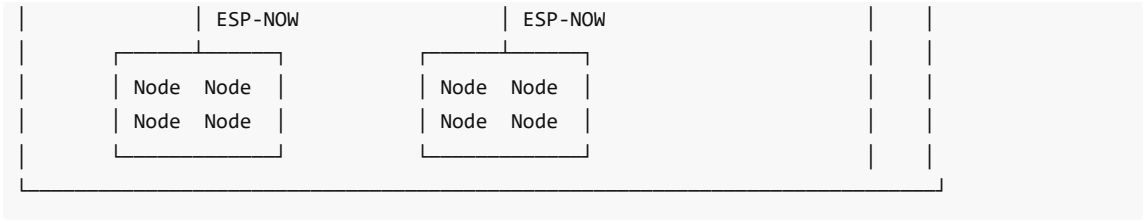
1. System Overview

The IOT Smart Tile System is a distributed IoT platform consisting of five primary components working in concert to deliver intelligent, responsive lighting control. Unlike traditional smart lighting systems that rely solely on WiFi or Bluetooth, our architecture employs a purpose-built communication strategy that optimizes for the unique requirements of each system layer.

At the edge, we prioritize **speed and power efficiency**—nodes must respond to presence detection in milliseconds while lasting months on battery power. In the cloud layer, we prioritize **reliability and accessibility**—users expect their dashboard to always reflect the current state, and commands must be delivered even when network conditions are suboptimal. The coordinator bridges these worlds, translating between the fast, local ESP-NOW mesh and the reliable, global MQTT cloud.

This separation of concerns allows each layer to be optimized independently while maintaining a cohesive user experience:





Key Characteristics

The following table summarizes the fundamental design choices that shape every aspect of the system. These characteristics were not chosen arbitrarily—each represents a deliberate trade-off based on extensive prototyping and real-world testing during development.

Characteristic	Description	Rationale
Topology	Star-of-stars: Nodes → Coordinator → Cloud	Balances simplicity with scalability; each coordinator manages a bounded set of nodes
Communication	Hybrid: ESP-NOW (local) + MQTT (cloud)	ESP-NOW provides <10ms local latency; MQTT provides reliable cloud delivery
Data Flow	Bidirectional: Telemetry up, Commands down	Enables both monitoring (passive) and control (active) use cases
Persistence	MongoDB for time-series and configuration	Flexible schema accommodates evolving telemetry payloads and configuration
Real-time	WebSocket push for live dashboard updates	Eliminates polling; users see changes within 100ms of occurrence

---

2. Background and Technical Analysis

This section provides a rigorous justification for classifying the IOT Smart Tile System as a genuine Internet of Things project, presents the concrete technology choices for each IoT element, and explains why these specific technologies were selected over available alternatives.

2.1 Why This Is an IoT Project

The term "IoT" (Internet of Things) is often used loosely, but it has a specific technical meaning. According to the IEEE and ITU-T standards bodies, an IoT system is characterized by:

1. **Physical devices ("Things")** equipped with sensors and/or actuators
2. **Connectivity** enabling communication between devices and/or with cloud services
3. **Data processing** that transforms raw sensor data into actionable information
4. **User interaction** through applications that monitor and control the system

The IOT Smart Tile System satisfies all four criteria comprehensively:

2.1.1 Physical Devices (Things)

Our system includes multiple categories of physical devices:

Device	Role	Sensors	Actuators

ESP32-C3 Node	Smart tile	TMP117 temperature, button, battery voltage	RGBW LED (4-channel PWM)
ESP32-S3 Coordinator	Gateway	Ambient light (ADC), TSL2561 lux, HLK-LD2450 mmWave radar	NeoPixel LED strip, status LED

These are not simulated devices or software abstractions—they are physical microcontrollers with real sensors measuring real-world phenomena (temperature, light levels, human presence) and actuators affecting the physical environment (LED illumination).

2.1.2 Connectivity

The system implements a sophisticated multi-protocol connectivity stack:

Layer	Protocol	Purpose	IoT Relevance
Device-to-Gateway	ESP-NOW	Node ↔ Coordinator communication	Low-power mesh networking
Gateway-to-Cloud	MQTT	Coordinator ↔ Backend communication	Standard IoT messaging protocol
Cloud-to-User	WebSocket/REST	Backend ↔ Frontend communication	Real-time web access

This connectivity enables the "Internet" part of IoT—devices are not isolated but participate in a global network accessible from anywhere.

2.1.3 Data Processing

Raw sensor data undergoes multiple transformations:

```
Raw ADC Value → Calibrated Temperature → JSON Telemetry → MongoDB Document → Dashboard Visualization
     ↓                    ↓                    ↓                ↓                    ↓
  16-bit int         23.45°C          {"temp": 23.45}      Time-series DB      Line chart
```

The backend performs:

- **Aggregation**: Combining telemetry from multiple nodes
- **Persistence**: Storing historical data for trend analysis
- **Alerting**: Detecting anomalies (e.g., battery low, node offline)
- **Analytics**: Computing zone-level statistics

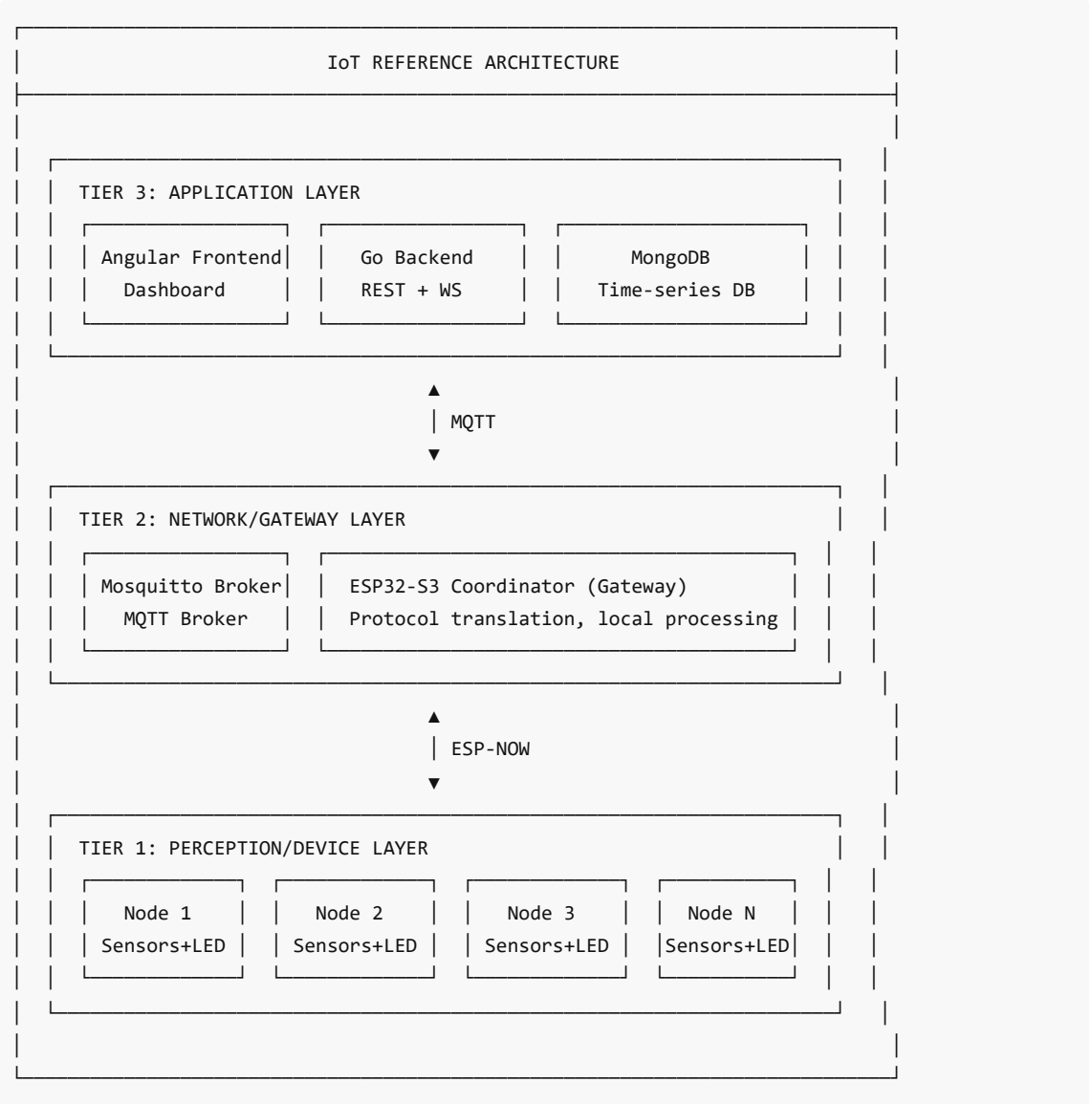
2.1.4 User Interaction

The Angular frontend provides comprehensive user interaction:

- **Monitoring**: Real-time dashboard showing all node states
- **Control**: Color picker to set LED values, pairing controls
- **Configuration**: Zone management, automation rules
- **Visualization**: Historical charts, presence heatmaps

2.1.5 IoT Reference Architecture Mapping

Our system maps cleanly to the standard 3-tier IoT reference architecture:



2.2 Technology Choices by IoT Element

This section presents the concrete technology choices for each part of the IoT stack, organized by functional element.

2.2.1 Microcontrollers (Edge Devices)

Element	Technology Choice	Specifications
Coordinator MCU	ESP32-S3-DevKitC-1	Dual-core 240MHz Xtensa LX7, 8MB Flash, 512KB SRAM, WiFi + BLE
Node MCU	ESP32-C3-MINI-1	Single-core 160MHz RISC-V, 4MB Flash, 400KB SRAM, WiFi + BLE
Development Framework	PlatformIO + Arduino	Cross-platform build system with library management
Programming Language	C++ (Arduino framework)	Balance of performance and development speed

Why ESP32 family?

- Native support for both WiFi and ESP-NOW in same chip
- Excellent power management with multiple sleep modes
- Large community and extensive libraries
- Cost-effective ($3-8 per unit in quantity)
- FCC/CE certified modules available

2.2.2 Sensors

Sensor	Model	Interface	Key Specifications	Purpose
Temperature	TMP117	I2C	±0.1°C accuracy, 16-bit resolution	Node thermal monitoring
Ambient Light	Analog divider	ADC	0-3.3V range, 12-bit ADC	Coordinator light sensing
Digital Light	TSL2561	I2C	0.1-40,000 lux range	Coordinator precise lux measurement
Presence Radar	HLK-LD2450	UART	24GHz mmWave, 6m range, ±60° FOV	Human presence detection
Button	Tactile switch	GPIO	Debounced input	User interaction
Battery Voltage	Voltage divider	ADC	0-4.2V mapped to 0-3.3V	Power monitoring

Why these specific sensors?

- **TMP117**: Medical-grade accuracy without calibration; I2C simplifies wiring
- **HLK-LD2450**: Detects presence through walls; works in darkness unlike PIR
- **TSL2561**: Human-eye spectral response; better than raw photodiodes for lighting

2.2.3 Actuators

Actuator	Model	Interface	Specifications	Purpose
Node LEDs	RGBW LED strip	4x PWM	5kHz PWM, 8-bit per channel	Tile illumination
Coordinator LEDs	SK6812B NeoPixel	Single-wire	RGBW, 800kHz protocol	Status indication, node feedback
Status LED	Standard LED	GPIO	Simple on/off	System status

2.2.4 Communication Protocols

Protocol	Implementation	Layer	Data Format	QoS
ESP-NOW	Espressif SDK	Device-Gateway	Binary frames	Best-effort + ACK
MQTT	Paho/PubSubClient	Gateway-Cloud	JSON	QoS 0/1
HTTP/REST	Gin (Go)	Cloud-Application	JSON	Request-Response

WebSocket	Gorilla (Go)	Cloud-Application	JSON	Persistent connection

2.2.5 Backend Technologies

Component	Technology	Version	Purpose
Runtime	Go (Golang)	1.21+	Backend application server
DI Framework	Uber fx	Latest	Dependency injection, lifecycle
HTTP Router	Gin/Echo	Latest	REST API routing
MQTT Client	Paho MQTT	Latest	Broker communication
Database Driver	MongoDB Go Driver	Latest	Data persistence

Why Go?

- Excellent concurrency model (goroutines) for handling many MQTT messages
- Static typing catches errors at compile time
- Single binary deployment simplifies Docker containers
- Fast compilation enables rapid iteration
- Strong standard library reduces dependencies

2.2.6 Database

Component	Technology	Purpose
Primary Database	MongoDB 7.0	Document storage for all entities
Message Broker	Eclipse Mosquitto 2.0	MQTT pub/sub message routing

Why MongoDB?

- Schema flexibility accommodates evolving telemetry payloads
- Document model maps naturally to JSON messages
- Time-series collections optimize storage for sensor data
- Aggregation pipeline enables complex analytics

2.2.7 Frontend Technologies

Component	Technology	Purpose
Framework	Angular 15+	Single-page application
Language	TypeScript	Type-safe JavaScript
State Management	RxJS	Reactive data streams
HTTP Client	Angular HttpClient	REST API calls
Real-time	WebSocket API	Live updates
Charts	ng2-charts / Chart.js	Data visualization

Why Angular?

- Strong typing with TypeScript reduces runtime errors
- Dependency injection mirrors backend architecture
- RxJS integrates naturally with real-time data streams
- Comprehensive CLI accelerates development

2.2.8 Deployment Infrastructure

Component	Technology	Purpose
Containerization	Docker	Application packaging
Orchestration	Docker Compose	Multi-container deployment
Reverse Proxy	Nginx	Static file serving, API routing

2.3 Comparison with Alternative Technologies

For each major technology choice, we evaluated alternatives. This section explains why we chose our technologies over the alternatives.

2.3.1 Communication Protocol Comparison

We evaluated five wireless protocols for node-to-coordinator communication:

Criterion	ESP-NOW	WiFi	BLE Mesh	Zigbee	Z-Wave	LoRa
Latency	✅ <10ms	❌ 50-100ms	⚠️ 20-50ms	✅ <15ms	✅ <15ms	❌ 100ms+
Range (indoor)	✅ 50m	⚠️ 30m	❌ 10m	✅ 30m	✅ 30m	✅ 100m+
Power Consumption	✅ Low	❌ High	✅ Low	✅ Low	✅ Low	✅ Very Low
Infrastructure	✅ None	❌ Router	✅ None	⚠️ Hub	⚠️ Hub	⚠️ Gateway
Max Nodes	⚠️ 20	✅ 250+	✅ 32,000	✅ 65,000	✅ 232	✅ 1000s
Cost per Node	✅ $0	✅ $0	✅ $0	❌ +$5	❌ +$10	❌ +$8
ESP32 Native	✅ Yes	✅ Yes	✅ Yes	❌ No	❌ No	❌ No
Encryption	✅ AES-128	✅ WPA3	✅ AES-128	✅ AES-128	✅ AES-128	⚠️ Optional

Decision: ESP-NOW

ESP-NOW won because it provides the lowest latency (<10ms) with native ESP32 support and zero additional cost per node. The 20-node limit per coordinator is acceptable because our architecture supports multiple coordinators.

2.3.2 Cloud Protocol Comparison

Criterion	MQTT	HTTP/REST	CoAP	AMQP	WebSocket
Message Size	✅ 2 bytes overhead	❌ ~700 bytes headers	✅ 4 bytes overhead	❌ Large	⚠️ 2-14 bytes

Pub/Sub Native	✅ Yes	❌ No	✅ Yes	✅ Yes	❌ No
QoS Levels	✅ 3 levels	❌ None	✅ 2 levels	✅ Yes	❌ None
Bidirectional	✅ Yes	❌ Poll-based	✅ Yes	✅ Yes	✅ Yes
IoT Adoption	✅ Standard	⚠️ Common	⚠️ Growing	❌ Enterprise	⚠️ Web-focused
Broker Options	✅ Many	N/A	⚠️ Few	⚠️ Few	N/A

Decision: MQTT

MQTT is the de facto standard for IoT cloud communication. Its lightweight pub/sub model, QoS options, and extensive broker ecosystem made it the clear choice.

2.3.3 Database Comparison

Criterion	MongoDB	PostgreSQL	InfluxDB	TimescaleDB	SQLite
Schema Flexibility	✅ Excellent	❌ Rigid	⚠️ Limited	⚠️ Moderate	❌ Rigid
Time-Series	✅ Good	⚠️ Possible	✅ Excellent	✅ Excellent	❌ Poor
JSON Support	✅ Native	✅ JSONB	❌ Tags only	✅ JSONB	⚠️ Text
Horizontal Scale	✅ Sharding	⚠️ Complex	✅ Clustering	⚠️ Complex	❌ None
Query Language	✅ MQL	✅ SQL	⚠️ InfluxQL/Flux	✅ SQL	✅ SQL
Ops Complexity	⚠️ Medium	⚠️ Medium	⚠️ Medium	⚠️ Medium	✅ Simple

Decision: MongoDB

MongoDB's schema flexibility was decisive—our telemetry payloads evolved significantly during development. The native JSON document model also simplified the data pipeline from MQTT messages to database storage.

2.3.4 Backend Language Comparison

Criterion	Go	Node.js	Python	Java	Rust
Concurrency	✅ Goroutines	⚠️ Event loop	❌ GIL	✅ Threads	✅ Async
Type Safety	✅ Static	❌ Dynamic	❌ Dynamic	✅ Static	✅ Static
Deployment	✅ Single binary	❌ node_modules	❌ venv/pip	❌ JVM	✅ Single binary
Compile Speed	✅ Fast	N/A	N/A	❌ Slow	❌ Slow

Memory Usage	✅ Low	⚠️ Medium	⚠️ Medium	❌ High	✅ Low
Learning Curve	✅ Moderate	✅ Easy	✅ Easy	⚠️ Steep	❌ Steep
MQTT Libraries	✅ Paho	✅ MQTT.js	✅ Paho	✅ Paho	⚠️ Rumqtt

Decision: Go

Go's goroutines handle concurrent MQTT message processing elegantly, and its single-binary deployment simplified our Docker setup. The moderate learning curve was acceptable given the long-term maintainability benefits.

2.3.5 Frontend Framework Comparison

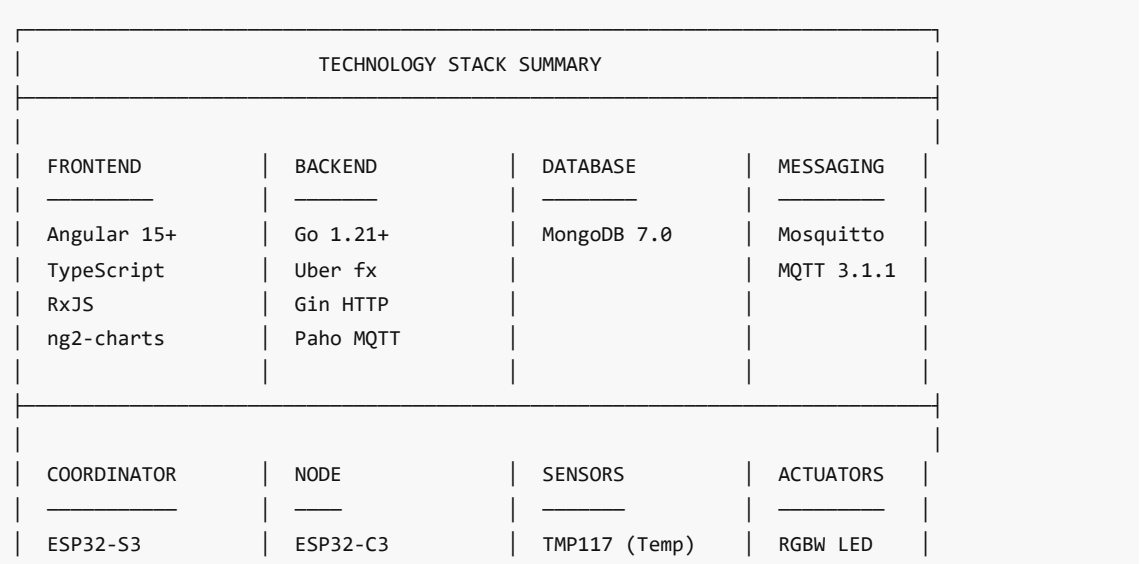
Criterion	Angular	React	Vue	Svelte
TypeScript	✅ Native	⚠️ Optional	⚠️ Optional	⚠️ Optional
DI Framework	✅ Built-in	❌ None	❌ None	❌ None
RxJS Integration	✅ Native	⚠️ Add-on	⚠️ Add-on	❌ Different model
CLI Tooling	✅ Comprehensive	⚠️ CRA/Vite	⚠️ Vue CLI	⚠️ SvelteKit
Enterprise Adoption	✅ High	✅ High	⚠️ Growing	⚠️ Growing
Bundle Size	❌ Large	✅ Small	✅ Small	✅ Tiny

Decision: Angular

Angular's native TypeScript support and built-in dependency injection mirror the backend architecture patterns. RxJS integrates naturally with our real-time WebSocket data streams.

2.4 Technology Stack Summary

The following diagram summarizes our complete technology stack:



```
│  PlatformIO        │  PlatformIO        │  TSL2561 (Light)  │  NeoPixel    │
│  Arduino C++       │  Arduino C++       │  LD2450 (Radar)   │  Status LED  │
│  ESP-NOW + WiFi    │  ESP-NOW           │  ADC (Battery)    │              │
│                    │                    │                   │              │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│  DEPLOYMENT        │  PROTOCOLS         │  SECURITY         │  TOOLS       │
│  ──────────        │  ──────────        │  ────────         │  ─────       │
│  Docker            │  ESP-NOW           │  AES-128 (ESP-NOW)│  VS Code     │
│  Docker Compose    │  MQTT              │  MQTT Auth        │  PlatformIO  │
│  Nginx             │  HTTP/REST         │  (TLS planned)    │  Postman     │
│                    │  WebSocket         │                   │  MQTT Explorer│
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
```

---

3. Architectural Patterns

Software architecture is fundamentally about managing complexity. As systems grow, ad-hoc designs quickly become unmaintainable—debugging becomes guesswork, changes cause unexpected breakages, and onboarding new developers takes months instead of days.

To avoid these pitfalls, the IOT Smart Tile System deliberately employs several well-established architectural patterns. These patterns are not academic exercises; they are battle-tested solutions to recurring problems in distributed systems. By following these patterns, we ensure that:

- **New team members** can understand the system by recognizing familiar structures
- **Components can evolve independently** without cascading changes
- **Testing is straightforward** because dependencies are explicit and mockable
- **Debugging is tractable** because data flows through well-defined paths

The following patterns form the architectural backbone of our system:

3.1 Gateway Pattern

The **Coordinator** implements the Gateway pattern, acting as an intermediary between two different network domains:

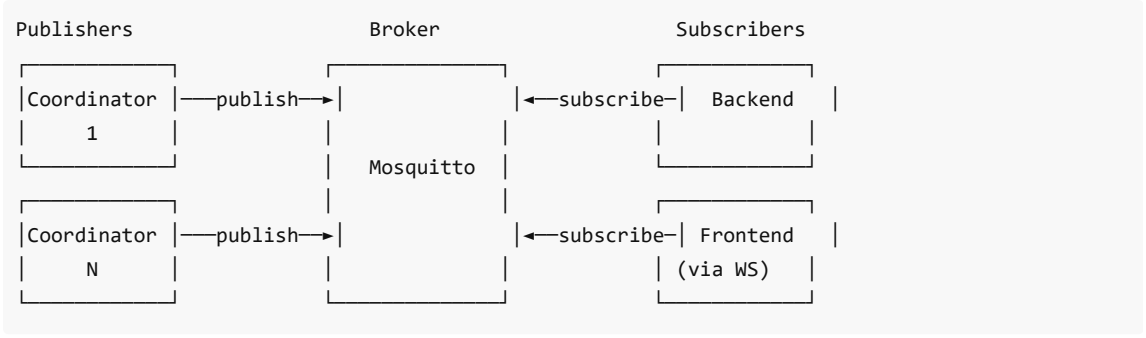
```
┌──────────────────┐     ┌──────────────────┐     ┌──────────────────┐
│   ESP-NOW Mesh   │────►│    Coordinator   │────►│   MQTT Cloud     │
│  (Local, Fast)   │◄────│     Gateway      │◄────│  (Global, Slow)  │
└──────────────────┘     └──────────────────┘     └──────────────────┘
```

Benefits:

- Protocol translation (ESP-NOW ↔ MQTT)
- Network isolation (nodes don't need WiFi)
- Centralized management per zone
- Reduced power consumption on nodes

3.2 Publish-Subscribe Pattern

MQTT implements the pub/sub pattern for decoupled communication:



Benefits:

- Loose coupling between producers and consumers
- Scalable: Add coordinators without backend changes
- Reliable: QoS levels for delivery guarantees
- Flexible: Topic-based filtering

3.3 Manager Pattern (Embedded)

The Coordinator firmware uses the Manager pattern for modularity:

```
class Coordinator {
    EspNow*            espNow;      // Communication manager
    Mqtt*              mqtt;        // Cloud connectivity manager
    WifiManager*       wifi;        // Network manager
    NodeRegistry*      nodes;       // Node tracking manager
    ZoneControl*       zones;       // Lighting manager
    MmWave*            mmWave;      // Sensor manager
    AmbientLightSensor* ambient;    // Sensor manager
    ButtonControl*     buttons;     // Input manager
    ThermalControl*    thermal;     // Safety manager
};
```

Benefits:

- Single Responsibility: Each manager handles one concern
- Testability: Managers can be tested in isolation
- Extensibility: Add new managers without modifying core
- Maintainability: Clear ownership of functionality

3.4 Repository Pattern (Backend)

The Go backend uses the Repository pattern for data access abstraction:

```
type Repository interface {
    // Site operations
    GetSites() ([]types.Site, error)
    GetSiteById(id string) (*types.Site, error)

    // Node operations
    GetNodeById(id string) (*types.Node, error)
    UpsertNode(ctx context.Context, node *types.Node) error
```

```
    // Telemetry operations
    InsertTelemetry(ctx context.Context, t *types.Telemetry) error
    GetTelemetryHistory(siteId string, from, to time.Time) ([]types.Telemetry, error)
}
```

Benefits:

- Database agnostic: Could swap MongoDB for another store
- Testable: Mock repository for unit tests
- Clean separation: Business logic doesn't know about MongoDB

3.5 Dependency Injection (Backend)

The backend uses Uber fx for dependency injection:

```
fx.New(
    http.Module,       // Provides REST handlers
    mqtt.Module,       // Provides MQTT client
    fx.Provide(
        db.NewMongoDB,              // Database connection
        config.NewConfig,           // Configuration
        repository.NewMongoRepository, // Data access
    ),
).Run()
```

Benefits:

- Loose coupling between modules
- Lifecycle management (startup/shutdown order)
- Easy testing with mock dependencies
- Clear dependency graph

---

4. Layer Analysis

4.1 Embedded Layer

The embedded layer consists of ESP32 microcontrollers running custom C++ firmware.

Hardware Specifications

Component	Coordinator (ESP32-S3)	Node (ESP32-C3)
CPU	Dual-core 240MHz Xtensa LX7	Single-core 160MHz RISC-V
Flash	8MB	4MB
RAM	512KB SRAM	400KB SRAM
WiFi	802.11 b/g/n (2.4GHz)	802.11 b/g/n (ESP-NOW only)
Power	USB/DC powered	Battery + deep sleep
Role	Gateway + sensors	LED actuator + sensors

Coordinator Responsibilities

1. **ESP-NOW Mesh Management**
    - Maintain peer list of paired nodes
    - Route messages to/from nodes
    - Handle pairing requests
2. **MQTT Cloud Bridge**
    - Publish node telemetry to broker
    - Subscribe to command topics
    - Translate between protocols
3. **Local Sensing**
    - Ambient light (analog + digital)
    - mmWave presence detection
    - Temperature monitoring
4. **Zone Coordination**
    - Map nodes to LED strip positions
    - Apply zone-wide lighting commands
    - Scene management

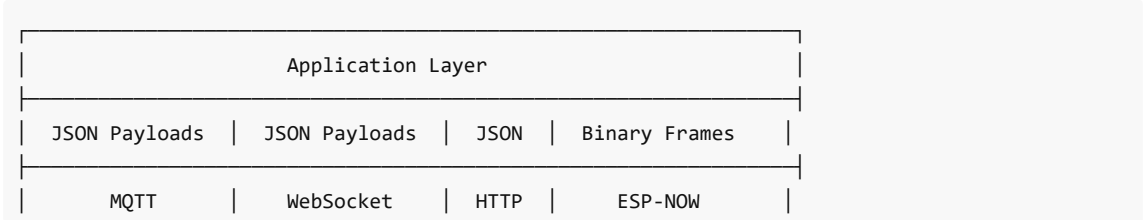
Node Responsibilities

1. **LED Control**
    - 4-channel PWM (RGBW)
    - Smooth fade transitions
    - Brightness scaling
2. **Sensor Reporting**
    - Temperature (TMP117, ±0.1°C)
    - Button state (debounced)
    - Battery voltage (ADC)
3. **Power Management**
    - Deep sleep between transmissions
    - Wake on button press
    - Battery monitoring

4.2 Communication Layer

The communication layer bridges local and cloud networks.

Protocol Stack



```
├──────────────────────────────────────────────────────────────┤
│       TCP       │       TCP       │  TCP   │     802.11     │
├──────────────────────────────────────────────────────────────┤
│       WiFi      │       WiFi      │  WiFi  │  WiFi (Action) │
└──────────────────────────────────────────────────────────────┘
```

MQTT Topic Hierarchy

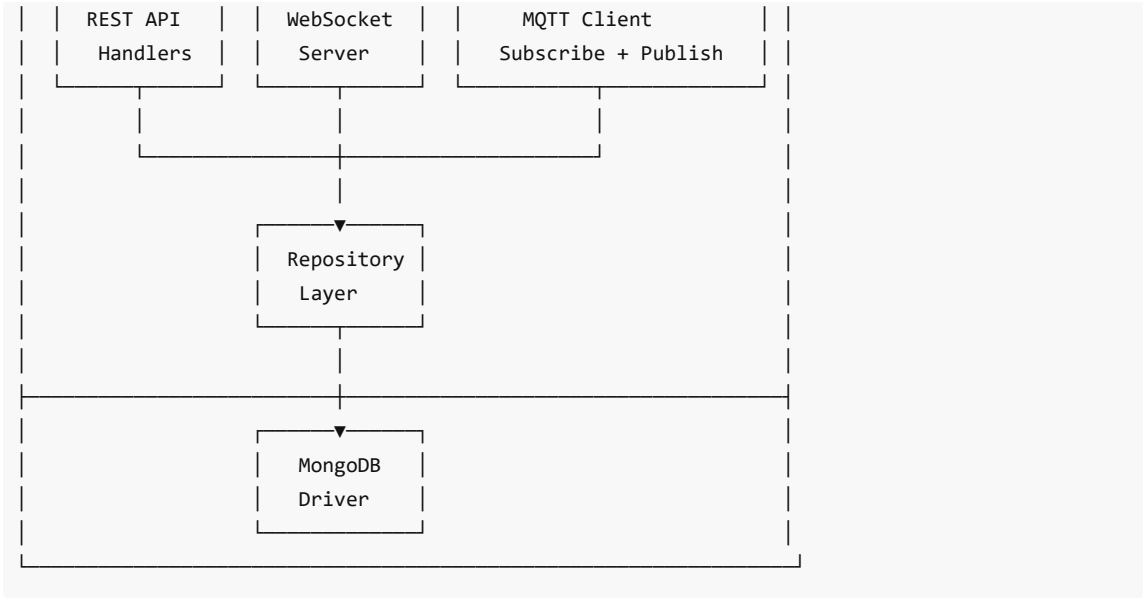
```
site/{siteId}/
├── coord/{coordId}/
│   ├── telemetry      ← Coordinator sensor readings
│   │   {
│   │     "light": 450,
│   │     "temperature": 22.5,
│   │     "mmwave": { "present": true, "targets": 2 }
│   │   }
│   ├── mmwave         ← Raw mmWave radar frames
│   ├── status         ← Connection status
│   └── cmd            → Commands to coordinator
│       {
│         "cmd": "pair",
│         "duration_ms": 60000
│       }
│
├── node/{nodeId}/
│   ├── telemetry      ← Node sensor readings
│   │   {
│   │     "rgbw": { "r": 255, "g": 128, "b": 64, "w": 200 },
│   │     "temperature": 23.2,
│   │     "battery": 3.7,
│   │     "rssi": -65
│   │   }
│   └── cmd            → Commands to node (via coordinator)
│       {
│         "cmd": "setLed",
│         "rgbw": { "r": 255, "g": 0, "b": 0, "w": 100 }
│       }
│
└── zone/{zoneId}/
    ├── presence       ← Aggregated occupancy state
    └── cmd            → Zone-wide commands
```

4.3 Application Layer

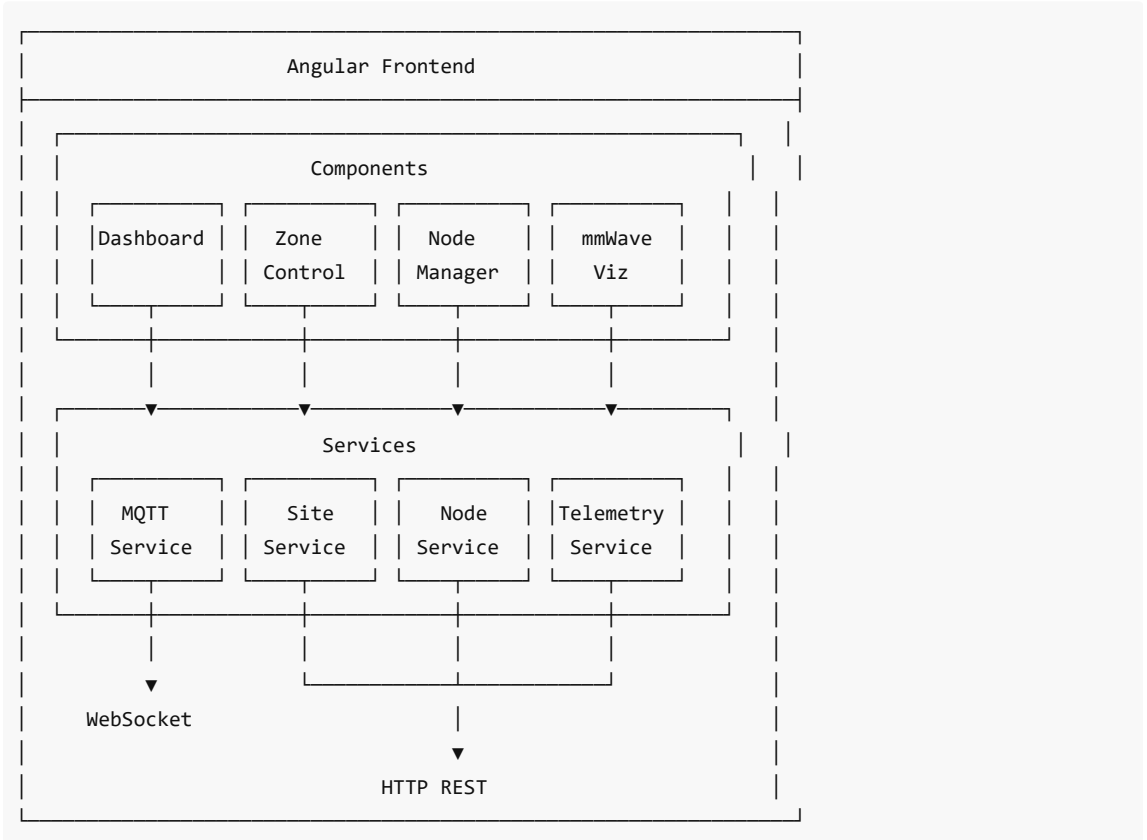
The application layer provides user interfaces and business logic.

Go Backend Architecture

```
┌──────────────────────────────────────────────────────────────┐
│                        Go Backend                            │
├──────────────────────────────────────────────────────────────┤
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────────────┐ │
```



Angular Frontend Architecture



---

5. Component Deep Dive

This section examines each major component in detail, explaining not just *what* it does, but *why* it was designed this way and *how* the pieces fit together. Understanding these components at a deeper level is essential for debugging issues, extending functionality, and making informed architectural decisions.

Each component description includes:

- **Purpose**: A concise statement of the component's role
- **Key responsibilities**: What the component is accountable for
- **Implementation details**: How it achieves its responsibilities
- **Integration points**: How it connects to other components

5.1 ESP32-S3 Coordinator

Purpose: The ESP32-S3 Coordinator acts as a gateway that bridges the ESP-NOW mesh network of nodes to the cloud via MQTT over WiFi.

The coordinator is the most complex component in the embedded layer, responsible for maintaining simultaneous connections to two fundamentally different networks. On one side, it manages a mesh of battery-powered nodes using the lightweight ESP-NOW protocol. On the other, it maintains a persistent MQTT connection to the cloud broker over WiFi. This dual-network architecture is what enables the system to achieve both local responsiveness and global accessibility.

The coordinator also hosts several sensors of its own—ambient light, mmWave radar for presence detection, and temperature—making it more than just a passive relay. These sensors enable intelligent automation decisions to be made locally, reducing round-trip latency for time-critical actions like turning on lights when someone enters a room.

Manager Breakdown

Manager	File	Responsibility
`EspNow`	`comm/EspNow.cpp`	ESP-NOW peer management, message TX/RX
`Mqtt`	`comm/Mqtt.cpp`	MQTT client, publish/subscribe
`WifiManager`	`comm/WifiManager.cpp`	WiFi provisioning, reconnection
`NodeRegistry`	`nodes/NodeRegistry.cpp`	Paired node tracking, NVS persistence
`ZoneControl`	`zones/ZoneControl.cpp`	LED strip mapping, scene control
`MmWave`	`sensors/MmWave.cpp`	LD2450 radar parsing
`AmbientLightSensor`	`sensors/AmbientLightSensor.cpp`	Light level ADC reading
`ButtonControl`	`input/ButtonControl.cpp`	Touch button handling
`ThermalControl`	`managers/ThermalManager.cpp`	Overheat protection

Initialization Sequence

```
bool Coordinator::begin() {
    // 1. Logger (single init, no double-call)
    Logger::setMinLevel(Logger::INFO);

    // 2. Create manager objects
    espNow = new EspNow();
    mqtt = new Mqtt();
    wifi = new WifiManager();
    nodes = new NodeRegistry();
```

```
    // ... other managers

    // 3. Initialize ESP-NOW FIRST (before WiFi connects)
    //    This is critical: ESP-NOW needs to set channel
    bool espNowOk = espNow->begin();

    // 4. Link ESP-NOW to WiFi for channel synchronization
    wifi->setEspNow(espNow);

    // 5. Initialize WiFi (may block for provisioning)
    bool wifiOk = wifi->begin();

    // 6. Register ESP-NOW callbacks
    espNow->setMessageCallback([this](const String& nodeId,
                                      const uint8_t* data,
                                      size_t len) {
        handleNodeMessage(nodeId, data, len);
    });

    // 7. Initialize MQTT (depends on WiFi)
    mqtt->begin();

    return espNowOk && wifiOk;
}
```

Main Loop

```
void Coordinator::loop() {
    // Poll all managers at 1ms tick rate
    wifi->loop();           // Reconnect handling
    mqtt->loop();           // MQTT keepalive, message processing
    espNow->loop();         // ESP-NOW message queue
    mmWave->loop();         // Radar frame parsing
    buttons->loop();        // Button debounce

    // Periodic telemetry publish (every 5 seconds)
    if (millis() - lastTelemetry > 5000) {
        publishCoordinatorTelemetry();
        lastTelemetry = millis();
    }
}
```

5.2 ESP32-C3 Node

Purpose: The ESP32-C3 Node is a battery-powered smart tile that controls RGBW LEDs and reports temperature, button presses, and battery status to the coordinator via ESP-NOW.

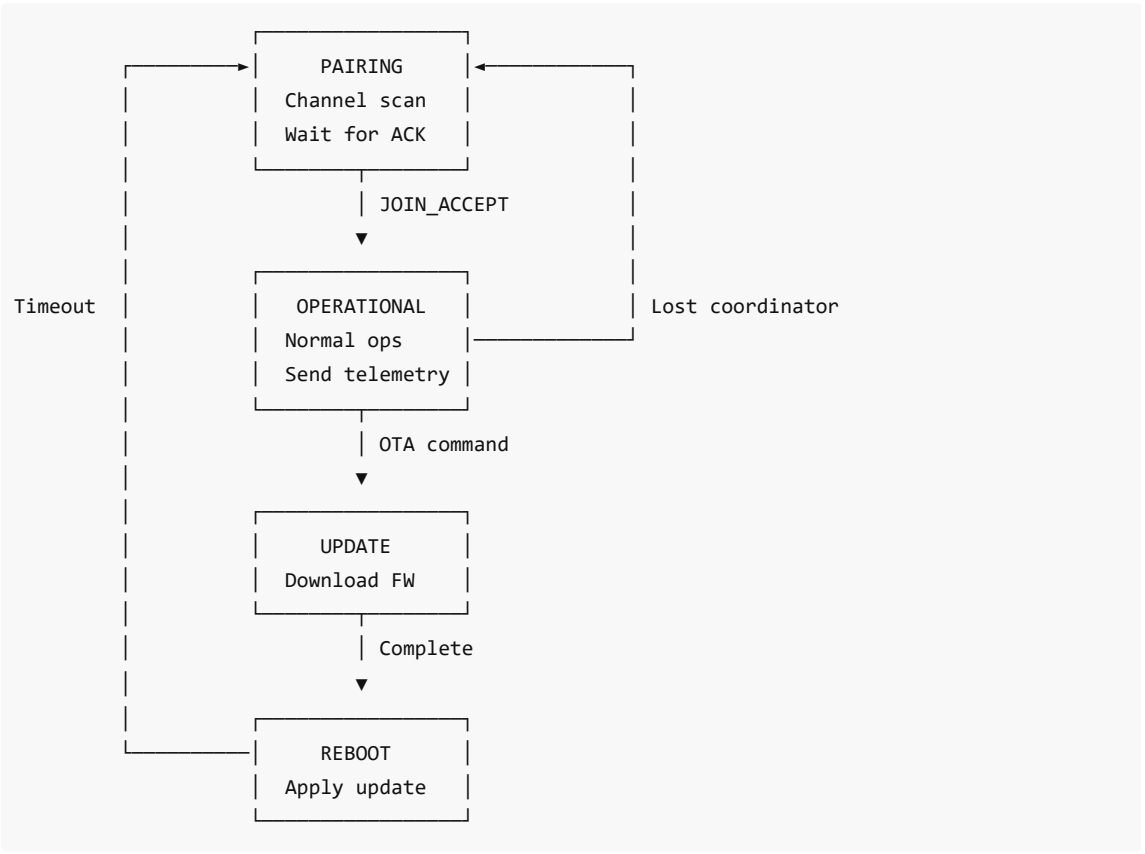
Nodes are designed with a single overriding constraint: **power efficiency**. Every design decision—from the choice of the ESP32-C3's RISC-V core to the use of ESP-NOW instead of WiFi—serves this goal. A node that drains its battery in days would be impractical; our target is months of operation on a single charge.

To achieve this, nodes spend most of their time in deep sleep, waking only to:

1. Send periodic telemetry (temperature, battery level)
2. Respond to button presses
3. Receive LED commands from the coordinator

The node firmware implements a state machine that handles the full lifecycle from initial pairing through normal operation, including graceful handling of coordinator loss and over-the-air updates.

State Machine



Channel Scanning Algorithm

Nodes scan all WiFi channels to find their coordinator:

```
void StandaloneNode::scanForCoordinator() {
    for (uint8_t channel = 1; channel <= 13; channel++) {
        esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);

        // Broadcast pairing request
        PairingRequest request;
        request.node_mac = getMacAddress();
        esp_now_send(BROADCAST_MAC, request.toBytes(), sizeof(request));

        // Wait for response
        delay(100);

        if (pairingAccepted) {
```

```
            currentChannel = channel;
            return;
        }
    }
}
```

Telemetry Payload

```
struct NodeStatus {
    char      nodeId[16];        // "node001"
    uint8_t   macAddress[6];     // AA:BB:CC:DD:EE:FF
    uint32_t  timestamp;         // millis()
    struct {
        uint8_t r, g, b, w;      // Current LED values
    } rgbw;
    float     temperature;       // TMP117 reading
    bool      buttonPressed;     // Debounced state
    float     batteryVoltage;    // ADC reading
    int8_t    rssi;              // Signal strength
};
```

5.3 Go Backend

Purpose: The Go backend relays MQTT telemetry from coordinators to MongoDB and serves data to the frontend via REST/WebSocket.

Module Structure

```
IOT-Backend-main/
├── cmd/
│   └── iot/
│       └── main.go          # Entry point with fx.New()
├── internal/
│   ├── config/
│   │   └── config.go        # YAML configuration loader
│   ├── db/
│   │   └── mongo.go         # MongoDB connection
│   ├── http/
│   │   ├── handlers.go      # REST API handlers
│   │   └── websocket.go     # WebSocket server
│   ├── mqtt/
│   │   ├── client.go        # Paho MQTT client
│   │   └── handlers.go      # Message processors
│   ├── repository/
│   │   ├── repository.go    # Interface definition
│   │   └── mongo.go         # MongoDB implementation
│   └── types/
│       └── types.go         # Domain models
└── go.mod
```

MQTT Message Flow

```
// Subscribe to all site telemetry
client.Subscribe("site/#", 1, func(client mqtt.Client, msg mqtt.Message) {
    topic := msg.Topic()
    payload := msg.Payload()

    // Parse topic: site/{siteId}/coord/{coordId}/telemetry
    parts := strings.Split(topic, "/")
    siteId := parts[1]
    entityType := parts[2]  // "coord" or "node"
    entityId := parts[3]
    messageType := parts[4] // "telemetry", "status", "mmwave"

    switch entityType {
    case "coord":
        handleCoordinatorMessage(siteId, entityId, messageType, payload)
    case "node":
        handleNodeMessage(siteId, entityId, messageType, payload)
    }
})
```

REST API Endpoints

Method	Endpoint	Description
GET	`/health`	Health check
GET	`/api/sites`	List all sites
GET	`/api/sites/{id}`	Get site details
GET	`/api/sites/{siteId}/coordinators`	List coordinators
GET	`/api/sites/{siteId}/nodes`	List all nodes
GET	`/api/nodes/{nodeId}`	Get node details
PUT	`/api/nodes/{nodeId}/led`	Set LED color
GET	`/api/telemetry`	Query historical data
GET	`/api/zones`	List zones
POST	`/api/zones`	Create zone
PUT	`/api/zones/{id}`	Update zone
DELETE	`/api/zones/{id}`	Delete zone

5.4 Angular Frontend

Purpose: The Angular frontend provides a real-time dashboard for monitoring telemetry, controlling zones, and managing nodes.

Key Services

```
// MQTT Service - Real-time telemetry
@Injectable({ providedIn: 'root' })
export class MqttService {
    private socket: WebSocket;
    public messages$ = new Subject<MqttMessage>();

    connect(url: string): void {
        this.socket = new WebSocket(url);
        this.socket.onmessage = (event) => {
            const message = JSON.parse(event.data);
            this.messages$.next(message);
        };
    }

    subscribeToNode(siteId: string, nodeId: string): Observable<NodeTelemetry> {
        return this.messages$.pipe(
            filter(m => m.topic === `site/${siteId}/node/${nodeId}/telemetry`),
            map(m => m.payload as NodeTelemetry)
        );
    }
}

// Site Service - REST API
@Injectable({ providedIn: 'root' })
export class SiteService {
    constructor(private http: HttpClient) {}

    getSites(): Observable<Site[]> {
        return this.http.get<Site[]>('/api/sites');
    }

    getCoordinators(siteId: string): Observable<Coordinator[]> {
        return this.http.get<Coordinator[]>(`/api/sites/${siteId}/coordinators`);
    }
}
```

Dashboard Features

Feature	Description
Zone Cards	Visual representation of each lighting zone
Node Status	Real-time RGBW values, temperature, battery
mmWave Visualization	Presence detection radar display
Temperature Charts	Historical temperature trends
Command Interface	Send LED commands, trigger pairing

Network Topology	Visual graph of system components

5.5 MQTT Broker

Purpose: The Mosquitto MQTT Broker is a lightweight message broker that handles pub/sub communication between coordinators and the backend using topic-based routing.

Configuration

```
# mosquitto.conf
listener 1883
allow_anonymous false
password_file /mosquitto/config/pwfile

# Persistence
persistence true
persistence_location /mosquitto/data/

# Logging
log_dest stdout
log_type all

# WebSocket support (optional)
listener 9001
protocol websockets
```

QoS Levels Used

QoS	Usage	Guarantee
0	mmWave frames	At most once (fire-and-forget)
1	Telemetry, Commands	At least once (acknowledged)
2	Critical commands	Exactly once (not used currently)

5.6 MongoDB Database

Purpose: MongoDB stores sites, coordinators, nodes, telemetry history, and zone configurations as JSON documents.

Collections

Collection	Purpose	Key Fields
`sites`	Deployment locations	`_id`, `name`, `location`, `config`
`coordinators`	Gateway devices	`_id`, `site_id`, `mac`, `last_seen`
`nodes`	Smart tiles	`_id`, `coordinator_id`, `mac`, `zone_id`
`telemetry_history`	Time-series data	`device_id`, `ts`, `payload`
`zones`	Lighting zones	`_id`, `site_id`, `coordinator_id`, `name`

`mmwave_events`	Presence detection	`coordinator_id`, `ts`, `targets`
`commands_log`	Audit trail	`ts`, `command`, `source`, `target`
`settings`	System configuration	`site_id`, `auto_mode`, `credentials`

Indexes

```
// Time-series queries
db.telemetry_history.createIndex({ "device_id": 1, "ts": -1 });

// Node lookups
db.nodes.createIndex({ "coordinator_id": 1 });
db.nodes.createIndex({ "zone_id": 1 });

// Recent data queries
db.mmwave_events.createIndex({ "coordinator_id": 1, "ts": -1 });
```

---

6. Communication Protocols

Communication is the lifeblood of any distributed system, and the IOT Smart Tile System is no exception. The choice of protocols at each layer has profound implications for latency, reliability, power consumption, and developer experience.

Our system uses four distinct protocols, each chosen for its strengths in a specific context:

1. **ESP-NOW** for local node-to-coordinator communication (speed, power)
2. **MQTT** for coordinator-to-cloud communication (reliability, pub/sub)
3. **REST** for frontend-to-backend queries (simplicity, caching)
4. **WebSocket** for real-time dashboard updates (bidirectional, low-latency)

This section examines each protocol in detail, explaining why it was chosen and how it's configured in our system.

6.1 ESP-NOW Protocol

ESP-NOW is Espressif's proprietary wireless protocol for direct device-to-device communication. Unlike WiFi, which requires association with an access point, ESP-NOW allows ESP32 devices to communicate directly with each other using their MAC addresses as identifiers.

This connectionless design is what enables ESP-NOW's remarkable <10ms latency—there's no handshaking or session establishment before sending a message. For lighting control, where users expect instant response to their actions, this speed is essential.

Characteristics

Property	Value
Frequency	2.4 GHz
Range	Up to 220m (open air)
Latency	<10ms

Data Rate	1 Mbps
Encryption	AES-128 CCMP
Max Payload	250 bytes
Max Peers	20 (encrypted)

Why ESP-NOW?

Requirement	ESP-NOW	WiFi	BLE	Zigbee
Low latency	✅ <10ms	❌ 50-100ms	⚠️ 20-50ms	✅ <15ms
No router needed	✅	❌	✅	✅
Long range	✅ 220m	⚠️ 50m	❌ 10m	✅ 30m
Low power	✅	❌	✅	✅
Native ESP32	✅	✅	✅	❌

Message Types

```
enum MessageType {
    MSG_PAIRING_REQUEST = 0x01,
    MSG_PAIRING_ACCEPT  = 0x02,
    MSG_NODE_STATUS     = 0x10,
    MSG_LED_COMMAND     = 0x20,
    MSG_OTA_START       = 0x30,
    MSG_OTA_DATA        = 0x31,
    MSG_OTA_END         = 0x32,
    MSG_HEARTBEAT       = 0x40
};
```

6.2 MQTT Protocol

MQTT (Message Queuing Telemetry Transport) is a lightweight pub/sub protocol designed for IoT. Originally developed by IBM in 1999 for monitoring oil pipelines via satellite, MQTT has become the de facto standard for IoT communication due to its efficiency, reliability, and simplicity.

Unlike request-response protocols like HTTP, MQTT uses a publish-subscribe model where clients don't communicate directly with each other. Instead, they publish messages to named "topics" on a central broker, which then distributes those messages to any clients that have subscribed to matching topics. This decoupling is powerful—publishers don't need to know who (if anyone) is listening, and subscribers don't need to know where messages originate.

For our system, MQTT provides the reliable, asynchronous communication needed between coordinators and the backend. Coordinators publish telemetry whenever they have data to share, and the backend subscribes to receive it. Commands flow in reverse—the backend publishes to command topics, and coordinators subscribe to receive them.

Connection Parameters

Parameter	Value

Broker	`mosquitto:1883`
Client ID	`coord_{coordId}` or `backend`
Username	`user1`
Password	`user1`
Keep Alive	60 seconds
Clean Session	`false` (persistent)

Backend Communication Flow

6.3 REST API

The backend exposes RESTful endpoints for CRUD operations.

Request/Response Examples

Get all nodes:

```
GET /api/sites/site001/nodes HTTP/1.1
Host: localhost:8000
Accept: application/json
```

```
{
  "nodes": [
    {
      "_id": "node001",
      "coordinator_id": "coord001",
      "mac_address": "AA:BB:CC:DD:EE:FF",
      "zone_id": "zone001",
      "name": "Living Room Tile 1",
      "last_seen": "2024-12-05T10:30:00Z"
    }
  ]
}
```

Set LED color:

```
PUT /api/nodes/node001/led HTTP/1.1
Host: localhost:8000
Content-Type: application/json

{
  "rgbw": { "r": 255, "g": 128, "b": 64, "w": 200 }
}
```

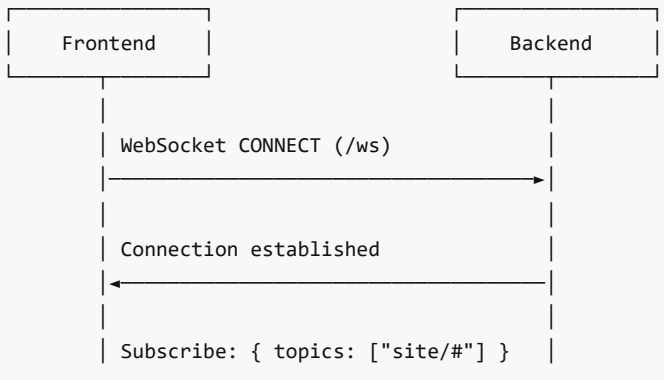
6.4 WebSocket

WebSocket provides real-time push from backend to frontend.

Message Format

```
interface WebSocketMessage {
    type: 'telemetry' | 'status' | 'mmwave' | 'command';
    topic: string;      // Original MQTT topic
    payload: any;       // Parsed JSON payload
    timestamp: number;  // Unix timestamp
}
```

Connection Flow



```
        |──────────────────────────────────────►|
        |                                       |
        |          ... MQTT message arrives     |
        |                                       |
        | Push: { type: "telemetry", ... }      |
        |◄──────────────────────────────────────|
        |                                       |
```

---

7. Data Flow Analysis

Understanding how data moves through the system is crucial for debugging, optimization, and extending functionality. This section traces the complete path of data through all system layers, from sensor reading to dashboard display.

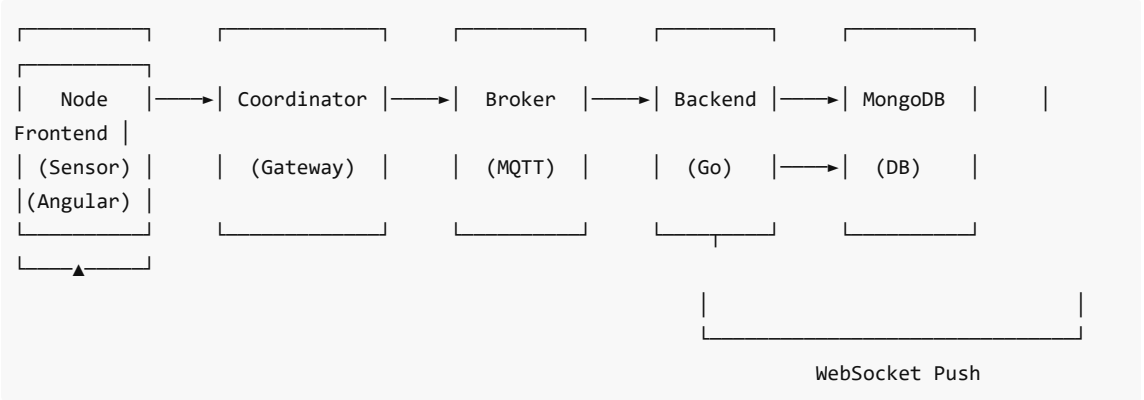
Data flows in our system are fundamentally **asynchronous** and **event-driven**. Rather than polling for updates, components publish events when state changes, and interested parties subscribe to receive those events. This design minimizes wasted bandwidth and CPU cycles while ensuring updates are delivered as quickly as possible.

7.1 Telemetry Uplink

The telemetry uplink path moves sensor data from nodes to the dashboard. This is the most common data flow in the system, occurring every few seconds for each active node.

The key insight here is that data is transformed at each hop:

- **Node** → **Coordinator**: Binary ESP-NOW frame (compact, fast)
- **Coordinator** → **Broker**: JSON over MQTT (structured, debuggable)
- **Broker** → **Backend**: Same JSON (no transformation)
- **Backend** → **MongoDB**: BSON document (indexed, queryable)
- **Backend** → **Frontend**: JSON over WebSocket (real-time push)



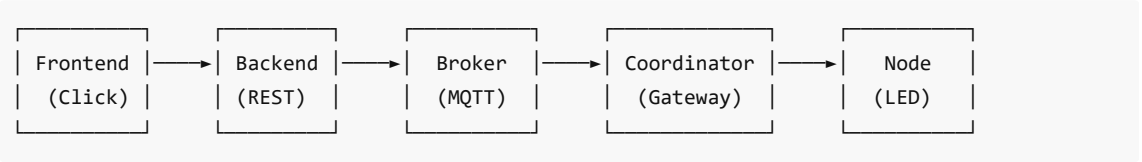
Latency Breakdown:

Hop	Protocol	Typical Latency
Node → Coordinator	ESP-NOW	<10ms
Coordinator → Broker	MQTT/WiFi	20-50ms
Broker → Backend	MQTT	<5ms

Backend → MongoDB	TCP	<10ms
Backend → Frontend	WebSocket	<20ms
Total		**~100ms**

7.2 Command Downlink

The command downlink path moves user commands to the nodes.

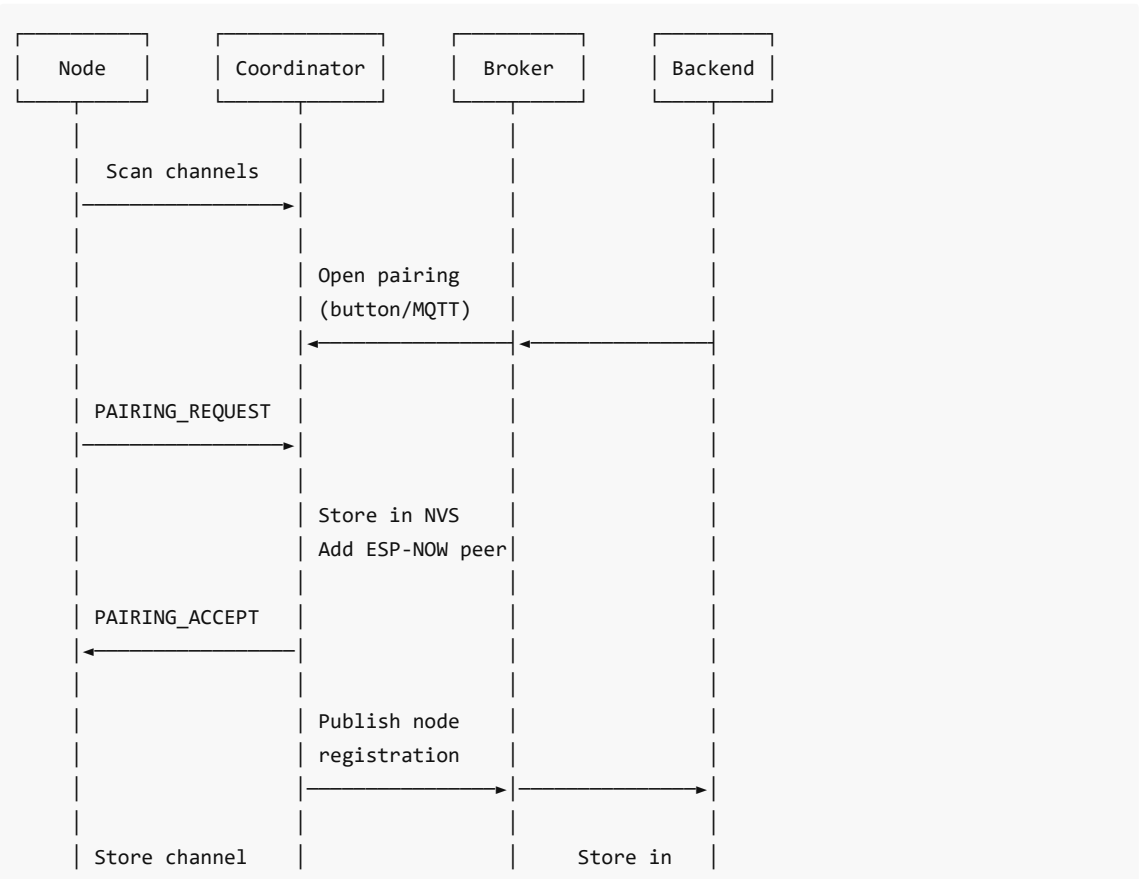


Example: Set LED Color

1. User clicks color picker in dashboard
2. Frontend sends `PUT /api/nodes/node001/led` with `{ "rgbw": {...} }`
3. Backend publishes to `site/site001/node/node001/cmd`
4. Broker routes to subscribed coordinator
5. Coordinator translates to ESP-NOW message
6. Node receives and updates PWM values

7.3 Pairing Flow

The pairing flow adds new nodes to a zone.



```
      | Enter OPERATIONAL|                |     MongoDB    |
      |                  |                |                |
```

---

8. Security Considerations

Security in IoT systems is notoriously challenging. Devices are often physically accessible to attackers, firmware updates are difficult to deploy, and the long lifespans of embedded devices mean vulnerabilities discovered years after deployment must still be addressed.

Our current implementation prioritizes **functionality over security** during the development phase. This is a deliberate choice—securing a system that doesn't work correctly is pointless. However, before any production deployment, the security gaps identified below must be addressed.

The good news is that our architecture supports security enhancements without major redesign. Each protocol we've chosen (ESP-NOW, MQTT, HTTPS) has well-defined security extensions that can be enabled incrementally.

Current Implementation

The following table provides an honest assessment of our current security posture:

Layer	Security Measure	Status
ESP-NOW	AES-128 CCMP encryption	✅ Implemented
MQTT	Username/password auth	✅ Implemented
MQTT	TLS encryption	❌ Not implemented
REST API	No authentication	❌ Not implemented
WebSocket	No authentication	❌ Not implemented
MongoDB	No authentication	❌ Not implemented

Recommended Improvements

1. **TLS for MQTT**
    - Enable `listener 8883` with certificates
    - Use Let's Encrypt for production
2. **API Authentication**
    - Implement JWT tokens
    - Add rate limiting
3. **Database Security**
    - Enable MongoDB authentication
    - Use role-based access control
4. **Network Isolation**
    - Place IoT devices on separate VLAN
    - Use firewall rules

---

9. Scalability Analysis

Current Limits

Component	Limit	Reason
Nodes per Coordinator	20	ESP-NOW encrypted peer limit
Coordinators per Site	Unlimited	MQTT scales horizontally
Sites	Unlimited	MongoDB sharding available
Telemetry Rate	~200 msg/sec	Single MongoDB instance

Scaling Strategies

Horizontal Scaling:

- Add more coordinators for more nodes
- Shard MongoDB by site ID
- Add MQTT broker clustering

Vertical Scaling:

- Increase MongoDB resources
- Use dedicated MQTT broker hardware

---

10. Failure Modes and Recovery

Distributed systems fail. Networks partition, devices lose power, software crashes. The question isn't whether failures will occur, but how the system behaves when they do.

A well-designed system degrades gracefully—partial failures don't cascade into total system outages. Our architecture achieves this through:

1. **Loose coupling**: Components communicate via messages, not direct calls
2. **Local autonomy**: Coordinators can operate without cloud connectivity
3. **Automatic recovery**: Failed components reconnect without manual intervention
4. **Visibility**: Failures are detected and surfaced to users quickly

The following tables enumerate the failure modes we've identified and how the system handles each.

Node Failures

Nodes are the most failure-prone components due to their battery power and physical exposure.

Failure	Detection	Recovery
Node offline	Missing heartbeat (30s)	Dashboard shows offline status
Battery depleted	Voltage <3.0V	Deep sleep, wake on charge
Lost coordinator	ESP-NOW timeout	Re-scan channels, re-pair

Coordinator Failures

Failure	Detection	Recovery
WiFi disconnect	Connection callback	Auto-reconnect with backoff
MQTT disconnect	Connection callback	Auto-reconnect with backoff
NVS corruption	Init failure	Erase and reinitialize
Crash/reboot	Watchdog timer	Automatic restart

Backend Failures

Failure	Detection	Recovery
Backend crash	Health check fails	Docker auto-restart
MongoDB unavailable	Connection error	Retry with exponential backoff
MQTT unavailable	Connection error	Retry with exponential backoff

---

11. Design Trade-offs

Every architectural decision involves trade-offs. There are no perfect solutions—only solutions that optimize for particular constraints at the expense of others. Understanding these trade-offs is essential for:

1. **Evaluating alternatives**: Knowing why we chose X over Y helps assess future options
2. **Communicating limitations**: Being explicit about what we sacrificed avoids unrealistic expectations
3. **Planning evolution**: Understanding constraints helps identify when they might change

This section documents the major trade-offs we made and our reasoning behind each choice.

ESP-NOW vs WiFi for Nodes

Chose ESP-NOW because:

- ✅ Lower power consumption (no WiFi stack)
- ✅ Faster connection (no association)
- ✅ Works without router infrastructure
- ❌ Limited to 20 peers per coordinator
- ❌ Requires coordinator gateway

MongoDB vs Time-Series Database

Chose MongoDB because:

- ✅ Flexible schema for varied payloads
- ✅ Good enough for moderate telemetry rates
- ✅ Simpler operational model
- ❌ Less efficient than InfluxDB for pure time-series

Single Coordinator per Zone vs Mesh

Chose single coordinator because:

- ✅ Simpler architecture
- ✅ Clear data ownership
- ✅ Easier debugging
- ❌ Single point of failure per zone
- ❌ Range limited to coordinator coverage

WebSocket vs Server-Sent Events

Chose WebSocket because:

- ✅ Bidirectional (can send commands too)
- ✅ Lower latency
- ✅ Better browser support
- ❌ More complex than SSE
- ❌ Requires connection management

Conclusion

The IOT Smart Tile System represents a thoughtful approach to the challenges of modern IoT development. Rather than treating the system as a monolith, we've decomposed it into specialized components that communicate through well-defined interfaces. This separation allows each component to be optimized for its specific constraints:

- **Nodes** optimize for power efficiency through deep sleep and lightweight protocols
- **Coordinators** optimize for reliability through dual-network bridging and local caching
- **Backend** optimizes for flexibility through clean abstractions and dependency injection
- **Frontend** optimizes for responsiveness through reactive programming and WebSocket updates

The architecture successfully balances the competing demands of:

- **Real-time responsiveness** (<100ms end-to-end latency from sensor to dashboard)
- **Power efficiency** (battery-powered nodes achieving months of operation)
- **Scalability** (20 nodes per coordinator, unlimited coordinators per site)
- **Maintainability** (modular manager pattern, clean separation of concerns)
- **Developer experience** (modern stack: Go, Angular, TypeScript with strong typing)

The hybrid ESP-NOW + MQTT approach provides the best of both worlds: ultra-low latency local communication that makes lights feel instant, combined with reliable cloud connectivity that enables remote access from anywhere.

Future Directions

While the current architecture is functional and maintainable, several areas warrant future investment:

1. **Security hardening**: TLS for MQTT, JWT authentication for REST API, MongoDB access control
2. **Scaling optimizations**: Database sharding for multi-site deployments, MQTT broker clustering
3. **Advanced automation**: Machine learning for occupancy prediction, adaptive lighting based on circadian rhythms
4. **Platform expansion**: Mobile applications, integration with additional smart home ecosystems

The modular architecture we've built provides a solid foundation for these enhancements without requiring fundamental redesign.

Document Version: 1.0 Last Updated: December 2024