

# 대규모 사물인터넷 단말 관리를 위한 클라우드 기반 OTA 기술 개발



201924479 박준우

202255665 송승우

202255670 장민준

지도교수 김태운

---

## 목 차

1. 서론	1
1.1. 연구 배경	1
1.2. 기존 문제점	2
1.3. 연구 목표	3
2. 연구 배경	5
2.1. 용어 정리	5
2.2. 기능 정의	9
2.3. 개발 환경	10
3. 연구 내용	10
3.1. 서비스 전체 구성도	10
3.1.1. 시스템 설계 배경	11
3.1.2. 클라우드 네이티브 아키텍처 채택	11
3.1.3. 글로벌 콘텐츠 배포를 위한 CDN 활용	11
3.1.4. 마이크로서비스 아키텍처 적용	11
3.1.5. 전체 시스템 구성	12
3.2. 프론트엔드	12
3.2.1. 페이지 라우팅	12
3.2.2. 디렉토리 구조	13
3.2.3. 펌웨어 관리 페이지	15
3.2.4. 기기 관리 페이지	21
3.2.5. 콘텐츠 관리 페이지	23

---

3.2.6. 상태 관리	26
3.2.7. 사용자 UX 개선	29
3.3. 클라우드 인프라	30
3.3.1. 클라우드 인프라 개요	30
3.3.2. 고가용성 및 확장성 구현	31
3.3.3. 강화된 보안 아키텍처 구현	34
3.3.4. 기타 리소스 정의	39
3.3.5. 개발 편의성을 위한 데브옵스(DevOps)	40
3.3.6. 테스트를 위한 기기 시뮬레이터 구현	42
3.4. 백엔드	45
3.4.1. 아키텍처	45
3.4.2. 관리자 API 서버	46
3.4.3. MQTT 핸들러	51
3.4.4. 데이터베이스 설계	54
3.5. 임베디드 시스템	56
3.5.1. 시스템 개요 및 하드웨어 구조	56
3.5.2. 임베디드 펌웨어 아키텍처	57
3.5.3. 하드웨어 드라이버 및 UI	62
3.5.4. 네트워크 및 보안 통신	64
3.5.5. OTA 업데이트 및 롤백	66
4. 연구 결과 분석 및 평가	68
4.1. 펌웨어 배포 성능 실험	68
4.2. CloudFront(CDN) 기반 콘텐츠 전송 성능 실험	69
5. 결론 및 향후 연구 방향	70



---

# 1. 서론

## 1.1. 연구 배경

지속적인 기술 발전은 산업 전반의 디지털 전환(Digital Transformation)을 가속화하였다. 특히 코로나 19 팬데믹을 계기로 대면 업무와 현장 중심의 관리 방식이 한계에 직면하면서, 비대면 서비스 및 업무 자동화, 원격 관리에 대한 수요가 폭발적으로 증가하였다. 이러한 흐름 속에서 제조업, 물류, 상업 서비스, 공공 인프라 등 다양한 분야에 IoT(Internet of Things) 기술이 급격히 확산되었으며, 그 결과 기기 간의 연결과 중앙 집중식 관리 체계의 중요성이 한 층 더 부각되었다.

그러나 모든 산업 분야가 이러한 변화를 쉽게 수용할 수 있는 것은 아니다. 여전히 많은 산업 현장에서는 MCU(Microcontroller Unit) 기반 임베디드 장치가 핵심적인 역할을 차지하고 있다. 이들 기기는 부족한 접근성과 메모리 및 저장 공간의 제약, 연산 능력의 한계로 인해 상대적으로 자원이 충분한 리눅스 기반 임베디드 장치에 비해 확장성과 유지보수성에 불리하다. 특히, 이러한 장치를 대규모로 운영해야 하는 산업 환경에서는 체계적인 소프트웨어 업데이트 인프라 구축이 어렵고, 이로 인해 관리상의 부담이 크게 가중된다.

이러한 상황에서 OTA(Over-the-Air) 기술은 필수적인 역할을 한다. OTA 는 현장에 직접 접근하지 않고도 무선 네트워크를 통해 펌웨어와 애플리케이션을 원격으로 업데이트할 수 있는 기술로, 수백, 수천 대 이상의 분산된 장치를 관리해야 하는 환경에서 유지보수 비용을 절감하고 관리 효율성을 극대화할 수 있다. 디지털 전환이 본격화되며 원격 유지보수의 필요성이 더욱 커지면서, OTA 기술은 이제 임베디드 장치를 운영하는 산업 현장에서 핵심 인프라로 자리잡았다.

본 연구는 이러한 흐름 속에서, 실제 산업 환경과 유사한 조건으로 운용되는 대규모 MCU 기반 임베디드 장치에 적용 가능한 OTA 기술을 개발하는 것을 목표로 한다. 단순한 기능 구현 넘어 장치, 펌웨어, 네트워크 인프라, 관리 방식을 통합적으로 설계하여 산업 현장에 도입 가능한 수준의 실증 가능한 시스템을 제시하고자 한다.

---

특히, 클라우드 기반 관리 플랫폼을 도입하여 단일 장치가 아닌 대규모 장치 집합을 대상으로 한 통합 관리 가능성을 확보하는 데 중점을 두었다. 클라우드 서버를 통해 각 장치의 상태를 모니터링하고, 소프트웨어와 콘텐츠를 일괄 배포하며, 원격에서 효율적인 제어와 데이터 수집을 수행함으로써 운영자는 수많은 장치를 안정적이고 경제적으로 운용할 수 있다. 이는 단순한 개별 기기 관리가 아니라, 전 세계에 분산된 IoT 기기를 대상으로 한 중앙 집중식 관리 체계를 실질적으로 구현하기 위한 시도라 할 수 있다.

아울러 최근 IoT 기술에 대한 보안 위협이 급증하면서, 통신 보안 및 시스템의 신뢰성 확보는 산업 현장에 적용하기 위한 필수 요건으로 자리잡고 있다. 본 연구에서는 장치와 클라우드 서버 간의 데이터 전송 과정에 mTLS(Mutual TLS) 기반 암호화 및 인증을 적용하고, 배포되는 소프트웨어 및 콘텐츠의 무결성을 검증하며, 필요 시 롤백이 가능한 안정성을 제공함으로써, 단순 OTA 를 넘어 보안성과 신뢰성을 강화한 클라우드 기반 OTA 관리 기술을 제안한다.

## 1.2. 기존 문제점

### 1.2.1. 기존 MCU 기반 임베디드 시스템의 한계

현재 산업 현장에는 여전히 IoT 기능이 탑재되지 않은 전통적인 MCU 기반 임베디드 시스템이 다수 운영되고 있다. 이들 장치는 특정 기능만을 수행하도록 제작되었기 때문에, 네트워크 연결이나 원격 제어와 같은 현대적인 관리 기능을 포함하지 않는다. 특히 건물 내 기계실, 옥외 전력 설비 등 작업자가 접근하기 어려운 위치에 설치된 경우가 많아, 물리적 접근부터 큰 부담이 된다. 또한 MCU 기반 임베디드 장치들은 한정된 연산 성능과 메모리 및 저장 공간으로 인해 관리 체계를 개선하거나 기능을 확장하는 과정이 구조적 제약을 가진다.

이러한 장치들을 유지 및 관리하기 위해서는 주로 현장 방문을 통한 소프트웨어 교체나 메모리 카드 교체 방식이 사용된다. 그러나 수십 대, 수백 대에 달하는 장치가 넓은 지역에 분산 설치된 경우, 단순한 소프트웨어 교체 작업에도 막대한 시간과 인력이 소요된다. 때문에 물리적 접근에 의존하는 기존의 유지보수 방식은 대규모 운영 환경에서 지속 불가능하며, 결국 업데이트 주기를 늘리거나 보안 패치를 제때 적용하지 못하는 문제를 야기한다.

---

### 1.2.2. 기존 시스템에 IoT 기능 후속 도입의 어려움

실제 산업 현장에서는 기존의 장비 전체를 교체하기보다 기존 시스템에 IoT 기능을 후속으로 추가하려는 시도가 점차 늘어나고 있다. 보일러, 자판기, 음료 디스펜서 등 다양한 장치가 그 대상이 된다. 특히 OTA를 기반으로 한 펌웨어 및 콘텐츠 업데이트 수요는 빠르게 증가하고 있으나, 저자원 환경에서 안정적으로 OTA를 지원하기 위한 기술적 기반은 아직 부족한 상황이다. 이로 인해 장치별로 구현 방식이 제각각 달라지고, 통합된 운영 표준이 부재한 상황이 지속되면서 산업 전반의 관리 효율성을 저해하고 있다.

### 1.2.3. 규모 확장성과 관리 체계의 부족

스마트홈이나 웨어러블 기기와 같은 가정용 IoT 기기는 소규모 환경에서 운용되므로 스마트폰 애플리케이션을 통한 단순 관리만으로도 충분하다. 그러나 산업 현장이나 기업 단위에서 운용되는 IoT 시스템은 성격이 다르다. 이 경우 수백, 수천 대 이상의 장치를 동일한 상태로 유지하고, 관리 범위가 국내 전역 또는 전 세계로 확대되더라도 자동으로 확장 가능한 관리 체계가 요구된다. 하지만 기존의 시스템은 개별 기기 단위의 관리 방식에 머물러 있어 중앙 집중식 관리 및 상태 모니터링 체계가 부재하다. 이로 인해 운영자는 장치의 상태를 실시간으로 파악하거나 일괄적인 업데이트 및 정책 변경을 수행하기 어렵고, 이는 대규모 IoT 시스템의 확산을 가로막는 주요 걸림돌로 작용한다.

### 1.2.4. 보안 및 신뢰성 확보의 어려움

기존 MCU 기반 장치들은 설계 당시부터 보안 기능이 충분히 고려되지 않은 경우가 많다. 네트워크 기능을 후속으로 연결했을 때에도 데이터 암호화, 인증 절차, 무결성 검증과 같은 필수 보안 요소를 제대로 구현하기 어렵다. 그 결과, 장치가 악성 업데이트를 수용하거나, 통신이 도청되는 등 다양한 위협에 쉽게 노출될 수 있다. 이러한 보안과 신뢰성 문제는 실제 산업 현장에서 IoT 및 OTA 기술의 도입을 주저하게 만드는 핵심 요인이다.

## 1.3. 연구 목표

본 연구는 MCU 기반 임베디드 장치가 대규모로 운영되는 산업 환경에서, 클라우드 기반의 OTA 관리 기술을 구현하는 것을 목표로 한다. 이를 통해 현장 접근에 의존하던 기존의

---

유지보수 방식을 대체하고, 수많은 장치를 안정적이면서도 효율적으로 관리할 수 있는 기반을 마련하고자 한다. 본 연구에서는 크게 클라우드 기반 대규모 관리 체계 구축, 분산 콘텐츠 관리 및 배포 최적화, 장치 상태 원격 모니터링 및 데이터 수집, 저자원 환경에서의 OTA 기술 구현 및 최적화, 보안 및 신뢰성 확보의 다섯 가지 세부 목표를 설정한다.

첫째, 대규모 기기 동시 관리 및 확장성 확보를 위한 클라우드 기반 관리 체계를 구축한다. 수백, 수천 대 이상의 장치를 중앙에서 일괄적으로 관리할 수 있는 플랫폼을 설계하여, 운영자는 개별 장치 단위가 아닌 집합 단위로 대상으로 원격 제어와 소프트웨어 업데이트를 수행할 수 있다. 또한 클라우드 서버를 통해 각 장치의 상태를 모니터링하고, 정책 변경 및 콘텐츠 배포를 일관되게 수행할 수 있는 관리 체계를 마련한다. 이 과정에서 서버의 부하 분산과 스케일 아웃을 고려하여, 장치 수가 증가하더라도 안정적으로 확장 가능한 백엔드 인프라 구조를 구현한다.

둘째, 분산 콘텐츠 관리 및 배포를 최적화한다. 산업 현장에서 운영되는 IoT 기기는 단순 소프트웨어 업데이트뿐 아니라 광고와 같은 콘텐츠, 인터페이스 리소스 등 다양한 대용량 데이터를 주기적으로 수신해야 한다. 이를 위해 클라우드 스토리지와 CDN(Content Delivery Network)을 연계하여, 전 세계적으로 분산된 장치에 동일한 콘텐츠를 효율적으로 배포할 수 있는 체계를 구축한다. 이를 통해 네트워크 트래픽을 최적화하고, 특정 지역이나 네트워크 환경에 따른 편차를 최소화하며, 대규모 장치 운영의 안정성을 확보한다.

셋째, 장치 상태를 원격으로 모니터링 및 데이터 수집 기능을 구현한다. 개별 장치의 에러 로그, 네트워크 연결 상태, OTA 성공 여부 등 다양한 데이터를 클라우드로 수집하여, 전체 시스템의 상태를 실시간으로 파악할 수 있도록 한다. 이러한 데이터는 단순 현황 확인을 넘어, 운영 효율성 제고와 장애 대응을 위한 기초 자료로 활용될 수 있으며, 클라우드 기반 대시보드를 통해 운영자가 전체 장치의 상태를 직관적으로 파악할 수 있다. 나아가 장기적으로 수집된 데이터들을 통해 운영 지표와 정책 수립에 활용될 수 있도록 한다.

넷째, 저자원 임베디드 장치에서 OTA 기술을 구현하고 최적화한다. MCU 기반 환경에서도 안정적으로 동작하는 OTA 업데이트 메커니즘을 개발한다. OTA 과정에서 발생할 수 있는



---

오류나 네트워크 불안정 상황에 대응하기 위해 롤백 기능을 제공하고, 업데이트 파일의 무결성을 검증하여 안정적인 업데이트 프로세스를 보장한다. 이를 통해 자원이 제한된 환경에서도 대규모 OTA가 원활히 수행될 수 있도록 한다.

다섯째, 보안 및 신뢰성을 확보한다. IoT 환경은 수많은 보안 위협에 노출되어 있으며, 특히 OTA는 악성 코드 유입이나 데이터 변조 위험에 취약하다. 이를 방지하기 위해 본 연구에서는 mTLS(Mutual TLS)를 활용하여 클라우드 서버와 기기 간의 상호 인증을 수행하고, 데이터 전송 과정을 암호화한다. 또한 OTA 파일의 해시 검증을 통한 무결성 검증과 업데이트 과정 중 오류가 발생했을 때 신속히 복구할 수 있는 롤백 메커니즘을 적용함으로써, 시스템의 보안성과 안정성을 추가로 확보한다.

종합적으로 본 연구는 클라우드 인프라와 임베디드 OTA 기술을 결합하여, 대규모 분산 IoT 장치를 효율적이고 안전하게 관리할 수 있는 통합 관리 체계를 제시한다. 이는 단순히 개별 기기의 기능 확장을 넘어 산업 현장에서 실제로 적용 가능한 수준의 확장성과 안정성, 보안성을 갖춘 플랫폼을 구축한다는 점에서, 실용적 가치와 학술적 기여를 동시에 추구하는 시도라 할 수 있다.

## 2. 연구 배경

### 2.1. 용어 정리

#### ● 클라우드 컴퓨팅

클라우드 컴퓨팅은 인터넷을 통해 서버, 스토리지, 데이터베이스, 네트워킹, 소프트웨어 등의 컴퓨팅 서비스를 제공하는 기술이다. 물리적인 서버나 데이터 센터를 직접 소유하고 관리할 필요 없이, 필요할 때마다 필요한 만큼의 IT 리소스를 빌려 쓰는 방식이다. 이를 통해 기업과 개인은 비용을 절감하고 유연성을 높이며, 더 빠른 혁신을 이룰 수 있다. 대표적인 서비스로는 AWS, Google Cloud, Microsoft Azure가 있다.

#### ● ECS

---

ECS 는 AWS 에서 제공하는 완전 관리형 컨테이너 오케스트레이션 서비스이다. Docker 컨테이너를 손쉽게 배포, 관리, 확장할 수 있도록 지원한다. 사용자는 기본 인프라를 직접 관리할 필요 없이 컨테이너화된 애플리케이션을 안정적으로 실행하는 데 집중할 수 있다. ECS 는 VPC, ALB 등 다른 AWS 서비스와 긴밀하게 통합되어 강력한 클라우드 네이티브 환경을 구축하는 데 이상적이다.

## ● VPC

VPC 는 사용자의 AWS 계정 전용으로 논리적으로 분리된 가상 네트워크 공간이다. 이 공간 안에서 IP 주소 범위, 서브넷 생성, 라우팅 테이블 및 네트워크 게이트웨이 구성 등 가상 네트워크 환경을 완벽하게 제어할 수 있다. VPC 를 통해 AWS 클라우드 내의 리소스를 안전하게 격리하고, 기업의 자체 데이터 센터와 유사한 네트워크 환경을 구축할 수 있다.

## ● CloudFront(CDN)

CloudFront 는 AWS 에서 제공하는 콘텐츠 전송 네트워크(CDN) 서비스이다. 웹 콘텐츠(이미지, 동영상, 애플리케이션 등)를 전 세계에 분산된 엣지 로케이션(캐시 서버)에 복사하여 사용자에게 더 가까운 곳에서 콘텐츠를 제공한다. 이를 통해 데이터 전송 지연 시간을 최소화하고 로딩 속도를 크게 향상시켜 사용자 경험을 개선한다.

## ● EMQX

EMQX 는 대규모 사물인터넷(IoT) 환경을 위한 고성능 오픈소스 MQTT 메시지 브로커이다. 수백만 개의 동시 MQTT 연결을 안정적으로 처리할 수 있도록 설계되었다. 실시간 데이터 수집, 라우팅 및 처리를 위한 강력한 기능을 제공하며, 높은 확장성과 가용성을 보장하여 IoT 플랫폼의 핵심적인 통신 허브 역할을 수행한다.

## ● mTLS(Mutual TLS)

mTLS 는 클라이언트와 서버가 통신을 시작하기 전에 서로의 신원을 상호 인증하는 보안 프로토콜이다. 일반적인 TLS(전송 계층 보안)에서는 클라이언트가 서버의 신원만 확인하지만, mTLS 에서는 서버도 클라이언트의 인증서를 확인한다. 이를 통해 허가된

---

장치나 사용자만이 민감한 시스템에 접근하도록 보장하며, 제로 트러스트(Zero Trust) 보안 모델을 구현하는 데 핵심적인 역할을 한다.

### ● QuestDB

QuestDB 는 시계열 데이터(time-series data)를 위해 특별히 설계된 고성능 오픈소스 데이터베이스이다. 금융 시장 데이터, IoT 센서 데이터, 애플리케이션 모니터링 로그와 같이 시간에 따라 연속적으로 발생하는 데이터를 매우 빠른 속도로 수집, 저장 및 쿼리하는 데 최적화되어 있다. SQL 을 기본 쿼리 언어로 지원하여 사용이 편리하며, 빠른 데이터 수집(ingestion)과 실시간 분석에 강점을 가진다.

### ● React

React 는 페이스북에서 개발한 사용자 인터페이스(UI) 구축을 위한 JavaScript 라이브러리이다. 컴포넌트 기반 아키텍처를 채택하여, 독립적이고 재사용 가능한 UI 조각들을 조립해 복잡한 UI 를 효율적으로 개발할 수 있다. 가상 DOM(Virtual DOM)을 사용하여 UI 변경 사항을 효율적으로 처리하고 렌더링 성능을 최적화함으로써, 동적이고 빠른 웹 애플리케이션 제작에 널리 사용된다.

### ● Spring Boot

Spring Boot 는 자바 기반의 오픈소스 애플리케이션 프레임워크로, 복잡한 설정 과정을 최소화하고 빠르게 독립 실행형 애플리케이션을 개발할 수 있도록 지원한다. 내장 웹 서버와 자동 설정 기능을 제공하여 별도의 외부 서버나 복잡한 환경 구성이 필요하지 않으며, 다양한 Starter 패키지를 통해 데이터베이스, 보안, 메시징 등 필요한 기능을 손쉽게 확장할 수 있다.

### ● Go

Go 는 구글에서 개발한 오픈소스 프로그래밍 언어로, 단순하고 직관적인 문법과 정적 타입 기반의 높은 성능을 제공한다. 고루틴(Goroutine)과 채널(Channel)을 통한 동시성 지원으로 대규모 병렬 처리를 효율적으로 수행할 수 있으며, 네트워크 서버, 분산 시스템, 클라우드

---

네이티브 애플리케이션 개발에 널리 활용된다.

### ● MCU(Microcontroller Unit)

MCU 는 중앙처리장치(CPU), 메모리, 입출력 주변 장치를 단일 칩에 집적한 소형 컴퓨터 시스템이다. 제한된 전력과 비용으로 기본적인 연산과 제어를 수행할 수 있어 다양한 임베디드 시스템, 특히 가전제품과 산업용 제어 장치 등에 활발히 사용된다. 본 프로젝트에서는 ESP32-S3 MCU 를 기반으로 하여 OTA 업데이트, 보안 통신, 사용자 인터페이스 등을 통합적으로 수행하였다. ESP32-S3 는 듀얼 코어와 Wi-Fi 모듈을 내장하고 있어, 대규모 IoT 기기의 관리 및 클라우드 연동을 실증하기 위한 저자원 MCU 환경으로 적합하다.

### ● FreeRTOS(Free Real-Time Operating System)

FreeRTOS 는 임베디드 시스템에서 널리 사용되는 경량의 실시간 운영체제(RTOS)다. 태스크(Task) 단위의 멀티태스킹, 태스크 간 동기화, 인터럽트 처리 등을 지원하여 제한된 MCU 환경에서도 복잡한 병렬 처리를 가능하게 한다. 본 프로젝트에서는 FreeRTOS 를 활용하여 UI 태스크, 네트워크 태스크, OTA 다운로드 태스크, 시스템 상태 보고 태스크 등을 병렬로 운영하였다. 이를 통해 OTA 업데이트, 사용자 입력 처리, 네트워크 통신이 동시에 안정적으로 수행될 수 있도록 한다.

### ● LVGL(Light and Versatile Graphics Library)

LVGL 은 임베디드 장치에서 GUI(Graphical User Interface)를 구현하기 위한 오픈소스 그래픽 라이브러리다. 버튼, 슬라이더, 차트 등 다양한 위젯과 이벤트 처리 기능을 제공하며, 메모리 최적화와 부분 업데이트 방식을 통해 저자원 환경에서도 부드러운 화면 구동을 지원한다. 본 프로젝트에서는 LVGL 을 활용하여 ESP32-S3 기반 HMI 보드의 TFT-LCD 화면에 GUI 를 구현하였다. OTA 상태, 네트워크 상태, 광고 콘텐츠 등 다양한 정보를 사용자 친화적인 인터페이스로 시각화하였으며, 터치 드라이버와 연계하여 간편한 입력 방식을 제공한다.

## 2.2. 기능 정의

본 기능 정의서는 펌웨어 관리, 콘텐츠 관리, 기기·그룹·리전 관리 등 본 프로젝트의 핵심 기능들을 구체적으로 정리한 문서이다. 각 기능은 요구사항명, 기능명, 상세 설명으로 구분되어 있으며, 이를 통해 개발 범위와 구현 기준을 명확히 한다.

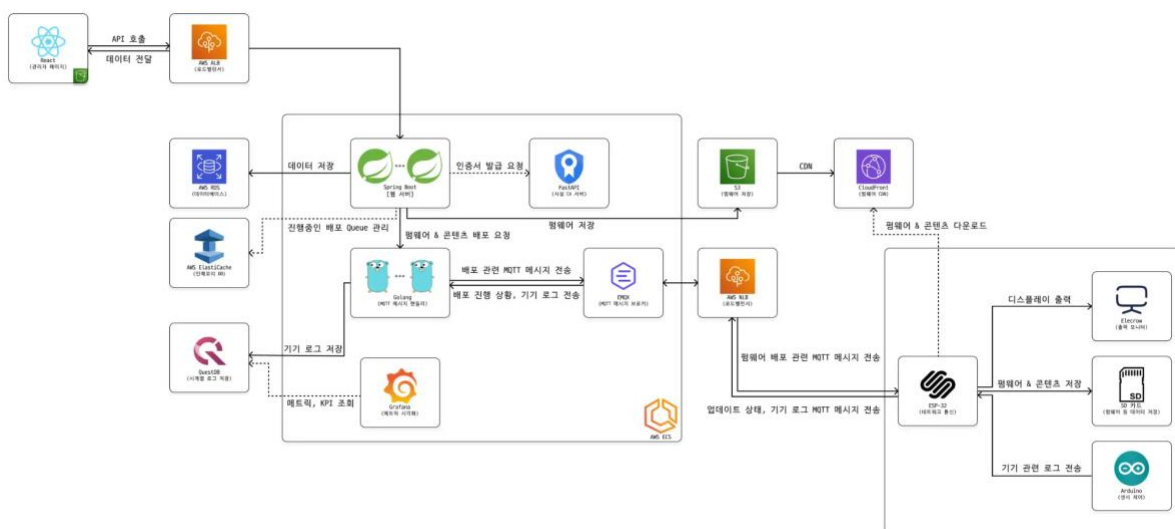
요구사항명	기능명	상세 설명
펌웨어/콘텐츠 관리	파일 업로드	관리자가 신규 펌웨어/콘텐츠를 업로드하기 위해 S3 Presigned URL 을 발급받는 기능
	펌웨어/콘텐츠 메타데이터 등록	업로드된 펌웨어/콘텐츠의 버전, 파일 크기, 해시 값 등 메타데이터를 등록하여 무결성 보장
	펌웨어/콘텐츠 목록 조회	현재 업로드된 펌웨어/콘텐츠들의 목록을 조회
	펌웨어/콘텐츠 상세 조회	특정 펌웨어/콘텐츠의 상세 정보(버전, 해시, 릴리즈 노트 등)을 확인
	펌웨어/콘텐츠 다운로드 URL 발급	펌웨어/콘텐츠를 다운로드할 수 있는 Signed URL 제공
	펌웨어/콘텐츠 배포 요청	선택한 기기/그룹/리전에 펌웨어/콘텐츠 배포 요청
	펌웨어/콘텐츠 배포 내역 조회	전체 펌웨어/콘텐츠 배포 이력을 리스트로 조회
	펌웨어/콘텐츠 배포 상세 조회	특정 배포 건에 대한 세부 현황을 조회
기기·그룹·리전 관리	각 리전에 등록된 기기 수 조회	리전 코드·이름, 각 리전에 등록된 디바이스 수를 제공
	각 그룹에 등록된 기기 수 조회	그룹 코드·이름, 각 그룹에 등록된 디바이스 수를 제공
	기기의 기본 현황 조회	기기명·속한 그룹/리전·활성화 여부를 제공
	기기 상세 조회	특정 기기의 상세 정보를 조회한다. 기본 속성(이름, 생성/수정 시각, 마지막 활성 시각)뿐만 아니라, 소속 그룹·리전, 현재 설치된 펌웨어 버전, 배포된 콘텐츠 이력까지 포함한다
	기기 등록 코드 발급	신규 기기 등록을 위해 인증 코드 발급

## 2.3. 개발 환경

분야	사용 기술 및 도구
프론트엔드	- React 19.0.0 - Vite 6.2.3
클라우드/인프라	- Terraform 1.11.2 - Hashicorp/aws 6.6.0
백엔드	- Java 17 - Go 1.24 - Spring boot 3.5.3
데이터베이스	- MySQL 8.0.41 - QuestDB 9.0.3
IoT 기기	- ESP-IDF 4.4.7 - LVGL 8.3.3

## 3. 연구 내용

### 3.1. 서비스 전체 구성도



---

### 3.1.1. 시스템 설계 배경

본 연구는 전 세계에 분산된 수많은 IoT 기기를 대상으로 펌웨어 및 콘텐츠를 원격으로 업데이트(OTA)하는 시스템을 다루고 있다. 이러한 시스템은 높은 가용성이 요구되며 신속한 파일 전송이 필수적이다. 특히 대용량 펌웨어 파일을 전 세계 기기들에게 동시에 배포해야 하므로, 확장성과 안정성을 동시에 만족하는 아키텍처 설계가 본 프로젝트의 핵심 목표라고 할 수 있다.

### 3.1.2. 클라우드 네이티브 아키텍처 채택

이러한 요구사항을 충족하기 위해 AWS 기반의 클라우드 네이티브 서비스로 전체 시스템을 구축하였다. 클라우드 네이티브 접근 방식을 선택한 이유는 전통적인 온프레미스 환경으로는 글로벌 규모의 IoT 기기들에 대한 동시 접속과 대용량 파일 전송을 효율적으로 처리하기 어렵기 때문이다. AWS의 관리형 서비스들을 활용함으로써 인프라 관리 부담을 줄이고 비즈니스 로직 구현에 집중할 수 있도록 하였다.

### 3.1.3. 글로벌 콘텐츠 배포를 위한 CDN 활용

전 세계 IoT 기기들에게 대용량 펌웨어 파일을 빠르게 전송하기 위해 Amazon CloudFront CDN을 핵심 구성요소로 도입하였다. CDN을 통해 전 세계 엣지 로케이션에 콘텐츠를 분산 배치함으로써 지역별 네트워크 지연 시간을 최소화하고, 원본 서버의 부하를 효과적으로 분산시킬 수 있다. 이는 특히 IoT 기기의 펌웨어 업데이트 시 발생하는 급격한 트래픽 증가 상황에서 안정적인 서비스 제공을 가능하게 한다.

### 3.1.4. 마이크로서비스 아키텍처 적용

기존 Monolithic 아키텍처에서 벗어나 마이크로서비스 아키텍처(MSA)를 적용하였다. 하나의 서비스가 많은 역할을 담당하는 구조에서는 시스템의 복잡도가 증가하고, 특정 기능의 장애가 전체 시스템에 영향을 미칠 위험이 크다. 이러한 문제를 해결하기 위해 Amazon ECS 기반의 컨테이너화된 마이크로서비스 아키텍처를 채택하여 시스템의 확장성과 유연성을 확보하였다. MSA를 통해 서비스들의 책임을 명확히 분산시키고, 각 서비스의 특성에 맞는 최적의 기술 스택과 프레임워크를 선택할 수 있도록 하였다. 또한 컨테이너 기반 구성을 통해 트래픽 변화에 따른 자동 확장 및 축소가 가능하며, 각

---

서비스별로 독립적인 배포와 운영이 가능하다. 이는 IoT 기기의 업데이트 요청이 집중되는 시간대에 유연하게 대응할 수 있는 탄력적인 인프라를 제공하며, 개발 팀의 생산성 향상과 시스템의 장애 격리에도 효과적이다.

### 3.1.5. 전체 시스템 구성

전체 아키텍처는 프론트엔드 계층, 마이크로서비스 계층, 데이터 저장소 계층, 메시징 및 통신 계층, 그리고 모니터링 계층으로 구성된다. React 기반의 웹 애플리케이션을 통해 관리 인터페이스를 제공하며, AWS의 다양한 관리형 서비스들을 활용하여 안정적인 백엔드 인프라를 구축하였다. 각 계층은 명확한 역할 분담을 통해 시스템의 복잡성을 관리하고, 개별 구성요소의 장애가 전체 시스템에 미치는 영향을 최소화하도록 설계되었다.

이러한 아키텍처를 통해 전 세계에 분산된 IoT 기기들에게 안정적이고 효율적인 OTA 업데이트 서비스를 제공할 수 있는 확장 가능한 플랫폼을 구축할 수 있었다. 이하에서는 각 구성요소의 상세한 구현 방법을 설명한다.

## 3.2. 프론트엔드

### 3.2.1. 페이지 라우팅

react-router를 사용하여 SPA(Single Page Application)의 라우팅을 구현했다. 루트 경로에 기본 Layout 컴포넌트를 배치하고, 그 하위에 각 페이지를 중첩(nested) 라우트로 구성하여 일관된 UI 레이아웃을 유지했다.

```
export const Router = createBrowserRouter([
  {
    path: "/",
    element: <Layout />, // 기본 레이아웃 컴포넌트, 모든 페이지가 같은 레이아웃 사용
    children: [
      { path: "firmware", element: <Navigate to="/firmware/list" replace /> },
      ...
    ],
  },
]);
```



Router 의 하위 경로는 다음과 같이 지정하였다.

페이지명	라우트 경로	기능 및 설명
펌웨어 목록	/firmware/list	등록된 펌웨어 목록 조회 및 새 펌웨어 등록 페이지
펌웨어 세부 정보	/firmware/:id	특정 펌웨어의 상세 정보 조회 및 배포 페이지
펌웨어 배포 관리	/firmware/deployment	펌웨어 배포 내역 및 진행중인 배포 조회 페이지
펌웨어 배포 세부 정보	/firmware/deployment/:id	특정 펌웨어 배포 진행 상황 및 결과 상세 조회 페이지
기기 관리	/device	등록된 기기 목록 조회 및 새 기기 등록 페이지
기기 세부 정보	/device/:deviceId	특정 기기 상세 정보 조회 페이지
콘텐츠 목록	/ads/list	등록된 콘텐츠 목록 조회 및 새 콘텐츠 등록, 배포 페이지
콘텐츠 세부 정보	/ads/:id	특정 콘텐츠 상세 조회 페이지
콘텐츠 배포 관리	/ads/deployment	콘텐츠 배포 내역 및 진행중인 배포 조회 페이지
콘텐츠 배포 세부 정보	/ads/deployment/:id	콘텐츠 배포 진행 상황 및 결과 상세 조회 페이지

### 3.2.2. 디렉토리 구조

프론트엔드의 확장성과 유지보수성을 높이기 위해 FSD(Feature-Sliced Design) 아키텍처를 채택하였다. 기능 단위로 코드를 분리하여 비즈니스 로직과 UI 컴포넌트 간의 결합도를 낮추고, 코드의 재사용성을 극대화하였다. 아래는 전체적인 프론트엔드의 디렉토리

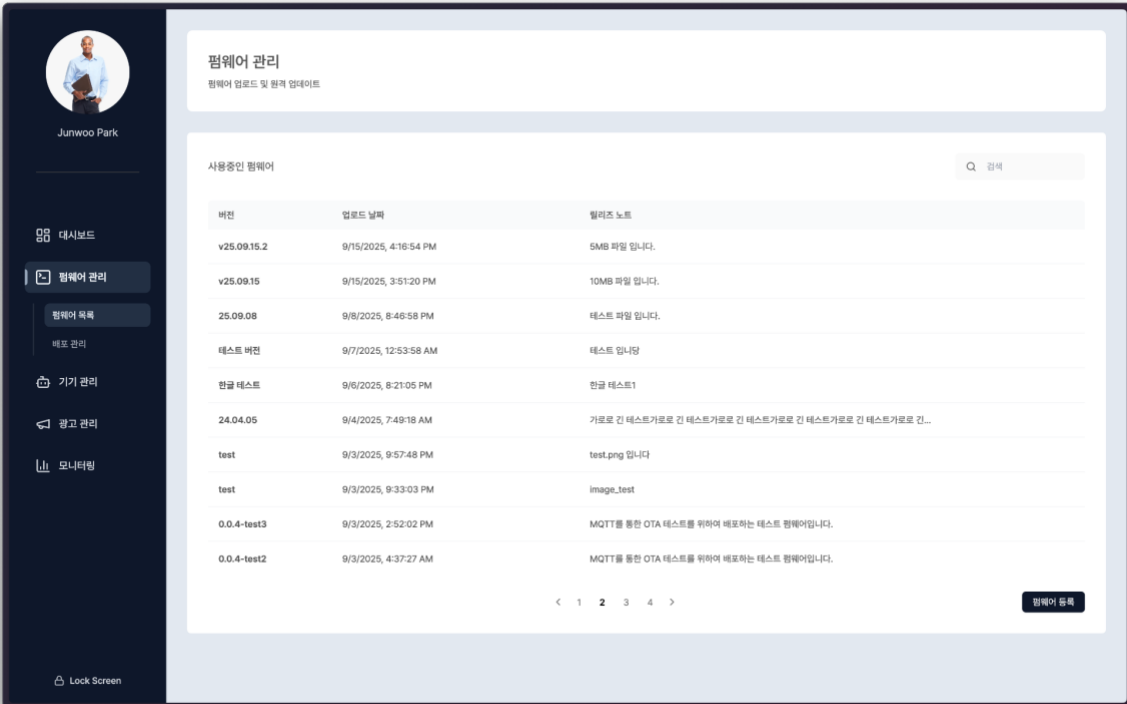
구조이다.

```
/
├── public/          # 정적 에셋 (이미지, mock-service-worker 등)
├── src/
│   ├── app/        # 어플리케이션 진입점, 전역 설정
│   │   ├── App.tsx  # 최상위 앱 컴포넌트
│   │   └── Router.tsx # 페이지 경로 및 라우팅 구조 정의
│   │
│   ├── pages/       # 라우팅 경로에 해당하는 페이지 컴포넌트
│   │   ├── DashboardPage.tsx
│   │   ├── FirmwareListPage.tsx
│   │   └── ... (각 페이지)
│   │
│   ├── widgets/     # 여러 페이지에서 재사용되는 복합 UI
│   │   ├── layout/  # 전체 페이지 레이아웃
│   │   └── sidebar/  # 사이드바 및 네비게이션 메뉴
│   │
│   ├── features/    # 특정 기능을 수행하는 컴포넌트와 로직 집합
│   │   ├── ad_deploy/ # 콘텐츠 배포 기능
│   │   ├── firmware_register/ # 펌웨어 등록 기능
│   │   └── ... (기능별 디렉토리)
│   │
│   ├── entities/    # 어플리케이션의 핵심 도메인 데이터
│   │   ├── device/  # 기기 (API, 타입, UI)
│   │   ├── firmware/ # 펌웨어 (API, 타입, UI)
│   │   ├── advertisement/ # 콘텐츠 (API, 타입, UI)
│   │   └── ... (도메인별 디렉토리)
│   │
│   └── shared/       # 여러 곳에서 공통으로 사용되는 모듈
│       ├── api/      # Axios 클라이언트, 공통 API 함수
│       ├── mocks/     # MSW 핸들러, mock 데이터
│       └── ui/        # Button, Modal 등 공용 UI 컴포넌트
```

shared - entities - features - widgets - pages - app 순으로 계층을 설정하여 상위 레이어가 하위 레이어, 혹은 동일 레이어를 임포트할 수 있도록 하고, 그 반대 방향은 허용하지 않는 단방향 의존성 규칙을 적용하였다. 이를 통해 코드의 결합도를 낮추어 유지보수를 쉽게 만들고 순환 참조를 방지할 수 있었다.

### 3.2.3. 펌웨어 관리 페이지

#### 3.2.3.1. 펌웨어 목록



The screenshot shows a web application interface for managing firmware. On the left is a dark sidebar with a user profile (Junwoo Park) and navigation links: 대시보드, 펌웨어 관리 (selected), 펌웨어 목록, 배포 관리, 기기 관리, 광고 관리, and 모니터링. The main content area is titled '펌웨어 관리' and '펌웨어 업로드 및 원격 업데이트'. It features a table of '사용중인 펌웨어' (Firmware in Use) with columns for version, upload date, and release notes. A search bar and a '펌웨어 등록' button are also visible.

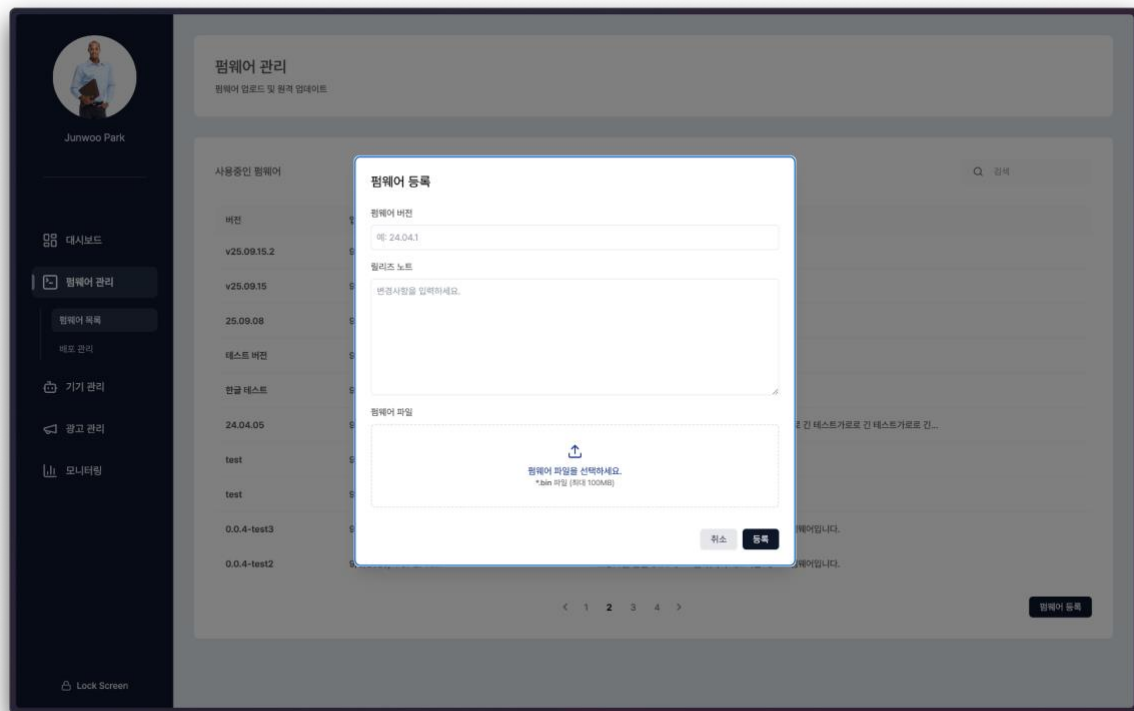
버전	업로드 날짜	릴리즈 노트
v25.09.15.2	9/15/2025, 4:16:54 PM	5MB 파일입니다.
v25.09.15	9/15/2025, 3:51:20 PM	10MB 파일입니다.
25.09.08	9/8/2025, 8:46:58 PM	테스트 파일입니다.
테스트 버전	9/7/2025, 12:53:58 AM	테스트 임나당
한국 테스트	9/6/2025, 8:21:05 PM	한국 테스트1
24.04.05	9/4/2025, 7:49:18 AM	가로로 긴 테스트가로로 긴 테스트가로로 긴 테스트가로로 긴 테스트가로로 긴...
test	9/3/2025, 9:57:48 PM	test.png입니다
test	9/3/2025, 9:33:03 PM	image_test
0.0.4-test3	9/3/2025, 2:52:02 PM	MQTT를 통한 OTA 테스트를 위하여 배포하는 테스트 펌웨어입니다.
0.0.4-test2	9/3/2025, 4:37:27 AM	MQTT를 통한 OTA 테스트를 위하여 배포하는 테스트 펌웨어입니다.

펌웨어 목록 페이지는 시스템에 등록된 모든 펌웨어의 현황을 한눈에 파악하고 효율적으로 관리할 수 있도록 설계되었다.

사용자가 펌웨어를 직관적으로 인지하고 등록, 조회, 배포 등 후속 작업을 신속하게 진행할 수 있도록 모든 정보는 테이블(Table) 형태로 제공된다. 또한 페이지네이션(Pagination) 기능을 적용하여 페이지의 가독성을 높였다.

페이지 상단에는 검색창을 배치하여 펌웨어 이름, 버전, 릴리즈 노트 등 다양한 조건으로 실시간 검색이 가능하도록 구현했다. 이를 통해 사용자는 원하는 펌웨어를 쉽게 찾을 수 있다.

본 페이지는 '펌웨어 관리' 메뉴의 진입점으로서 메인 페이지 역할을 수행한다. 사용자는 목록의 특정 항목을 클릭하여 해당 펌웨어의 상세 페이지로 이동할 수 있으며, 우측 하단의 '펌웨어 등록' 버튼을 통해 새로운 펌웨어를 등록하는 모달(Modal) 창을 열 수 있도록 하였다.

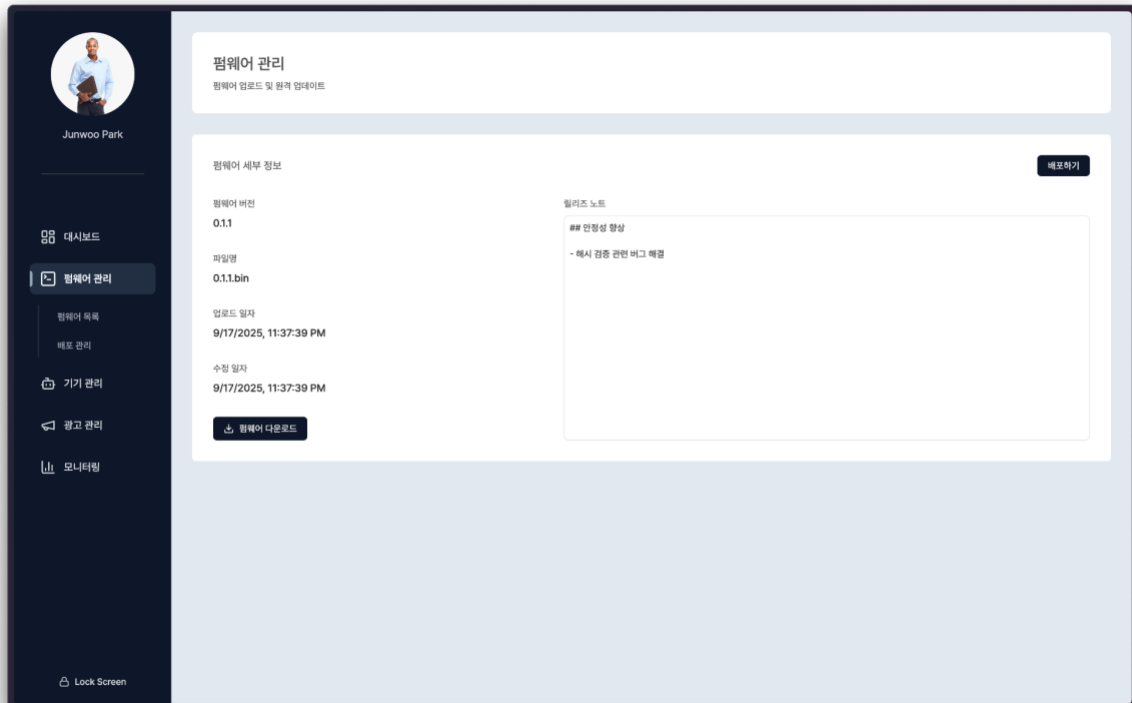


'펌웨어 등록' 버튼을 클릭하면 펌웨어 등록 모달창이 활성화된다.

이때 기존 페이지는 어두운 배경으로 처리하여 사용자가 등록 작업에 집중하도록 유도했으며, 모달 외부 영역을 클릭하면 창이 닫히는 직관적인 사용자 경험을 제공한다.

사용자는 모달 창을 통해 펌웨어 버전, 릴리즈 노트, 펌웨어 파일을 입력 및 업로드할 수 있다. 릴리즈 노트는 상세한 내용 작성이 가능하도록 텍스트 영역(Text Area)으로 구현하였고, 펌웨어 파일은 특정 확장자(.bin, .hex 등)만 업로드할 수 있도록 제한하여 사용자의 실수를 방지하도록 설계했다.

### 3.2.3.2. 펌웨어 세부 정보 페이지

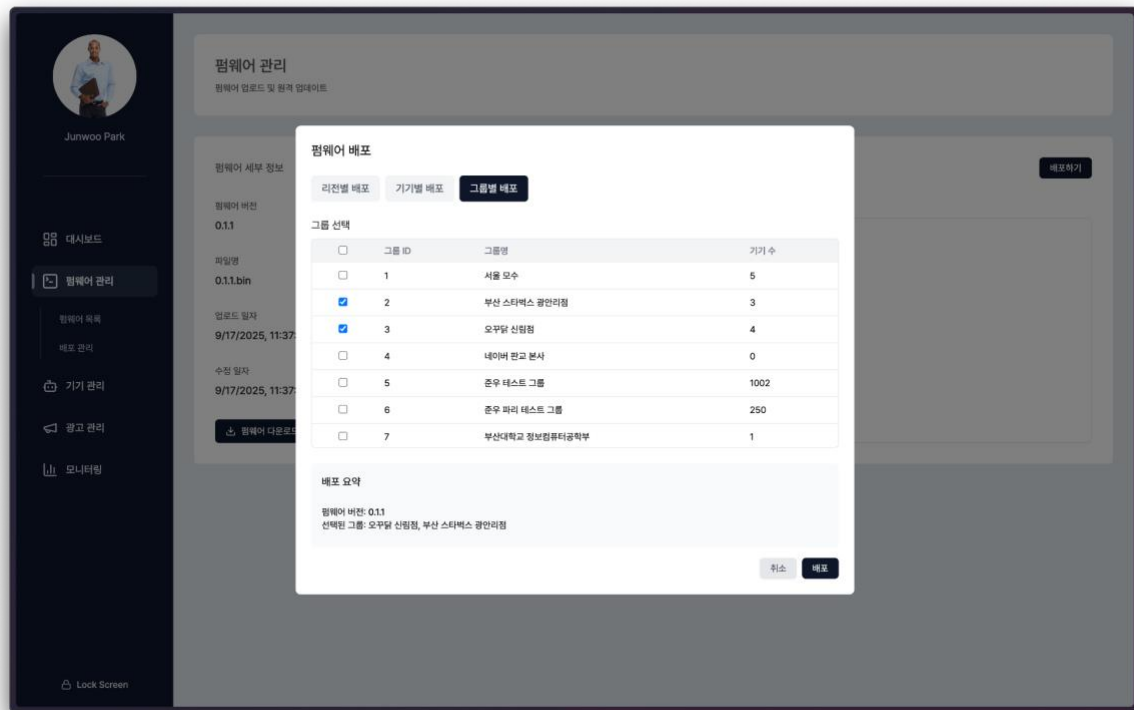


펌웨어 세부 정보 페이지는 특정 펌웨어에 대한 자세한 정보를 조회할 수 있도록 구성하였다. 사용자는 펌웨어 버전, 파일명, 업로드 일자, 수정 일자, 릴리즈 노트를 조회할 수 있고, '펌웨어 다운로드' 버튼을 통해 펌웨어 파일을 직접 다운로드 할 수 있다.

펌웨어 다운로드 로직은 펌웨어가 S3 버킷에 있다는 것과 브라우저 보안 규칙을 고려하여 다음과 같이 구현하였다.

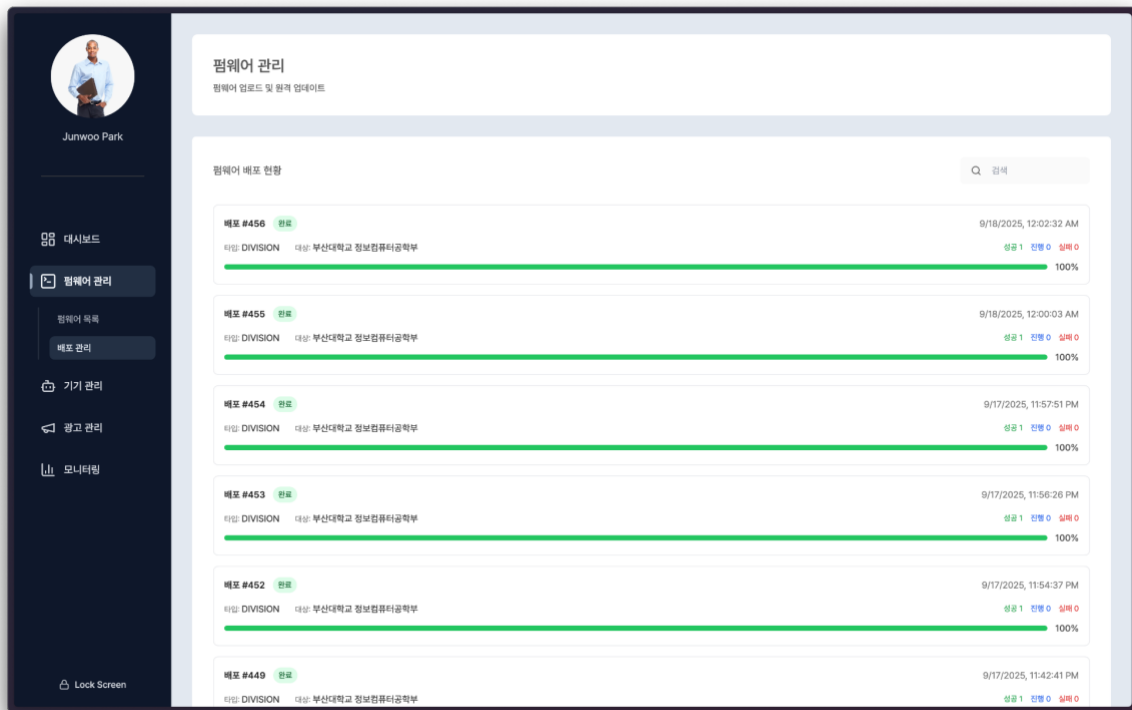
1. 펌웨어의 Presigned URL 을 웹 서버에 요청
2. Presigned URL 을 사용하여 'Blob' 타입으로 다운로드
3. 다운로드 한 파일의 가상 다운로드 링크를 만들고 자동으로 클릭

3 번 단계는 브라우저가 웹 페이지의 스크립트가 사용자 컴퓨터에 직접 파일을 생성하는 것을 허용하지 않는 점을 해결하기 위해 추가된 단계이다.



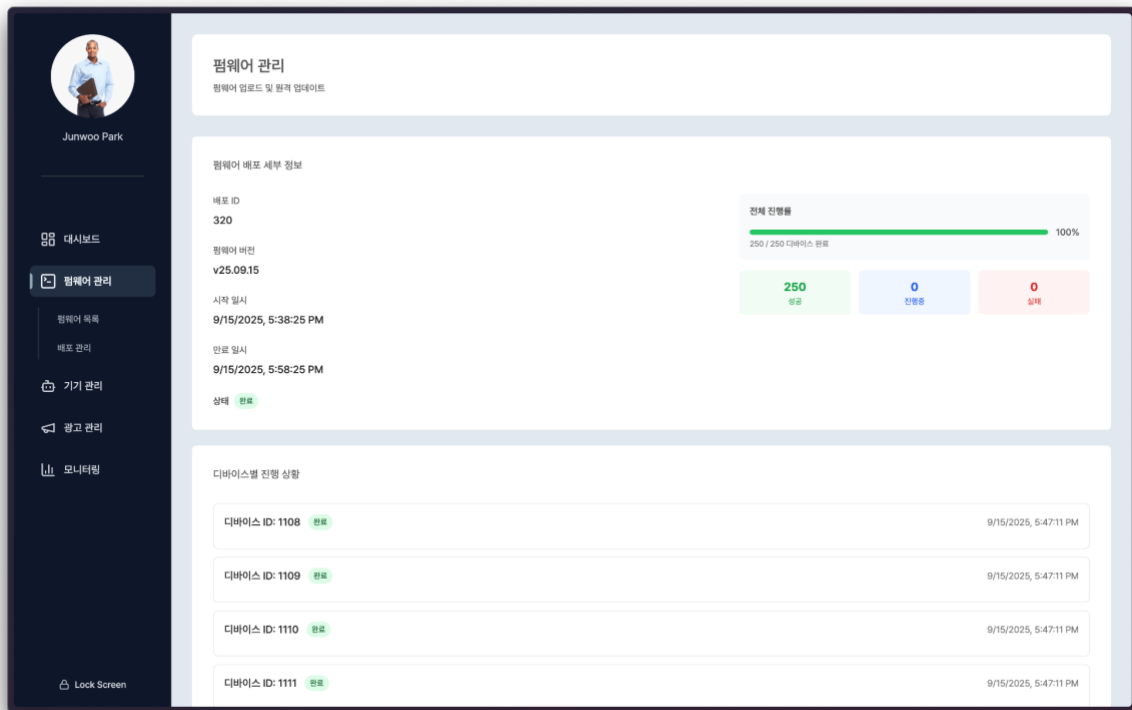
‘배포하기’ 버튼을 클릭하면 펌웨어 배포 설정 모달(Modal) 창이 활성화된다.

사용자는 배포 대상을 리전, 개별 기기, 그룹 중에서 선택할 수 있다. 선택 항목 하단에는 ‘배포 요약’ 영역을 두어 현재 설정된 배포 대상을 명확히 표시하도록 하였다. 이 기능은 관리자가 배포 전 설정 내용을 최종 점검하고 의도치 않은 오배포를 방지하도록 설계되었다.



펌웨어 배포 목록 페이지는 과거에 진행했거나 현재 진행 중인 모든 배포 이력을 조회하고 관리하는 공간이다. 사용자는 이 페이지의 테이블을 통해 배포 ID, 상태(완료, 진행 중 등), 유형(리전, 그룹, 기기), 대상, 진행률 등 각 배포의 핵심 정보를 확인할 수 있다.

특히, 각 배포 항목마다 전체 대상 기기 수와 함께 완료, 진행, 실패한 기기 수를 각각 표시하였다. 또한, 전체 진행률을 상태 바와 함께 시각적으로 제공하여 사용자가 배포 현황을 직관적으로 파악하도록 구성했다. 목록에서 특정 배포 항목을 클릭하면 해당 배포의 상세 정보를 볼 수 있는 페이지로 이동한다.

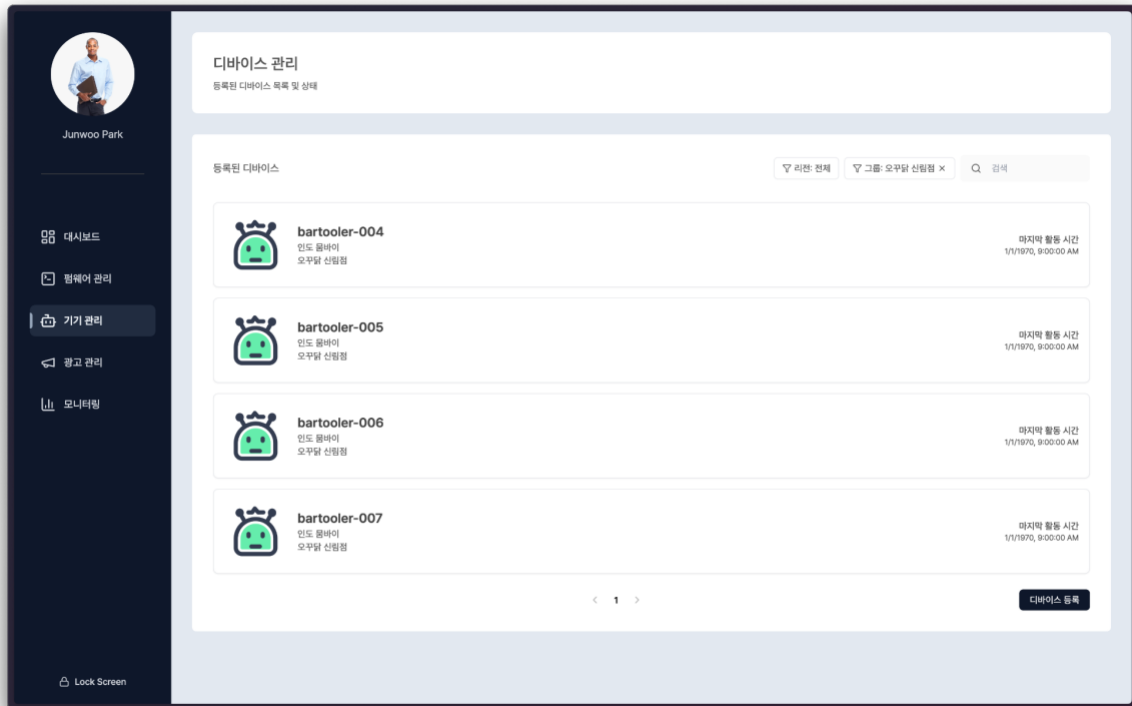


펌웨어 배포 세부 페이지에서는 펌웨어 배포의 세부 정보를 확인할 수 있다. 배포 ID, 배포할 펌웨어 버전, 시작일시 및 만료일시를 확인할 수 있고, 전체 배포 대상 기기 중 완료된 기기, 진행중인 기기, 실패한 기기들의 개수를 우측 상단에서 확인할 수 있다.

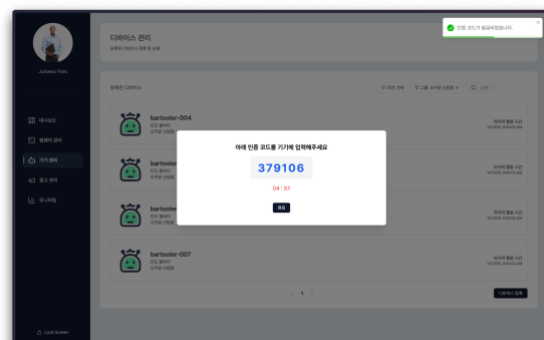
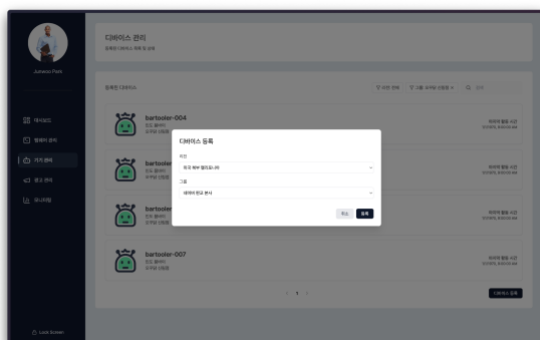
세부 정보 아래에는 디바이스별 진행 상황 타일을 구성하였다. 이 타일에서는 해당 배포의 모든 기기들의 진행 상태를 표시하고, 진행 중인 경우 다운로드 진행률을 추가로 표시하였다.



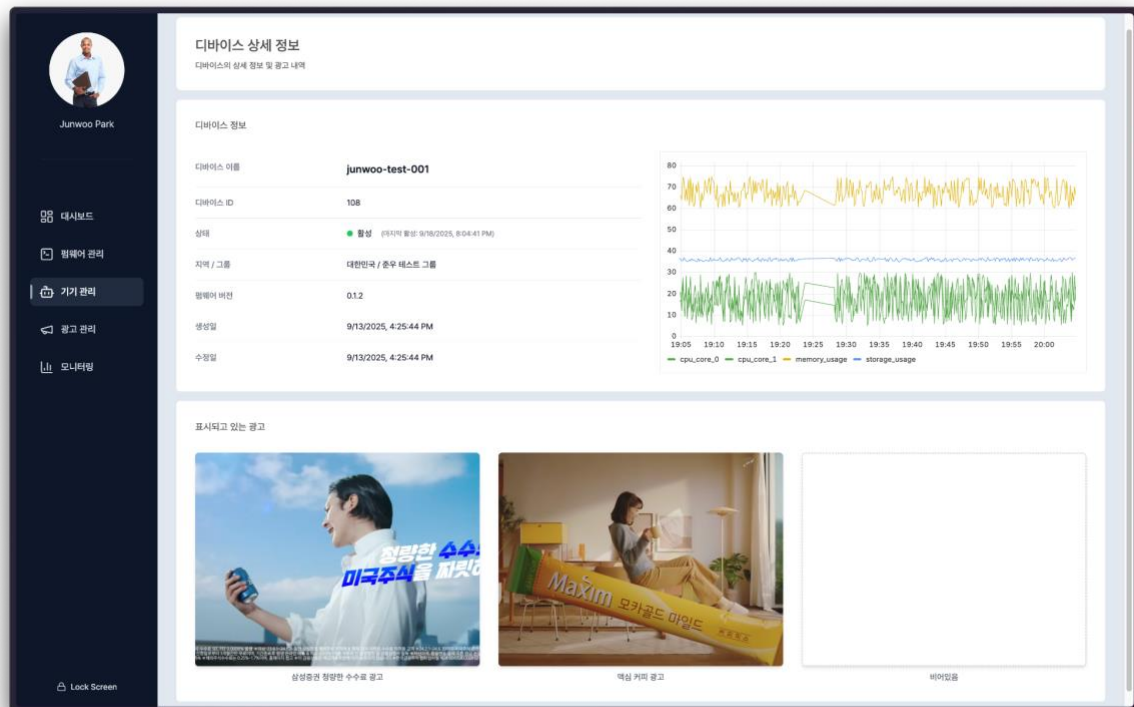
### 3.2.4. 기기 관리 페이지



기기 관리 페이지는 시스템에 등록된 모든 기기를 조회하고 관리하는 공간이다. 사용자는 우측 상단에 위치한 '리전 선택' 및 '그룹 선택' 필터를 사용하여 특정 리전이나 그룹에 속한 기기 목록을 선별적으로 조회할 수 있다. 또한, 우측 하단의 '디바이스 등록' 버튼을 통해 신규 기기를 등록하는 모달 창을 활성화하도록 구현했다.



'디바이스 등록' 버튼을 클릭하면 기기 등록 모달(Modal) 창이 활성화된다. 사용자는 먼저 기기를 할당할 리전과 그룹을 선택한 후 '등록' 버튼을 클릭한다. 이 요청을 받은 백엔드 서버는 기기 인증을 위한 일회용 비밀번호(OTP) 6 자리를 생성하여 반환한다. 모달 창에는 전달받은 OTP 6 자리가 표시되며, 사용자는 이 인증번호를 실제 기기에 입력하여 최종적으로 등록 절차를 완료한다.



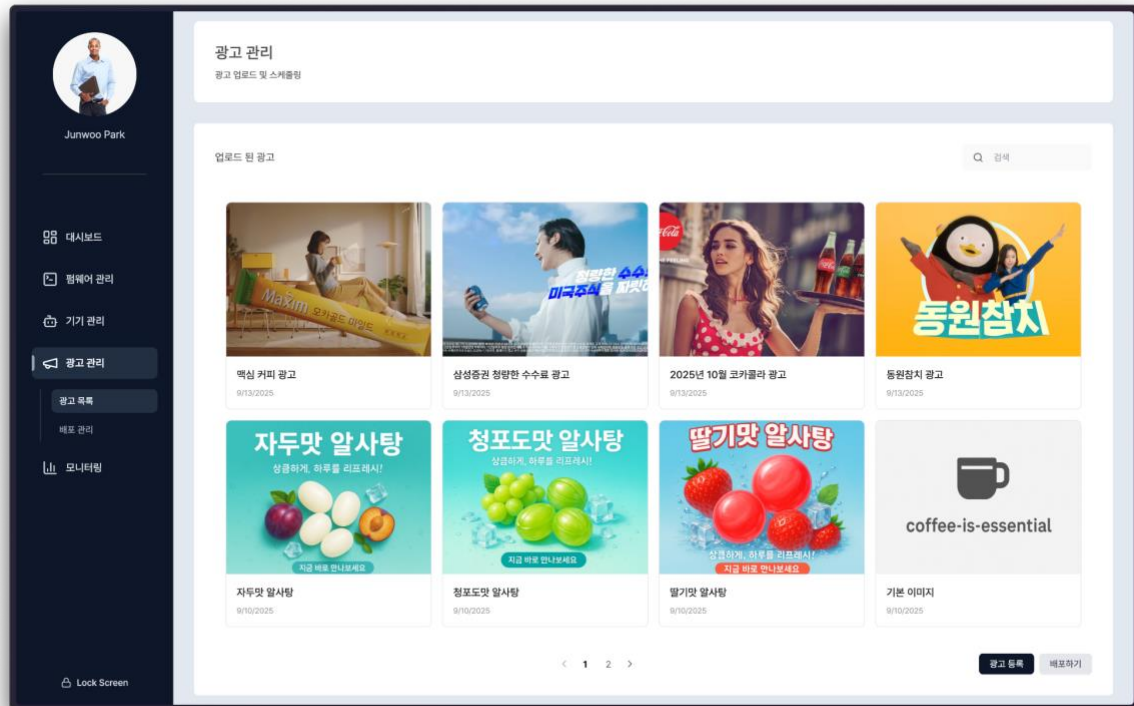
기기 상세 페이지는 개별 기기에 대한 종합적인 상태 모니터링과 펌웨어 및 콘텐츠 사용 현황을 조회하기 위해 설계되었다.

페이지 좌측에서는 디바이스 이름, ID, 리전, 그룹, 생성일 등 기기의 기본 정보를 확인할 수 있으며, 현재 적용된 펌웨어 버전도 함께 표시된다. 우측에는 CPU, 메모리, 디스크 사용량과 같은 주요 메트릭을 실시간 그래프로 시각화하여 모니터링 기능을 제공한다.

또한, 기본 정보 타일 하단에는 해당 기기에 현재 표시되고 있는 콘텐츠 콘텐츠를 미리보기와 함께 제공하여 사용 현황을 파악할 수 있도록 했다.

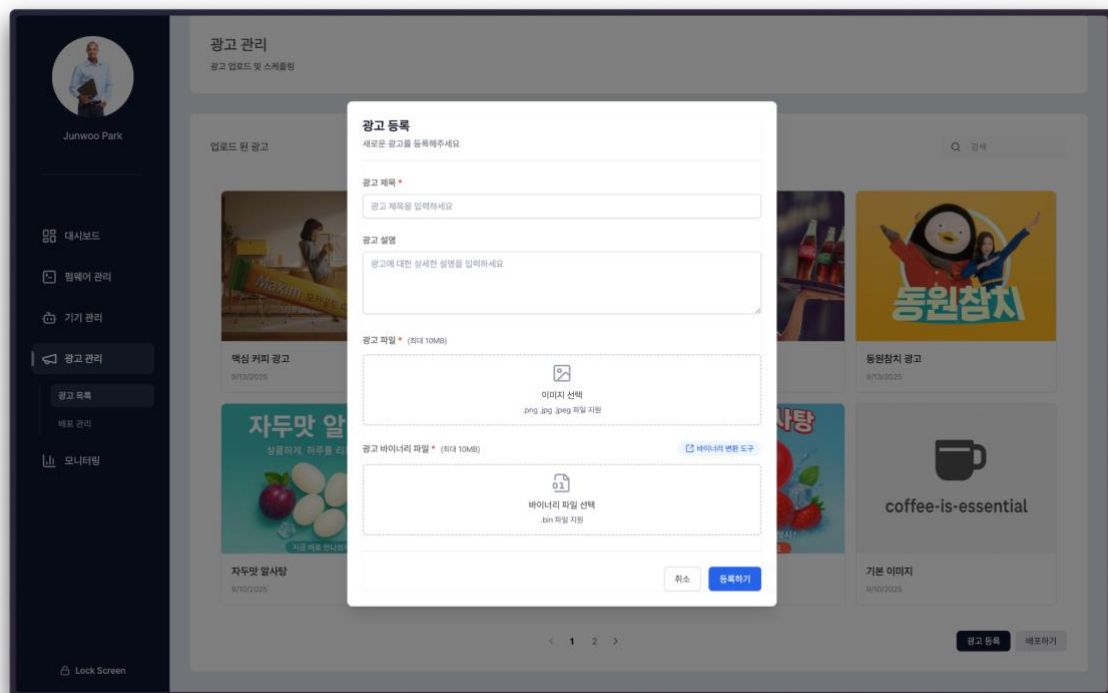
이와 같이 기기의 모든 핵심 정보를 한곳에 통합하여, 사용자가 상태를 한눈에 파악하고 쉽게 모니터링할 수 있도록 구성하였다.

### 3.2.5. 콘텐츠 관리 페이지

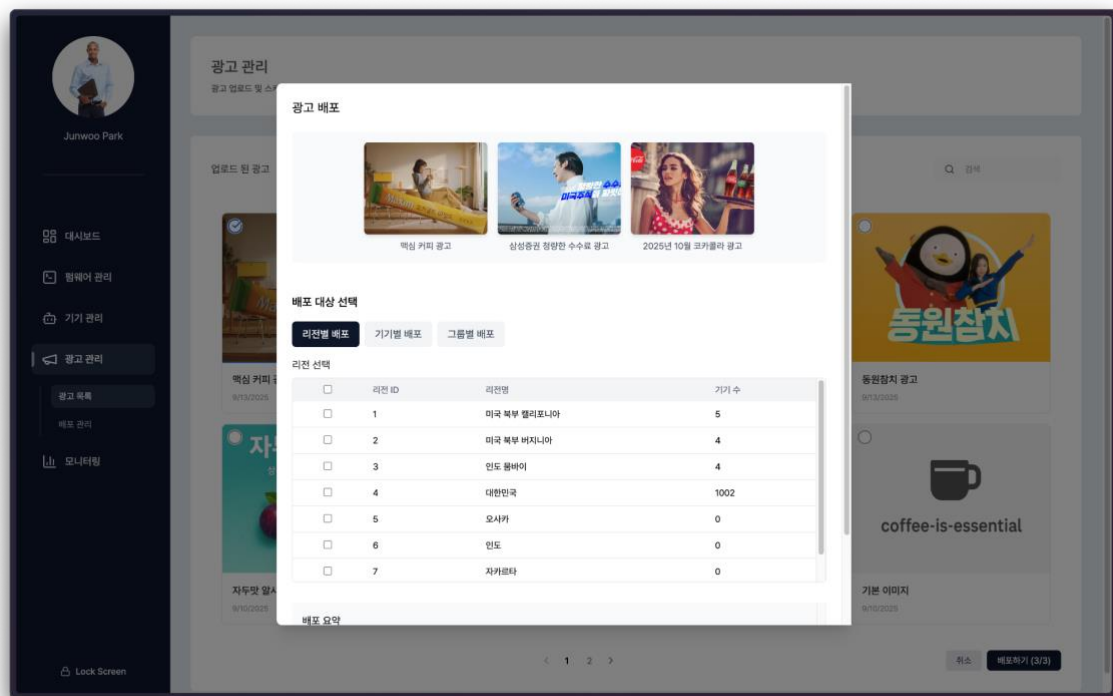


콘텐츠 관리 페이지는 등록된 콘텐츠 목록을 조회하고 신규 콘텐츠를 등록하거나 배포하는 기능을 제공한다. 각 콘텐츠 항목은 백엔드 API로부터 전달받은 Signed URL을 활용하여 이미지 미리보기를 제공하며, 제목과 생성 일자를 함께 표시하여 가시성을 높였다.

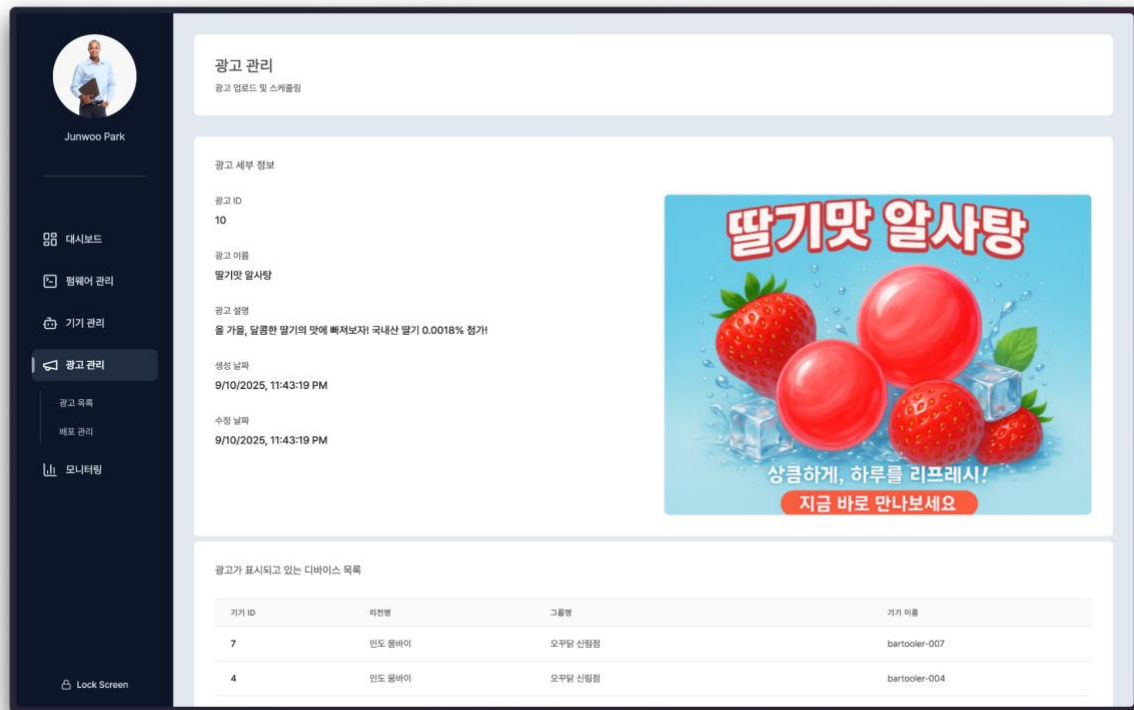
특히 콘텐츠 콘텐츠는 이미지가 중요하므로, 개별 이미지의 시인성을 확보하는 데 중점을 두었다. 한 페이지에 너무 많은 콘텐츠를 표시해 이미지가 작아지는 것을 방지하고자, 페이지당 8개의 콘텐츠를 표시하도록 페이지네이션(Pagination)을 설정했다.



‘콘텐츠 등록’ 버튼을 클릭하면 신규 콘텐츠 등록을 위한 모달(Modal) 창이 활성화된다. 콘텐츠를 등록하려면 제목, 설명, 콘텐츠 이미지 파일, 콘텐츠 바이너리 파일을 모두 업로드해야 한다. 여기서 콘텐츠 바이너리 파일은 IoT 기기의 디스플레이 출력 라이브러리와 호환성을 위해 필수적으로 요구되는 항목이다. 사용자의 편의를 위해, 이미지 파일을 바이너리 파일로 쉽게 변환할 수 있도록 외부 변환 도구 링크를 제공하여 등록 과정을 지원하도록 하였다.



콘텐츠 목록 페이지에서 '배포하기'를 누르면 콘텐츠 선택 모드로 전환된다. 사용자는 최대 3 개의 콘텐츠를 선택한 후 '배포하기' 버튼을 다시 클릭하여 배포 설정 모달(Modal) 창을 활성화할 수 있다. 콘텐츠 배포 대상은 펌웨어 배포와 동일하게 리전, 그룹, 개별 기기 단위로 유연하게 설정할 수 있도록 구현했다. 특히, 관리자의 실수를 방지하기 위해 모달 창 최상단에 선택된 콘텐츠들의 미리보기를 표시하여, 배포 전 내용을 최종 확인할 수 있도록 구성하였다.



콘텐츠 상세 페이지는 특정 콘텐츠의 세부 정보를 조회하는 공간이다. 페이지 상단에서는 콘텐츠 ID, 제목, 설명, 생성 및 수정 일시 등 기본 정보를 확인할 수 있다. 하단에는 '실시간 디바이스 목록' 타일을 배치했다. 이 타일에는 현재 해당 콘텐츠가 실제로 표시되고 있는 모든 기기의 목록을 보여준다. 이를 통해 관리자는 콘텐츠의 실시간 활용 현황을 직관적으로 파악하고 분석할 수 있다.

### 3.2.6. 상태 관리

본 프로젝트의 프론트엔드에서는 두 가지 주요 상태 관리 전략을 채택하였다.

1. 서버 상태 관리: 비동기 데이터, 즉 서버로부터 받아오는 모든 데이터는 @tanstack/react-query 를 사용하여 관리한다.
2. 클라이언트 상태 관리: 일부 전역적으로 필요한 UI 상태는 React 의 내장 기능인 Context API 를 통해 관리한다.

이러한 분리를 통해 서버 데이터와 클라이언트 상태의 관심사를 명확히 분리하고, 각 목적에 가장 적합한 도구를 사용하였다.

---

### 3.2.6.1. 서버 상태 관리

먼저 서버 상태 관리를 위해 @tanstack/react-query 를 도입하였다. @tanstack/react-query 는 API 와의 상호작용을 통해 얻는 서버 상태를 관리하는 핵심 라이브러리로 데이터 캐싱, 동기화, 업데이트를 효율적으로 처리하기 위해 사용된다. 본 프로젝트에서는 비동기 API 요청을 실행한 후 응답이 왔을 때 쿼리를 무효화하여 즉시 데이터를 동기화시키는 데에 적극적으로 활용하였다.

이 라이브러리를 사용하기 위해 애플리케이션 진입점인 App.tsx 에서 QueryClient 를 생성하고, 최상위 컴포넌트를 QueryClientProvider 로 감싸서 프로젝트 전역에서 react-query 의 기능을 사용할 수 있도록 구성하였다.

```
const queryClient = new QueryClient(); // react-query 클라이언트 생성

function App() {
  return (
    <QueryClientProvider client={queryClient}> // 전역에서 react-query 를 사용할 수 있음
      <LockScreenProvider>
        <AppContent />
      </LockScreenProvider>
    </QueryClientProvider>
  );
}

export default App;
```

react-query 를 사용할 때 컴포넌트가 useQuery 나 useMutation 을 직접 사용하는 것을 지양하고, 각 entity 또는 feature 별로 커스텀 훅을 만들어 사용하는 패턴을 적극적으로 활용하였다. 이는 다음과 같은 장점을 이끌어낼 수 있었다.

- queryKey 와 queryFn 등 react-query 관련 로직이 컴포넌트로부터 분리되어 재사용성이 높아진다.
- 컴포넌트는 데이터 Fetching 의 구체적인 구현을 알 필요 없이, 필요한 데이터와 상태만 전달받아 UI 렌더링에 집중할 수 있다.

- 
- API 로직의 중앙 관리 및 유지보수가 용이하다.

커스텀 혹은 다음과 같은 방식으로 설계하였다.

1. 검색어(query), 페이지네이션(page, limit)과 같은 상태는 내부적으로 관리한다.
2. queryKey 는 ["ads", query, page, limit]와 같이 쿼리에 영향을 주는 모든 변수를 포함하여, 데이터가 고유하게 캐싱되도록 설계하였다.
3. placeholderData 옵션을 keepPreviousData 로 설정하여, 페이지 이동 시 이전 데이터를 잠시 보여줌으로써 버벅이는 현상을 줄이고 사용자 경험을 향상시켰다.
4. 혹은 외부로 데이터 목록, 페이지 정보, 로딩 상태, 에러 상태 등 컴포넌트에서 필요한 값들만 정제하여 노출하였다.

데이터 생성, 수정, 삭제 작업은 useMutation 을 기반으로 한 커스텀 훅을 통해 처리하였다.

1. 데이터 변경에 필요한 비동기 로직 전체(예: 파일 업로드, 메타데이터 전송 등)를 mutationFn 에 포함하여 하나의 트랜잭션으로 묶었다.
2. onSuccess 콜백에서 queryClient.invalidateQueries()를 사용하여 쿼리를 무효화하고 데이터가 즉시 동기화될 수 있도록 하였다.

```
// 콘텐츠 등록 비동기 API 요청 이후 쿼리 무효화 예시
// features/ad_register/api/useAdRegister.tsx

{비동기 콘텐츠 등록 API 호출 로직 ... }

return useMutation({
  mutationFn: uploadAd,
  onSuccess: () => {
    queryClient.invalidateQueries({ queryKey: ["ads"] }); // 쿼리 무효화
  },
  onError: (error) => {
    console.error("콘텐츠 업로드에 실패했습니다:", error);
  },
});
```



```
});
```

### 3.2.6.2. 클라이언트 상태 관리

서버 상태가 아닌, 순수한 UI 상태나 여러 컴포넌트가 공유해야 하는 전역 상태는 Context API 를 사용하여 관리하였다.

대표적으로 화면 잠금 컴포넌트가 있는데, 아래 예시와 같이 구성되었다.

1. 화면 잠금(isLocked)와 같은 전역 UI 상태를 관리하기 위해 LockScreenContext 를 생성하였다.
2. LockScreenProvider 컴포넌트를 통해 하위 컴포넌트들에게 상태와 상태 변경 함수(lock, unlock)를 제공한다.
3. localStorage 와 연동하여 브라우저를 새로고침해도 잠금 상태가 유지되도록 하였다.
4. useLockScreen 이라는 Custom Hook 을 제공하여 컨텍스트를 더욱 쉽고 안전하게 사용할 수 있도록 하였다.

### 3.2.7. 사용자 UX 개선

#### 3.2.7.1. 토스트 메시지를 통한 사용자 경험 향상

본 프로젝트의 API 중에는 펌웨어 배포나 파일 업로드와 같이 응답 시간이 긴 비동기 작업이 다수 포함되어 있었다. 초기에는 await 구문을 사용해 API 요청이 완료될 때까지 대기하도록 구현했으나, 이는 사용자가 응답을 기다리는 동안 아무런 피드백을 받지 못해 사용자 경험을 저해하는 문제가 있었다.

이 문제를 해결하기 위해 react-toastify 라이브러리를 도입했다. toast.promise 기능을 활용하여 useMutation 으로 처리되는 비동기 작업의 대기, 성공, 실패 상태를 사용자에게 명확하게 시각적으로 안내하도록 개선했다. 이를 통해 사용자는 작업 요청에 대한 즉각적인 피드백을 받고 진행 상태를 쉽게 파악할 수 있게 되었다.

펌웨어 배포 요청은 이 방식을 적용한 대표적인 예시이며, 관련 로직은 다음과 같다.

```
await toast.promise(  
  deployFirmware({
```



---

프로젝트의 핵심 목표는 수많은 IoT 기기에 펌웨어 및 콘텐츠를 안정적이고, 안전하며, 확장 가능하게 배포하는 것이다. 이를 위해 고가용성과 강화된 보안이라는 두 가지 핵심 원칙에 중점을 두고 아키텍처를 설계했다.

### 3.3.2. 고가용성 및 확장성 구현

대규모 IoT 트래픽을 안정적으로 처리하고, 장애 발생 시에도 서비스 연속성을 보장하기 위해 다음과 같은 기술과 전략을 채택했다.

#### 3.3.2.1. 마이크로서비스 컨테이너 오케스트레이션

서버 프로비저닝 및 관리의 운영 부담을 줄이고 애플리케이션 개발에 집중하고자, 주요 서비스를 컨테이너화하여 Amazon ECS(Elastic Container Service)에서 실행하도록 구성했다.

특히 서버리스 엔진인 Fargate 를 시작 유형으로 채택하여 EC2 인스턴스 클러스터를 직접 관리할 필요성을 제거했다. 이를 통해 트래픽 변화에 따른 유연한 확장/축소가 가능해졌고, 인프라 관리의 복잡성을 크게 낮추는 효과를 얻었다.

ECS 기반의 마이크로서비스 아키텍처를 효과적으로 구현하기 위해, 모든 서비스를 Docker 컨테이너로 표준화했다. 각 서비스의 Dockerfile 을 생성하고 외부 설정은 환경 변수로 주입받도록 설계했으며, Terraform 이 ECS Task 실행 시점에 이 환경 변수를 동적으로 전달하도록 자동화했다.

결과적으로, terraform apply 명령어 하나만으로 각 서비스에 필요한 환경 변수와 의존성 설정이 자동으로 완료되어, 개발 및 재배포의 편의성과 속도를 확보할 수 있었다. 아래는 웹 서버(Spring Boot)의 ECS 서비스 구성 예시이다.

```
# backend.tf - 백엔드 서비스 정의 예시
resource "aws_ecs_service" "backend" {
  name           = "backend"
  cluster        = aws_ecs_cluster.main.id
  task_definition = aws_ecs_task_definition.backend.arn
  desired_count  = 1
  launch_type    = "FARGATE"
```

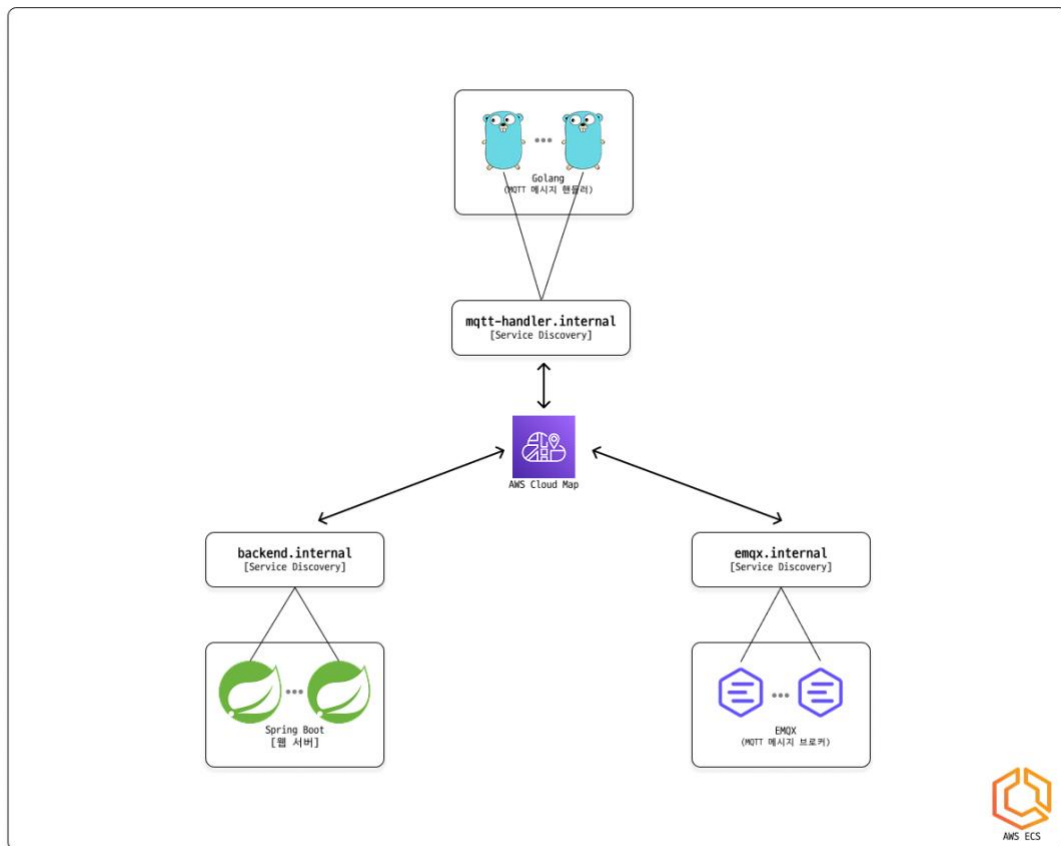
```
network_configuration {  
  subnets      = [aws_subnet.private_a.id, aws_subnet.private_b.id]  
  security_groups = [aws_security_group.backend.id,  
aws_security_group.ecs_internal_services.id]  
  assign_public_ip = false  
}  
# ...  
}
```

ECS 서비스는 고가용성을 위해 여러 개의 태스크(컨테이너)로 실행되는 경우가 일반적이다. 각 태스크는 고유하지만 동적인 사설 IP 주소를 할당받기 때문에, 다른 서비스가 특정 태스크의 IP 를 미리 알고 통신하는 것은 불가능에 가깝다.

이 문제를 해결하기 위해 Amazon ECS 서비스 디스커버리(Service Discovery) 기능을 도입했다.

이 기능은 AWS Cloud Map 과 통합되어 동작한다. ECS 서비스가 새 태스크를 시작하면, 해당 태스크의 사설 IP 주소를 Cloud Map 에 서비스 이름과 함께 자동으로 등록한다. 반대로 태스크가 중지되거나 비정상 상태가 되면 목록에서 자동으로 제거된다.

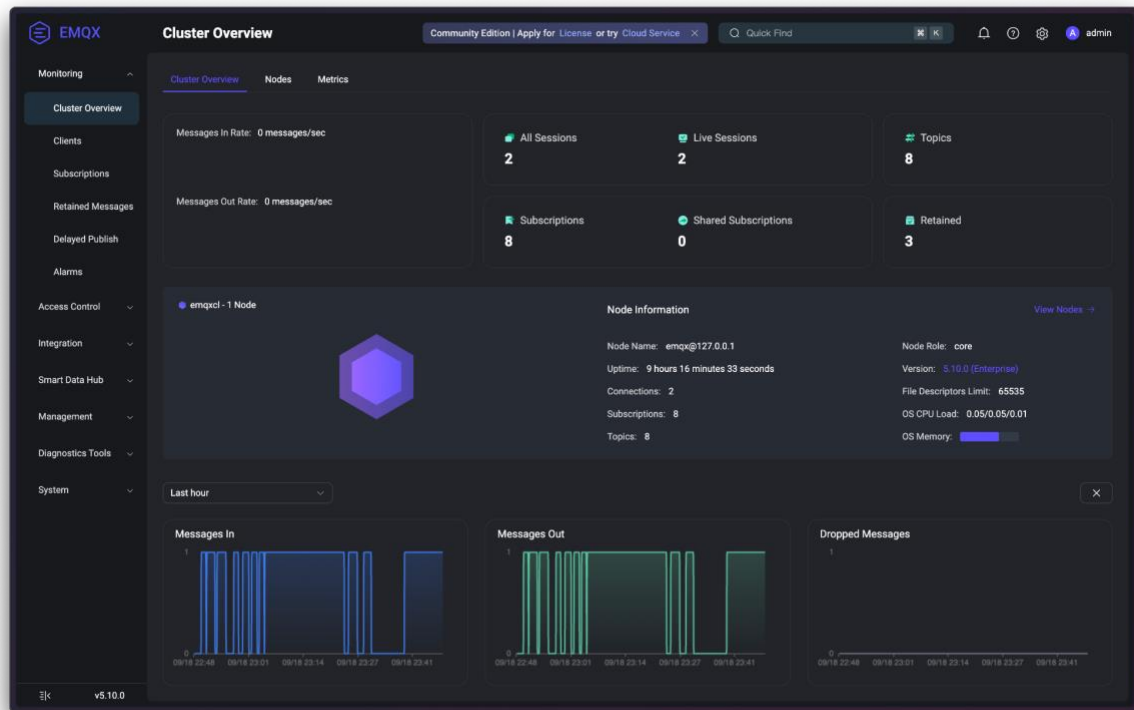
결과적으로, 다른 서비스는 더 이상 개별 태스크의 IP 주소를 추적할 필요가 없다. 대신, 예를 들어 `mqtt-handler.local` 과 같이 사전에 정의된 고유한 서비스 이름(DNS)을 호출하기만 하면, AWS Cloud Map 이 알아서 현재 실행 중인 정상 상태의 태스크 IP 중 하나로 요청을 연결해 준다. 이를 통해 마이크로서비스 아키텍처에서 서비스 간 통신이 유연하고 안정적으로 유지되도록 구성할 수 있었다. 간단히 도식화한 예시는 아래와 같다.



### 3.3.2.2. 대규모 MQTT 트래픽 처리를 위한 MQTT 브로커 구성

수십만 대 이상의 IoT 기기에서 발생하는 대규모 동시 연결 및 메시지 트래픽을 처리하는 것은 이 프로젝트의 핵심 요구사항이었다. 이를 해결하기 위해 고성능 MQTT 브로커인 EMQX를 ECS에 배포하고, 그 앞단에 Network Load Balancer(NLB)를 배치하는 아키텍처를 선택했다.

HTTP/HTTPS 트래픽에 최적화된 Application Load Balancer(ALB)와 달리, NLB는 TCP/UDP 트래픽을 처리하는 Layer 4 로드 밸런서로, 대량의 영구적인 TCP 연결을 유지해야 하는 MQTT 프로토콜에 훨씬 적합하다고 판단하였다. NLB를 사용함으로써 연결 상태를 유지하면서 높은 처리량과 낮은 지연 시간을 달성할 수 있었고, 이는 대규모 IoT 환경의 안정성을 확보하는 데 결정적인 역할을 했다.



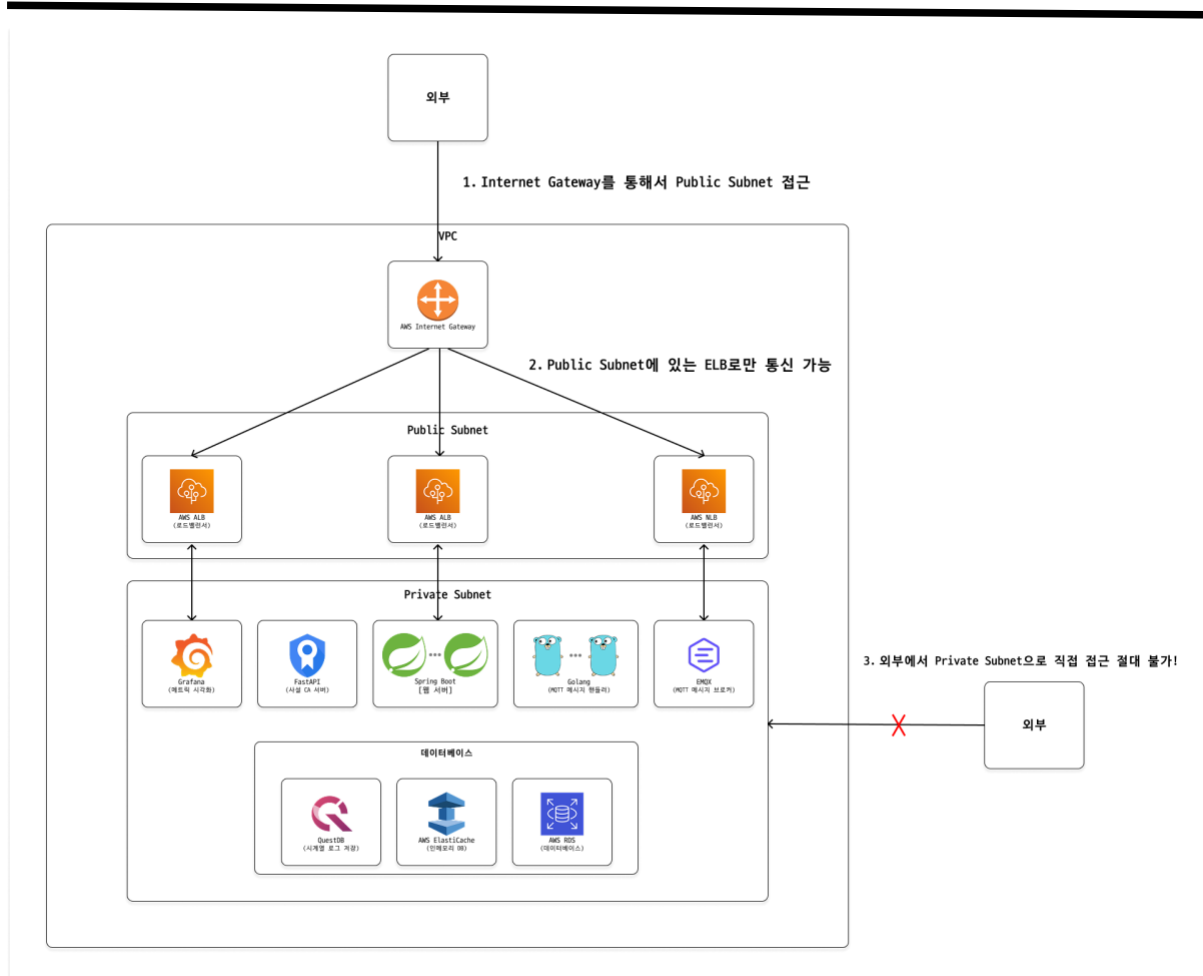
MQTT 브로커로 EMQX 를 도입함으로써, 높은 성능뿐만 아니라 강력한 관리 기능이라는 이점도 확보했다. EMQX 는 기본적으로 웹 대시보드를 제공하는데, 18883 포트를 통해 이 대시보드에 접속하면 현재 연결된 MQTT 클라이언트의 세션을 관리하고 주요 지표를 실시간으로 모니터링할 수 있었다. 이를 통해 별도의 개발 없이도 효율적인 운영 및 관리가 가능해졌다.

### 3.3.3. 강화된 보안 아키텍처 구현

IoT 환경에서는 기기, 네트워크, 클라우드에 이르는 모든 계층에서 강력한 보안이 필수적이다. 본 프로젝트는 외부 공격으로부터 내부 시스템을 보호하고, 허가된 기기만이 안전하게 통신할 수 있도록 다층적인 보안 전략을 적용했다.

#### 3.3.3.1. VPC 설계

보안 설계의 가장 기본 원칙은 공격 표면을 최소화하는 것이다. 본 프로젝트에서는 Virtual Private Cloud(VPC)를 구성하고 그 안에 Public Subnet A, B 와 Private Subnet A, B 를 구성하였다. 아래는 본 프로젝트의 네트워크 격리 수준을 나타낸 것이다.



인터넷에서 직접 접근할 필요가 없는 모든 핵심 리소스들을 Private Subnet 에 배치하였다. 데이터베이스(RDS), 캐시(ElastiCache), 그리고 모든 ECS 서비스들이 여기에 해당된다. 외부와의 통신은 반드시 필요한 경우에만 Public Subnet 에 위치한 로드 밸런서(ALB, NLB)나 NAT 인스턴스를 통해서만 이루어지도록 설계하여, 내부 시스템을 외부 위협으로부터 원천적으로 격리했다. 이 구조는 인프라 보안의 가장 중요한 첫 번째 방어선 역할이라고 볼 수 있다.

### 3.3.3.2. 사실 CA 를 이용한 상호 인증 (mTLS)

VPC 설계가 네트워크 수준의 격리였다면, 사실 CA 를 이용한 상호 인증(mTLS)은 신원이 확인된 기기만 접속을 허용하는 애플리케이션 수준의 인증 체계이다. 이는 기기를 위장한 악의적인 메시지 전송이나 도청을 방지하는 것을 목적으로 한다.

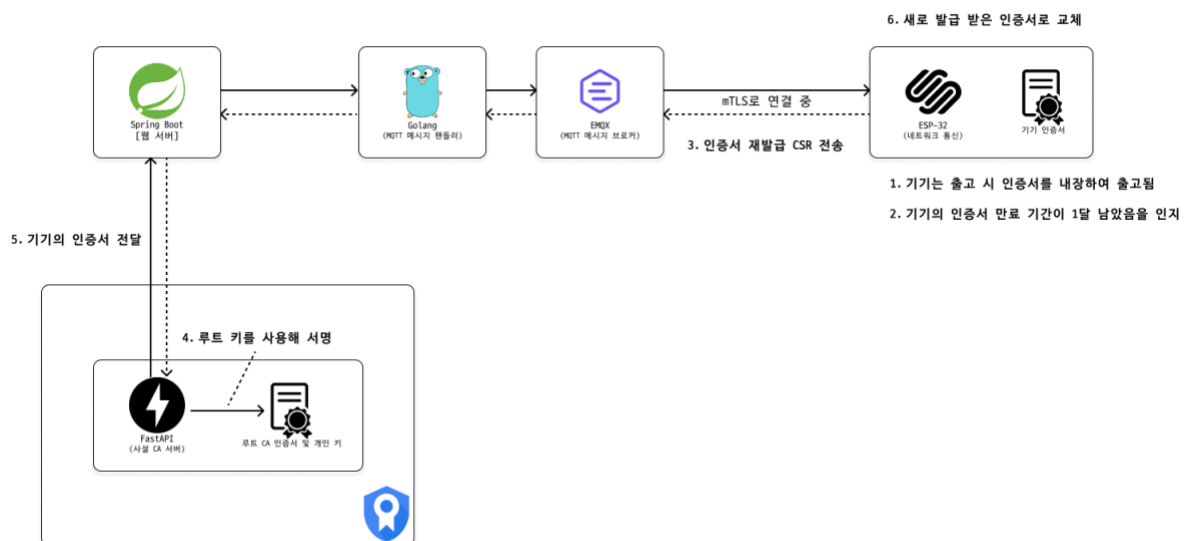
이를 위해 서버만 인증하는 단방향 TLS 를 넘어, 기기(클라이언트)와 서버가 서로의 신원을 확인하는 mTLS(Mutual TLS) 방식을 도입했다. mTLS 구현의 핵심인 인증 기관은 ECS

컨테이너 기반의 사설 CA(Private CA) 서비스를 직접 구축하여 활용했다. 이 사설 CA 는 각 IoT 기기에 고유한 클라이언트 인증서를 발급하고, MQTT 브로커(EMQX)는 이 인증서의 유효성을 검증하여 허가된 기기의 접속만 허용한다.

인증서를 발급받는 로직은 다음과 같다.

1. 초기 발급: 기기 출고 시, 사설 CA 가 발급한 고유 클라이언트 인증서를 내장하여 출고한다.
2. 갱신 요청: 기기는 인증서 만료 1 개월 전, 갱신을 요청하는 MQTT 메시지를 전송한다.
3. 요청 전달: 해당 MQTT 메시지는 EMQX → MQTT Handler → 웹 서버를 거쳐 사설 CA 로 전달된다.
4. 신규 발급: 사설 CA 는 요청을 검증하고, 새로운 인증서를 서명하여 발급한다.
5. 인증서 전달: 발급된 새 인증서는 웹 서버 → MQTT Handler → EMQX 를 거쳐 해당 기기로 다시 전달된다.
6. 인증서 교체: 기기는 수신한 새 인증서로 기존 인증서를 교체하여 통신을 지속한다.

위 과정이 일어나는 동안 MQTT 는 여전히 mTLS 를 사용하여 연결이 유지되고 있으므로, 안전하게 인증서를 교체할 수 있다.



이 방식을 통해 인증되지 않은 기기의 접속을 원천 차단하고, 중간자 공격(Man-in-the-Middle)과 같은 심각한 보안 위협을 효과적으로 방어할 수 있었다.



### 3.3.3.3. CloudFront Signed URL 을 통한 안전한 펌웨어 배포

펌웨어 파일과 같은 민감한 콘텐츠를 아무나 다운로드할 수 없도록 보호하는 것은 필수적이다. S3 버킷을 공개로 설정하는 대신, CloudFront Signed URL 을 사용하는 방식을 채택했다. 펌웨어 파일이 저장된 S3 버킷은 비공개로 유지하고, CloudFront 를 통해서만 접근할 수 있도록 설정했다. 이때, CloudFront 배포 설정에 `trusted_key_groups` 를 지정하여 서명된 요청만을 신뢰하도록 구성했다.

기기가 펌웨어 업데이트를 요청하면, 백엔드 서비스는 이 신뢰 키 그룹의 비공개 키를 사용하여 짧은 만료 시간을 가진 Signed URL 을 동적으로 생성하여 기기에 전달한다. 이 URL 을 통해서만 펌웨어 다운로드가 가능하므로, 허가되지 않은 사용자의 파일 접근을 막고 콘텐츠를 안전하게 보호하는 효과를 얻었다. CloudFront 는 다음과 같이 구성하였다.

```
# buckets.tf - 서명된 URL 을 요구하는 CloudFront 배포 설정
resource "aws_cloudfront_distribution" "firmware_distribution" {
  # ...
  default_cache_behavior {
    # ...
    viewer_protocol_policy = "redirect-to-https"
    trusted_key_groups     = [aws_cloudfront_key_group.signing_key_group.id]
  }
  # ...
}

resource "aws_cloudfront_key_group" "signing_key_group" {
  name = "cloudfront-signing-key-group"
  items = [aws_cloudfront_public_key.signing_key.id]
}
```

### 3.3.3.4. 최소 권한 원칙 기반의 Security Group 설정

네트워크 격리만으로는 내부망에서의 위협에 취약할 수 있다. 따라서 각 서비스(리소스)가 통신할 수 있는 대상을 엄격하게 제한하는 최소 권한의 원칙을 적용했다. AWS 의 Security Group 을 사용하여, 각 서비스는 명시적으로 허용된 대상하고만, 그리고 꼭 필요한 포트와

---

프로토콜로만 통신할 수 있도록 설정했다.

예를 들어, 아래 코드는 데이터베이스(MySQL)의 보안 그룹 설정이다.

```
# security_groups.tf - RDS(MySQL) 보안 그룹 정의
resource "aws_security_group" "mysql_sg" {
  name     = "iot-cloud-ota-mysql-sg"
  vpc_id   = aws_vpc.main.id

  ingress {
    from_port = 3306
    to_port   = 3306
    protocol  = "tcp"
    security_groups = [
      aws_security_group.bastion_sg.id,
      aws_security_group.backend.id,
    ]
    description = "MySQL access from trusted security groups"
  }

  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

ingress 규칙을 보면, 오직 백엔드 서비스(aws\_security\_group.backend.id)와 관리용 Bastion Host(aws\_security\_group.bastion\_sg.id) 등 허가된 보안 그룹으로부터의 3306 포트 TCP 요청만을 허용하고 있다. 만약 다른 서비스가 해킹되더라도 데이터베이스에 직접 접근하는 2차 피해를 막을 수 있는 중요한 보안 장치 역할을 하였다.

---

### 3.3.4. 기타 리소스 정의

#### 3.3.4.1. 시계열 데이터 저장을 위한 QuestDB 구성

IoT 환경에서는 기기의 상태, 센서 값, 펌웨어 업데이트 이력 등 시간의 흐름에 따라 대량으로 발생하는 데이터, 즉 시계열 데이터(Time-Series Data)가 주를 이룬다. 이러한 데이터를 효율적으로 처리하기 위해 범용적인 RDBMS(관계형 데이터베이스) 대신 시계열 데이터 처리에 특화된 QuestDB 를 도입했다.

펌웨어 배포 이력이나 기기의 주기적인 상태 보고와 같은 데이터는 '추가'는 매우 빈번하지만 '수정'이나 '삭제'는 거의 발생하지 않는 특성을 가진다. RDBMS 에 이러한 데이터를 대량으로 저장할 경우, 시간이 지남에 따라 인덱싱과 조회 성능이 저하될 수 있다. 반면, QuestDB 와 같은 시계열 데이터베이스(TSDB)는 빠른 속도로 데이터를 수집(Ingestion)하고, 시간 범위에 기반한 복잡한 집계 및 분석 쿼리를 매우 효율적으로 수행하도록 설계되었다. 대규모 기기로부터 수집되는 데이터를 실시간으로 저장하고, Grafana 를 통해 즉각적으로 시각화 및 분석할 수 있는 강력한 모니터링 시스템의 기반을 마련하기 위해 QuestDB 를 구성하였다.

QuestDB 는 EC2 인스턴스 위에서 Docker 컨테이너로 실행된다. 이때 발생할 수 있는 문제는 EC2 인스턴스가 장애로 종료되거나 교체될 경우 컨테이너 내부에 저장된 모든 데이터가 유실될 수 있다는 점이다. 이러한 문제를 해결하고 데이터의 영속성을 보장하기 위해, 데이터의 저장 위치를 EC2 인스턴스로부터 분리하는 전략을 사용했다.

EC2 인스턴스와는 별개의 생명주기를 가지는 EBS(Elastic Block Store) 볼륨을 생성하고, 이를 QuestDB 를 실행할 EC2 인스턴스에 연결(Attach)했다. 그리고 QuestDB 컨테이너를 실행할 때, 데이터가 저장되는 경로를 이 EBS 볼륨의 마운트 경로로 지정했다. 이 방식을 통해 EC2 인스턴스가 교체되더라도, 새로운 인스턴스에 기존 EBS 볼륨을 다시 연결하기만 하면 모든 과거 데이터를 그대로 보존할 수 있게 되었다. 이는 데이터의 안정성과 내구성을 확보하는 핵심적인 설계 중 하나이다.

QuestDB EC2 인스턴스가 시작될 때 볼륨을 마운트하고 Docker 컨테이너의 데이터 경로로 사용하도록 만들기 위해 user\_data 에 관련 쉘 명령어를 추가하였다.

```
resource "aws_instance" "questdb" {  
  # ...  
  user_data = <<-EOF
```

```
#!/bin/bash
# ... (볼륨 포맷 및 마운트)
mkdir -p /data
mount /dev/nvme1n1 /data

# ... (도커 설치)

# QuestDB 컨테이너 실행 시, 호스트의 /data 경로(EBS 볼륨)를
# 컨테이너의 데이터 저장 경로인 /var/lib/questdb 로 마운트
docker run -d --name questdb \
  -p 9000:9000 -p 8812:8812 \
  -v /data:/var/lib/questdb \
  questdb/questdb:9.0.3
EOF
}
```

### 3.3.5. 개발 편의성을 위한 데브옵스(DevOps)

#### 3.3.5.1. CI/CD 파이프라인

본 프로젝트는 여러 마이크로서비스가 하나의 리포지토리에서 관리되는 모노레포(Monorepo) 구조를 채택하고 있다. 모든 서비스는 Docker 컨테이너로 빌드되어 Amazon ECS 에서 실행되므로, 코드 변경 사항을 안정적이고 신속하게 통합하고 배포하는 자동화된 파이프라인은 필수적이다. 수동 배포는 실수를 유발하기 쉽고, 서비스 간의 의존성 관리를 복잡하게 만들어 전체 개발 및 운영 효율성을 저하시키기 때문이다.

이를 해결하기 위해, GitHub Actions 를 활용하여 `main` 브랜치에 코드가 푸시될 때마다 자동으로 빌드와 배포가 이루어지는 CI/CD 파이프라인을 구축했다.

CI/CD 파이프라인은 다음과 같이 동작하도록 구성하였다.

1. 변경 감지: 파이프라인이 시작되면, 가장 먼저 어떤 서비스의 코드가 변경되었는지 자동으로 감지한다. 이는 dorny/paths-filter 액션을 사용하여 구현되었다. 이 단계를 통해 변경되지 않은 서비스는 불필요하게 재빌드 및 재배포되지 않으므로, 리소스를 절약하고 배포 시간을 크게 단축한다.

```
# .github/workflows/ci_dev.yml - 변경 감지 부분
```

```
- uses: dorny/paths-filter@v2
```

```
  id: filter
```

```
  with:
```

```
    filters: |
```

```
      backend:
```

```
        - 'backend/**'
```

```
      emqx:
```

```
        - 'emqx/**'
```

```
      # ... (기타 서비스)
```

2. 병렬 빌드 및 ECR 푸시 (Parallel Build & Push): 변경이 감지된 각 서비스에 대해, 독립적인 빌드 작업이 병렬로 실행된다. 각 작업은 해당 서비스의 Dockerfile 을 사용하여 Docker 이미지를 빌드하고, 고유한 식별을 위해 Git 커밋 SHA 를 태그로 붙여 Amazon ECR(Elastic Container Registry)에 푸시한다. 이 방식은 전체 빌드 시간을 최소화하고, 어떤 코드가 어떤 이미지에 해당하는지 명확하게 추적할 수 있게 해준다.
3. Terraform 을 이용한 자동 배포 (Terraform Apply): 모든 빌드 작업이 성공적으로 완료되면, 마지막 배포 단계가 실행된다. 이 단계에서는 Terraform 을 사용하여 인프라 변경 사항을 적용한다. 특히, 이전 단계에서 ECR 에 푸시된 새로운 이미지의 태그(Git 커밋 SHA)를 변수로 받아 terraform apply 명령을 실행한다. 이때, 파이프라인은 변경된 서비스의 이미지 태그만 terraform apply 명령의 변수로 전달한다. 결과적으로 Terraform 은 해당 ECS 서비스의 작업 정의(Task Definition)만 새로운 이미지 버전으로 업데이트하여, 효율적이고 정확한 배포를 수행할 수 있도록 하였다.

```
# .github/workflows/ci_dev.yml - Terraform 배포 부분
```

```
- name: Terraform Apply
```

```
  env:
```

```
    BACKEND_IMAGE: ${ needs.ci-backend-dev.outputs.tag }
```

```
    # ... (기타 서비스 이미지 태그)
```

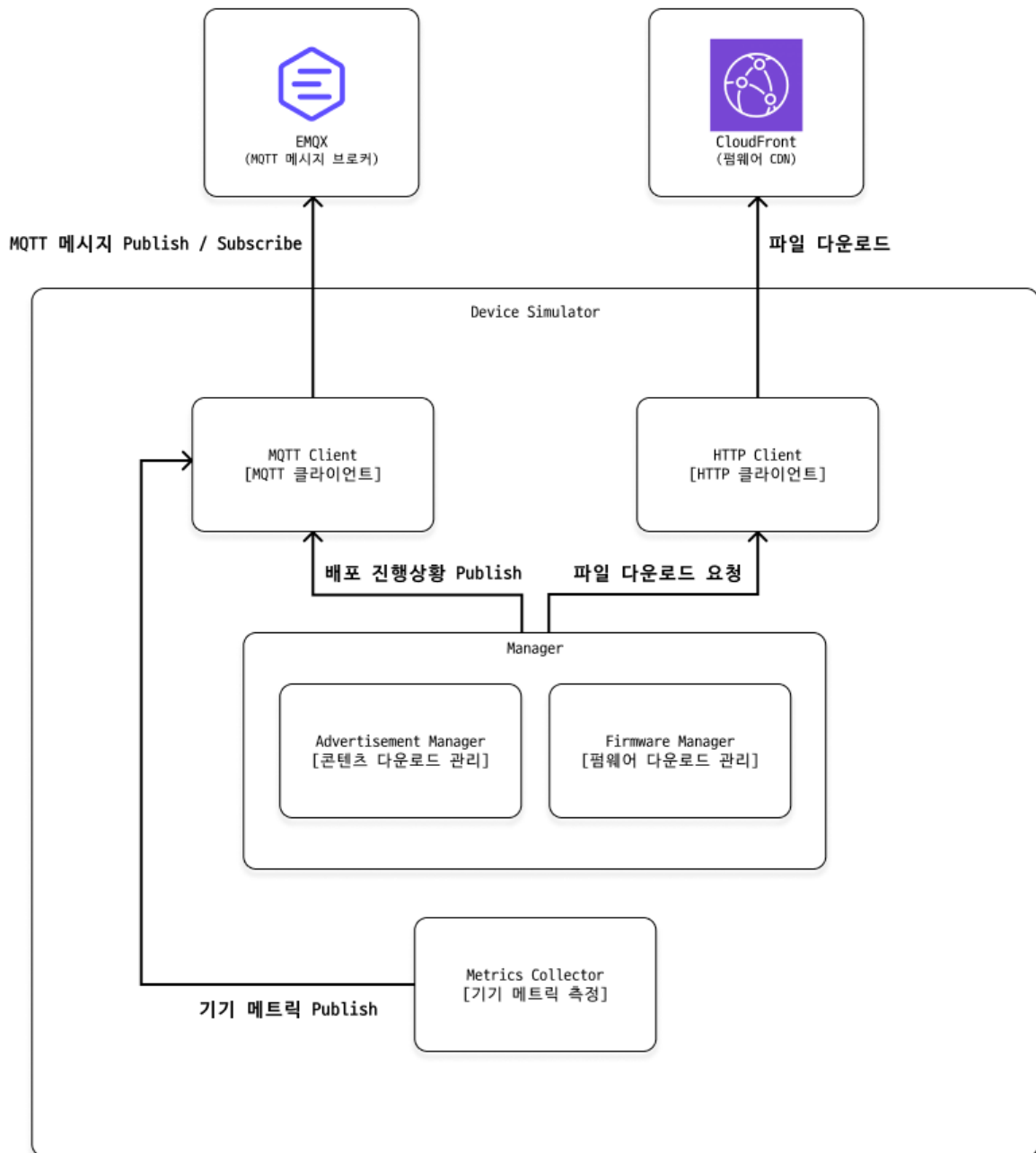
```
run: |
  image_tags=""
  if [ -n "$BACKEND_IMAGE" ]; then
    image_tags="$image_tags -var backend_image_tag=$BACKEND_IMAGE"
  fi
  # ... (변경된 다른 이미지에 대한 변수 추가)

  terraform apply -auto-approve $image_tags
```

CI/CD 파이프라인을 구성한 결과, 현재 기준으로 총 332 번의 Job 이 실행되었고, 123 번의 배포가 진행되었다. 자동화된 CI/CD 파이프라인을 통해 개발동안 인프라를 항상 최신 상태의 코드로 유지할 수 있었고, 배포 작업을 생략할 수 있으므로 코드 작성에 더 집중할 수 있는 환경을 구성할 수 있었다.

### 3.3.6. 테스트를 위한 기기 시뮬레이터 구현

대규모 하드웨어 구축의 현실적인 제약으로 인해, 수많은 기기를 대상으로 한 배포 테스트를 수행하기 위해 기기 시뮬레이터가 필요했다. 이에 Python 기반의 기기 시뮬레이터를 개발했으며, 이는 실제 기기와 동일하게 MQTT 메시지를 전송하고 파일을 다운로드하는 기능을 수행한다. 시뮬레이터의 전체 아키텍처는 다음과 같다.



시뮬레이터는 실제 기기의 복잡한 동작과 네트워크 상호작용을 정밀하게 재현하기 위해 다음과 같은 핵심 기능들을 구현했다.

- MQTT 통신: 각 시뮬레이터는 고유 ID 로 MQTT 브로커와 연결한다. 이를 통해 OTA 업데이트와 같은 명령을 비동기적으로 수신하고, 자신의 상태(온라인, 다운로드 중 등) 업데이트를 주기적으로 서버에 전송하는 양방향 통신을 수행한다.
- 보안 펌웨어 다운로드: 서버로부터 전달받은 일정 시간만 유효한 Presigned URL 을 통해 HTTPS 프로토콜로 펌웨어 파일을 안전하게 다운로드한다. 이는 인증되지 않은

---

접근으로부터 다운로드 링크를 보호하는 역할을 한다.

- 비동기 다운로드: 메인 통신 프로세스를 차단하지 않고 별도의 스레드에서 다운로드를 처리하여 동시성을 확보했다. 이를 통해 대용량 파일 다운로드가 다른 MQTT 메시지 처리나 상태 보고 작업을 막지 않도록 구현했다.
- 진행 상황 보고: 다운로드가 진행되는 동안 진행률(%), 속도, 예상 완료 시간 등을 주기적으로 MQTT 토픽에 게시(Publish)한다. 이는 서버가 수많은 기기의 개별 다운로드 현황을 실시간으로 추적할 수 있도록 지원한다.
- 무결성 검증: 다운로드 완료 후 SHA256 체크섬을 계산하여 원본 파일과 비교한다. 이를 통해 네트워크 전송 중 발생할 수 있는 파일 손상을 감지하고 데이터의 신뢰성을 보장했다.
- 최종 상태 보고: 다운로드의 최종 결과(성공, 실패, 시간 초과 등)를 서버에 다시 보고한다. 이 정보는 전체 배포 작업의 성공/실패 여부를 집계하는 데 사용된다.
- 고급 설정 (테스트 유연성): 디바이스 ID 나 MQTT 브로커 주소 같은 기본 정보뿐만 아니라, 다운로드 청크(Chunk) 크기 및 청크 간 지연 시간(Sleep)까지 환경 변수로 주입할 수 있도록 설계했다. 이를 통해 정상적인 네트워크 환경뿐만 아니라, 저속 네트워크나 불안정한 연결 상태를 시뮬레이션하여 시스템의 엣지 케이스를 검증할 수 있었다.

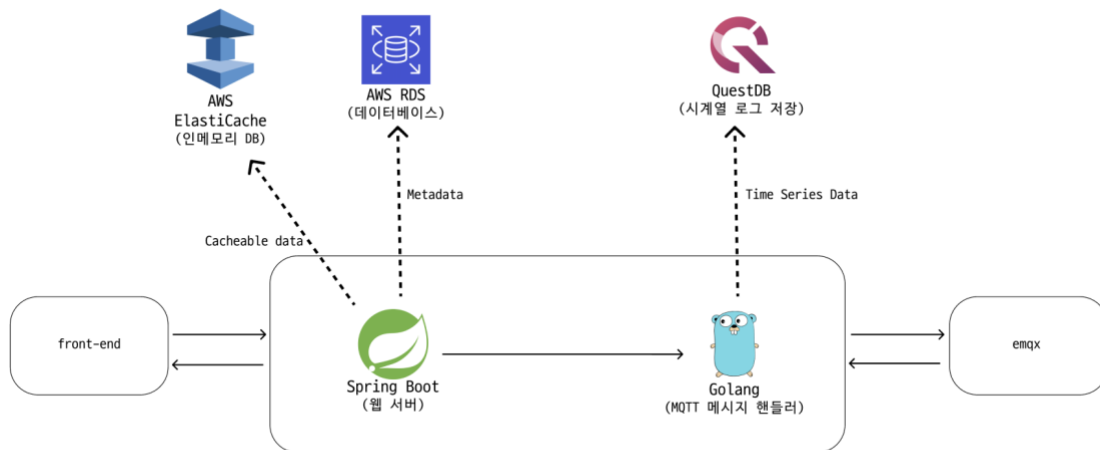
결론적으로, 이 시뮬레이터는 환경 변수를 통해 특정 기기 정보와 네트워크 조건을 설정하면, 배포 요청 수신, 비동기 다운로드 및 진행 상황 보고, 기기 메트릭 전송 등 실제 기기와 동일한 라이프사이클을 충실히 수행한다. 이를 통해 전 세계에 분산된 대규모 IoT 기기에 대한 배포를 효율적으로 테스트하고 검증할 수 있었으며, 상세한 결과는 “4. 연구 결과 분석 및 평가”에서 다루었다.



### 3.4. 백엔드

#### 3.4.1. 아키텍처

백엔드 아키텍처의 전체적인 구성은 다음과 같다.



본 프로젝트의 백엔드 아키텍처는 Spring Boot 기반의 REST API 서버와 Go 기반 MQTT 핸들러로 구성되었으며, AWS 클라우드 환경에서 동작하는 마이크로서비스 구조로 설계되었다.

Spring Boot 서버는 관리자 페이지와 연동되어 펌웨어, 콘텐츠 업로드 및 관리, 배포 요청, 기기 등록/인증 등의 기능을 제공한다.

한편, 수천 대의 기기와 동시에 통신해야 하는 특성상, 대규모 트래픽을 효율적으로 처리할 수 있는 별도의 서버가 필요했다. 일반적인 API 서버에서 기기와의 실시간 메시지까지 모두 담당할 경우, 관리자 기능과 기기 통신 간의 부하가 한곳에 집중되어 성능 저하와 장애 위험이 발생할 수 있다. 이를 해결하기 위해 MQTT 핸들러를 독립적으로 분리하였으며, 경량 메시지 프로토콜인 MQTT를 사용해 기기의 다운로드 진행률, 상태 로그, 에러 로그를 실시간으로 수집하도록 설계하였다. 해당 모듈은 Go 언어로 구현되었는데, Go는 고루틴(Goroutine)을 이용한 경량 스레드 기반 동시성 지원을 제공한다. 이는 전통적인 운영체제 스레드보다 훨씬 적은 자원으로, 수천 개 이상의 작업을 동시에 실행할 수 있어, 대규모 IoT 기기 환경에서 안정적이고 효율적인 메시지 처리가 가능하다.

이러한 구조를 통해 관리자 API 서버와 MQTT 핸들러가 각각의 역할에 집중할 수 있었으며,

---

결과적으로 Spring Boot 는 데이터 관리와 API 제공에, Go 는 대규모 기기와 실시간 통신에 최적화된 역할을 수행하도록 설계되었다.

### 3.4.2. 관리자 API 서버

관리자 API 서버는 펌웨어, 콘텐츠, 기기 및 그룹, 리전 관리와 배포를 총괄한다. RESTful API 설계를 기반으로 구축되었으며, AWS S3 및 CloudFront 를 활용한 파일 관리, MySQL 및 QuestDB 를 이용한 데이터 저장 및 분석, Redis 를 통한 실시간 배포 상태 관리 기능을 제공한다.

아래는 주요 기능 영역을 나누어 정리한 내용이다.

#### 3.4.2.1. 펌웨어 관리

펌웨어 관리 기능은 본 프로젝트의 핵심적인 기능 중 하나로, 단순히 파일을 저장하는 수준을 넘어, 업로드 → 메타데이터 등록 → 배포 요청 → 실시간 모니터링 및 이력 관리라는 일련의 과정을 체계적으로 제공한다.

#### 펌웨어 업로드

관리자는 새로운 펌웨어를 업로드하기 위해 서버에 파일 버전과 이름을 전달하면, 서버는 이를 기반으로 AWS S3 Presigned URL 을 발급한다. 발급된 URL 은 일정 시간 동안만 유효하며, 관리자는 이를 통해 인증 과정 없이 직접 S3 버킷에 업로드할 수 있다.

```
public UploadPresignedUrlResponseDto getPresignedUploadUrl(String version, String fileName) {
    // 동일한 버전.파일명이 이미 존재하는지 확인 (중복 방지)
    if (firmwareMetadataJpaRepository.findByVersionAndFileName(version, fileName).isPresent()) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "해당 펌웨어가 이미 존재합니다.");
    }
    // 랜덤 경로(UUID) 생성 후 Presigned URL 발급
    String path = UUID.randomUUID().toString();
    GeneratePresignedUrlRequest request = generatePresignedUploadUrl(bucketName, path);
    String url = amazonS3.generatePresignedUrl(request).toString();
}
```

```

        return new UploadPresignedUrlResponseDto(url, path);
    }

    private GeneratePresignedUrlRequest generatePresignedUploadUrl(String bucket, String path) {
        return new GeneratePresignedUrlRequest(bucket, path)
            .withMethod(HttpMethod.PUT) // 업로드 전용 URL
            .withExpiration(getPresignedUrlExpiration()); // 제한된 유효 시간(예: 5 분)
    }

```

### 펌웨어 메타데이터 등록 및 무결성 검증

업로드가 완료되면 관리자는 해당 파일의 메타데이터를 등록한다. 이 과정에서는 단순히 파일 이름과 버전만 저장하는 것이 아니라, SHA-256 해시값과 파일 크기를 함께 기록한다. 해시 알고리즘으로 SHA-256 을 선택한 이유는, 충돌 가능성이 극히 낮고 보안성이 검증되어 IoT 배포 환경에서도 안전하게 파일 무결성을 보장할 수 있기 때문이다. 추후 기기에서 다운로드된 파일과 원본 파일을 비교하여 손상 여부를 확인할 수 있다.

```

// SHA-256 해시 계산기 초기화
MessageDigest digest = MessageDigest.getInstance("SHA-256");

// 파일 내용을 읽어올 버퍼 생성
byte[] buffer = new byte[8192];
int bytesRead;

// 파일을 바이트 단위로 읽으면서 해시에 반영
while ((bytesRead = inputStream.read(buffer)) != -1) {
    digest.update(buffer, 0, bytesRead);
}

// 최종 해시값 계산
byte[] hashBytes = digest.digest();

// 바이트 배열을 16 진수 문자열로 변환
String fileHash = bytesToHex(hashBytes);

```

---

이 코드는 업로드된 파일을 바이트 단위로 읽어 SHA-256 알고리즘을 적용한 뒤, 최종적으로 해시 문자열을 생성한다. 해당 해시값과 파일 크기는 메타데이터와 함께 MySQL 데이터베이스에 저장되며, 주요 컬럼에는 version, fileName, releaseNote, path, fileHash, fileSize 가 포함된다.

### **펌웨어 배포 요청**

펌웨어 배포는 관리자가 특정 기기, 그룹, 혹은 리전을 지정하여 수행한다. 관리자는 배포 유형(DEVICE, GROUP, REGION)을 지정해 API 를 호출하면, 서버는 배포 명령을 생성하고 Redis 에 해당 정보를 기록한다. Redis 로 배포 만료 시점을 제어하며, 각 기기의 다운로드 상태를 실시간으로 반영한다.

### **실시간 모니터링 및 상태 관리**

배포 과정에서 발생하는 이벤트(예: 다운로드 진행률, 완료 여부, 오류 로그)는 Go 기반의 MQTT 핸들러를 통해 수집된다. 이 데이터는 QuestDB 에 저장되어 시계열 데이터 형태로 관리된다. 이를 통해 관리자는 특정 기기의 배포 속도, 네트워크 장애 발생 여부, 전체 배포 성공률 등을 실시간으로 파악할 수 있으며, Redis 와 QuestDB 의 결합으로 실시간성과 이력 관리 기능을 동시에 확보할 수 있다.

### **배포 이력 및 상세 조회**

마지막으로, 관리자는 과거의 배포 내역을 조회하고 상세한 로그를 확인할 수 있다. 이 기능은 단순한 성공/실패 여부를 넘어, 각 기기별 상태(성공, 실패, 대기), 배포 시각, 진행률, 오류 메시지 등을 종합적으로 제공한다.

#### **3.4.2.2. 콘텐츠 관리**

콘텐츠 관리 기능은 펌웨어 관리와 유사한 구조를 따르지만, 콘텐츠 특성상 원본 파일과 기기에서 재생 가능한 바이너리 파일을 함께 다룬다는 점에서 차별화된다.

---

## 콘텐츠 업로드

관리자는 새로운 콘텐츠를 등록하기 위해 먼저 콘텐츠 제목을 입력하고 요청을 보낸다. 서버는 이 요청을 바탕으로 두 개의 Presigned URL 을 반환한다.

- 첫 번째 URL 은 콘텐츠의 원본 파일을 업로드하는 용도이다.
- 두 번째 URL 은 기기에서 재생 가능한 바이너리 파일을 업로드하는 용도이다.

이처럼 두 개의 파일을 동시에 관리하는 이유는, 관리자가 원본 파일을 보존하면서도 IoT 기기에서 요구하는 경량화된 파일을 분리하여 제공할 수 있도록 하기 위함이다. 특히 다양한 해상도나 디바이스 성능을 고려해야 하는 콘텐츠 배포의 특성상, 원본과 변환 파일을 구분하는 구조는 매우 중요하다.

## 콘텐츠 메타데이터 등록 및 무결성 관리

업로드가 완료되면 관리자는 콘텐츠의 제목, 원본 파일 경로, 바이너리 파일 경로를 등록한다. 이 과정에서는 중복 등록 방지를 위해 동일한 제목을 가진 콘텐츠가 이미 존재하는지 확인한다.

또한 콘텐츠 파일 역시 펌웨어와 동일하게 파일 해시 검증을 거친다. 이를 통해 콘텐츠 파일이 손상되지 않았음을 보장하며, 추후 기기가 다운로드한 콘텐츠 파일과 비교해 무결성 확인을 수행할 수 있다.

등록된 콘텐츠 메타데이터는 MySQL 데이터베이스에 저장된다. 주요 항목으로는 title, description, originalS3Path, binaryS3Path 가 있으며, 관리자는 이를 기반으로 콘텐츠 목록을 조회하고 검색할 수 있다.

## 콘텐츠 배포 요청

콘텐츠 배포와 펌웨어 배포의 가장 큰 차이는, 콘텐츠는 여러 개의 파일을 동시에 배포할 수 있다는 점이다. 특정 기기에 두 개의 새로운 콘텐츠를 동시에 배포할 경우, 기기는 배포 명령에 포함된 두 개의 Signed URL 을 받아 순차적으로 다운로드 한다. 이를 통해 관리자는 다양한 콘텐츠를 동시에 제공할 수 있다.

특히 콘텐츠 관리에서는 기기별로 활성화된 콘텐츠 리스트를 확인하는 기능도 제공된다. 관리자는 특정 매장에서 현재 어떤 콘텐츠가 노출되고 있는지 실시간으로 모니터링 할 수

---

있으며, 새로운 콘텐츠 배포 시 기존 콘텐츠가 정상적으로 종료되었는지도 검증할 수 있다.

### 3.4.2.3. 기기·그룹·리전 관리

관리자 API 서버는 단순히 펌웨어와 콘텐츠 파일을 다루는 데 그치지 않고, 이를 배포할 대상인 기기(DEVICE), 그룹(GROUP), 리전(REGION)을 관리한다.

- 기기 관리: 각 기기의 등록, 인증, 상세 조회, 활성화 여부 확인 기능 제공
- 그룹 관리: 매장/조직 단위의 그룹 생성 및 디바이스 매핑 관리
- 리전 관리: 대규모 단위(예: 지역)로 디바이스 묶음 관리

관리자는 단일 기기부터 전국 단위 리전까지 다양한 스케일에서 배포를 제어할 수 있다. 예를 들어, 특정 리전 전체에 긴급 보안 펌웨어를 배포하거나, 특정 그룹에만 프로모션 콘텐츠를 배포하는 것이 가능하다.

### 3.4.2.4. 배포 상태 관리 및 실시간 모니터링

본 시스템은 대규모 IoT 기기 환경에서 안정적인 배포와 실시간 모니터링을 위해 Spring Scheduler, Redis, QuestDB, MySQL 을 통합하여 설계하였다. Spring Scheduler 는 배포 단위별로 독립적인 스케줄러를 실행해 진행 상황을 점검하며, Redis 는 배포 대상 기기를 Set 구조로 관리하여 완료 여부를 빠르게 집계한다. QuestDB 는 각 기기의 다운로드 이벤트를 시계열 데이터로 저장해 최신 상태를 판별하고, MySQL 은 최종 결과와 이력을 영구 보관한다. 이를 통해 관리자는 실시간 모니터링, 실패 기기 추적을 수행할 수 있다.

해당 로직은 다음과 같다.

1. 관리자가 펌웨어 또는 콘텐츠 배포를 요청하면, 대상 기기 ID 들이 Redis Set 에 저장된다.
2. Spring Scheduler 가 배포 단위별로 주기적으로 동작하며, Redis 에 남아있는 기기들을 확인한다.
3. Redis 에 남아 있는 각 기기 ID 에 대해 QuestDB 에서 최신 다운로드 이벤트를 조회한다.
4. 상태가 SUCCESS, FAILED, TIMEOUT, ERROR 중 하나라면 완료 처리 대상이 된다.
5. 완료된 기기는 MySQL 에 기록하고 Redis Set 에서 제거한다.
6. Redis Set 이 비어있으면 배포 완료(COMPLETED)로 판정하고 스케줄러를 종료한다.
7. 배포 만료 시간(expiresAt)을 초과했는데 Redis 에 기기가 남아 있으면, 남은 기기를

---

모두 TIMEOUT 처리하고 스케줄러를 종료한다.

### 3.4.3. MQTT 핸들러

MQTT 핸들러는 IoT 기기와 백엔드 서버간의 실시간 메시지 중계 및 이벤트 수집을 담당한다. Go 기반으로 구현되었으며, 경량 스레드(goroutine)과 채널(channel)을 활용하여 수천 대의 기기를 동시에 안정적으로 처리할 수 있도록 설계 되었다.

핸들러는 MQTT 브로커와의 연결 관리, 펌웨어·콘텐츠 배포 명령 발행, 기기 이벤트 수집 및 저장소 연동 등 다양한 기능을 수행하며, 관리자 API 서버(Spring Boot)와 분리된 독립적인 모듈로 운영된다.

아래는 주요 기능 영역을 나누어 정리한 내용이다.

#### 3.4.3.1. MQTT 연결 및 구독 관리

핸들러는 Eclipse Paho MQTT 라이브러리를 이용해 브로커(EMQX)에 연결하며, 자동 재연결 및 세션 복구 기능을 활성화하여 장시간 안정적인 통신을 보장한다.

- 재연결 처리: 연결이 끊어지면 자동으로 재시도하며, OnConnect 콜백을 통해 모든 토픽을 다시 구독한다.
- 구독 토픽:
  - v1/+/update/request/ack: 배포 요청 수신 확인
  - v1/+/update/progress: 다운로드 진행률 보고
  - v1/+/update/result: 다운로드 결과 보고
  - v1/+/update/cancel/ack: 취소 요청 수신 확인
  - v1/+/status/system: 기기 시스템 상태 보고
  - v1/+/status/error\_log: 기기 에러 로그 전송
  - v1/+/sales/data: 판매 데이터 수집
  - v1/+/regist: 기기 등록 요청

이처럼 토픽 구독 구조를 세분화함으로써, 펌웨어/콘텐츠 배포, 상태 모니터링, 로그 수집 등 다양한 이벤트를 독립적으로 처리할 수 있다.

### 3.4.3.2. 펌웨어 및 콘텐츠 배포 요청 (Publish)

핸들러는 관리자가 생성한 배포 요청을 수신하여 MQTT 메시지로 변환, 각 기기로 전달한다. 이 과정에서 고루틴(goroutine)과 채널(channel)을 활용해 수천 대 기기에도 병렬로 빠르게 명령을 전달할 수 있다.

```
func (m *MQTTClient) PublishDownloadRequest(req *types.FirmwareDeployRequest) {
    go func() {
        for _, deviceInfo := range req.Devices {
            topic := fmt.Sprintf("v1/%d/update/request/firmware", deviceInfo.DeviceId)
            m.mqttClient.Publish(topic, 1, false, payload)
            // 초기 상태 이벤트 비동기 기록
            repository.DownloadInsertChan <- types.DownloadEvent{
                CommandID: command.CommandID,
                DeviceID: deviceInfo.DeviceId,
                Status: "WAITING",
            }
            time.Sleep(10 * time.Millisecond) // 과도한 트래픽 방지
        }
    }()
}
```

본 구조는 배포 요청을 독립된 경량 스레드(goroutine)에서 실행해 메인 프로세스를 블로킹하지 않으며, 동시에 다수 기기에 빠르게 명령을 전달할 수 있다. 또한 이벤트를 비동기 채널에 기록하여 발행과 모니터링이 병렬로 진행되어, 퍼블리시 사이에 짧은 지연(10ms)을 두어 과도한 네트워크 부하를 방지하였다.

이를 통해 대규모 IoT 환경에서도 빠른 배포 속도와 안정성을 동시에 확보할 수 있었다.

### 3.4.3.3. MQTT Subscribe 기반 이벤트 수집

본 시스템의 MQTT 핸들러는 IoT 기기에서 발생하는 다양한 이벤트를 실시간으로 수집하기 위해 여러 토픽을 구독한다. 모든 이벤트는 메시지 수신 → JSON 파싱 → 비동기 채널 전달 → QuestDB 저장이라는 동일한 흐름을 따르며, 이벤트 유형만 다를 뿐 구조는 일관되게 유지된다.



---

아래는 펌웨어 다운로드 이벤트 구독의 예시 코드이다. 메시지를 수신하면 JSON 으로 파싱하여 비동기 채널로 전달하여 QuestDB 에 저장한다.

```
func (m *MQTTClient) subscribe(
    topic string,
    parse func(mqtt.Message) (*types.DownloadEvent, error),
    label string) {

    handler := func(client mqtt.Client, msg mqtt.Message) {
        event, err := parse(msg)
        if err != nil {
            log.Printf("[ERROR] %s 파싱 실패: %v", label, err)
            return
        }
        // 비동기 채널로 전달 → QuestDB 저장
        repository.DownloadInsertChan <- *event
    }
    m.mqttClient.Subscribe(topic, 1, handler)
}
```

시스템 상태, 에러 로그 등의 이벤트도 이와 동일한 방식으로 처리되며, 각각 전용 채널을 통해 QuestDB 에 기록된다. 디바이스 등록 이벤트의 경우에는 Spring Boot 서버로 HTTP 요청을 전달하여 기기 등록 여부를 검증한 뒤, 결과를 다시 MQTT ACK 메시지로 발행하는 예외적 흐름을 가진다.

이러한 이벤트 수집 구조를 적용함으로써 시스템은 일관성, 확장성, 안정성, 실시간성이라는 네가지 핵심 효과를 얻을 수 있었다.

#### 3.4.3.4. 이벤트 저장 및 Consumer 구조

이벤트 Consumer 는 IoT 기기에서 발생한 다운로드 진행률, 시스템 상태, 에러 로그, 판매 데이터 등을 비동기 채널을 통해 전달받아 QuestDB 에 저장하는 모듈이다.

이 구조는 이벤트 수집과 저장을 분리하여 처리 병목을 방지하고, 1 초 단위의 Flush 로 안정성을 확보하였다.

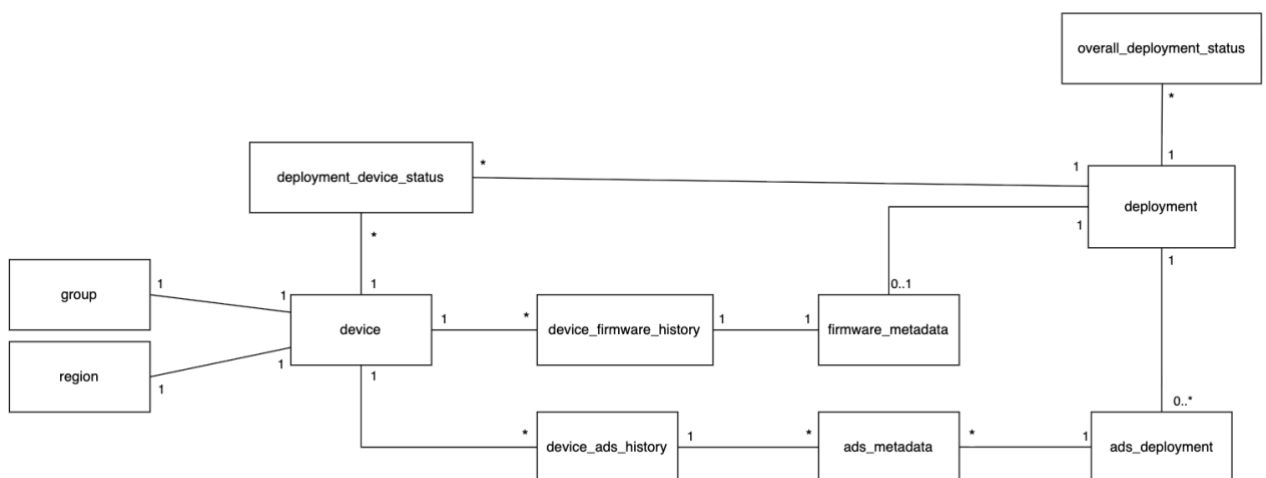
// 다운로드 이벤트 Consumer 예시

```
func (c *DBClient) StartEventConsumer() {
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()

    for {
        select {
        case event := <-DownloadInsertChan:
            ctx := context.TODO()
            c.sender.Table("download_events").
                Symbol("command_id", event.CommandID).
                Int64Column("device_id", event.DeviceID).
                Int64Column("progress", event.Progress).
                At(ctx, event.Timestamp.UTC())
        case <-ticker.C:
            c.sender.Flush(context.TODO()) // 주기적 flush
        }
    }
}
```

결론적으로 수천 대 기기에서 발생하는 대규모 이벤트를 안정적으로 수집·저장할 수 있었다.

#### 3.4.4. 데이터베이스 설계



본 시스템에서는 기기(Device) 단위의 배포 및 이력 관리가 중요한 핵심 기능으로 설계되었다. 각 기기는 하나의 그룹(Group)과 하나의 리전(Region)에 소속되며, 관리자는 해당 기기에 펌웨어 또는 콘텐츠를 배포할 수 있다.

배포 진행률과 개별 상태는 실시간 처리를 위해 QuestDB 에 시계열(Time-series) 데이터로 저장된다. 다만, QuestDB 의 시계열 데이터는 영구적인 저장을 목적으로 하지 않기 때문에, MySQL 에 배포 상태 및 이력을 별도로 기록하여 장기적인 추적과 분석이 가능하도록 하였다.

전체 엔티티에 대한 세부 내용은 다음과 같다.

엔티티	설명
group	기기들을 묶어 관리하기 위한 그룹 정보
region	기기가 속한 지리적 지역 정보
device	실제 배포 대상이 되는 개별 기기 정보
ads_metadata	광고 콘텐츠 파일 및 속성에 대한 메타데이터
firmware_metadata	펌웨어 파일 및 버전에 대한 메타데이터
deployment_device_status	특정 배포에 대해 기기별 상태를 기록하는 테이블
deployment	펌웨어나 광고 콘텐츠 등의 배포 요청 및 실행 단위
ads_deployment	하나의 배포와 여러 광고 콘텐츠 간의 매핑된 테이블
overall_deployment_status	배포 전체의 진행 상태
device_firmware_history	특정 기기에 어떤 펌웨어가 언제 배포 및 적용되었는지 기록
device_ads_history	특정 기기에 어떤 광고 콘텐츠가 언제 배포 및 적용되었는지 기록

---

### 3.5. 임베디드 시스템

#### 3.5.1. 시스템 개요 및 하드웨어 구조

임베디드 시스템 파트는 본 연구에서 구축한 클라우드 기반 OTA 관리 기술을 실제 저자원 MCU 기반 하드웨어에 적용하여 기술적 실증을 수행하는 핵심 단계이다. 최초 목표는 범용적으로 적용 가능한 네트워크 모듈을 개발하여 기존 MCU 기반 임베디드 시스템에 확장성을 제공하는 것이었으나, 대규모 펌웨어 및 콘텐츠 전송 과정에서 발생하는 물리적 제약으로 인해 졸업과제 범위 내에서는 현실적으로 어려움이 있었다. 이에 따라 본 연구에서는 네트워크 모듈을 포함하는 임베디드 보드를 선정하고, 이를 기반으로 다양한 환경에 적용 가능한 시스템 구조와 기능 정의를 제안하는 방향으로 전환하였다.

저수준 하드웨어 API에 상당히 종속적인 베어메탈 MCU의 특성상, 본 연구에서 개발한 구조를 산업 현장의 모든 장치에 직접적으로 적용하기에는 제약이 존재한다. 그러나 MCU 기반 임베디드 하드웨어 전반에서 공통적으로 활용 가능한 아키텍처와 기능 정의를 제시함으로써, 유사한 로직을 바탕으로 다양한 산업용 시스템에 적용할 수 있음을 실증하였다. 이를 위해 교육계와 산업계 모두에서 가성비와 활용성 면에서 높은 평가를 받고 있는 ESP32-S3 플랫폼을 채택하였으며, LCD 터치 디스플레이가 포함된 ESP32 기반 HMI 보드를 사용하여 시스템을 구축하였다. 특히 듀얼 코어 구조와 FreeRTOS의 태스크 기반 병렬 처리 방식을 활용하여 UI 태스크와 네트워크 태스크를 분리 및 병행할 수 있도록 설계하였다. 이를 통해 OTA 업데이트, GUI 기반 사용자 입력 처리, 콘텐츠 갱신 기능까지 통합적으로 검증할 수 있는 시스템을 구현하였다.

하드웨어 구현에는 ESP32-S3 MCU를 기반으로 하는 Elecrow사의 CrowPanel 7.0 HMI 보드를 활용하였다. 해당 보드는 800×480 해상도의 7.0인치 TFT-LCD 터치 디스플레이와 ESP32-S3 칩셋이 단일 보드에 집적된 형태로, GUI 표시와 네트워크 통신을 동시에 수행할 수 있다. HMI 보드에는 LVGL 기반 GUI 프로그램을 탑재하여 사용자가 직관적으로 시스템을 제어할 수 있도록 하였으며, 외부 장치 제어를 위해 Arduino Uno를 연결하여 UART 통신을 통한 서보모터 제어를 수행하였다. 이를 통해 "디지털 캔디 머신"을 모사하는 목업 제품을 제작하고, OTA 업데이트 및 콘텐츠 배포 결과를 물리적으로 확인할 수 있는 시연 환경을 구성하였다.

---

### 3.5.2. 임베디드 펌웨어 아키텍처

#### 3.5.2.1. 기술적 구조 개요

본 연구의 임베디드 펌웨어는 저자원 MCU 환경에서 널리 활용되는 경량 RTOS 인 FreeRTOS 를 기반으로 하여, 멀티태스킹 구조를 바탕으로 설계되었다. ESP32-S3 의 듀얼 코어 환경을 적극 활용하여 코어별로 기능을 분산 배치함으로써 사용자 인터페이스와 네트워크 통신이 원활하게 병행될 수 있도록 하였다. 이러한 멀티태스킹 구조는 GUI 인터페이스를 통한 사용자 입력 처리와 OTA 업데이트, 콘텐츠 관리와 같은 다양한 기능이 동시에 실행되면서도 끊김 없는 응답성을 보장하기 위해 필수적인 요소다.

```
bool init_ui_task(void) {  
    Serial.println("[coffee/ui_task][info] initializing UI...");  
    ui_init();  
    if (xTaskCreatePinnedToCore(ui_task, "ui", 8192, nullptr, tskIDLE_PRIORITY + 2,  
        nullptr, 1) != pdPASS) {  
        Serial.println("[coffee/ui_task][error] failed to create ui_task");  
        return false;  
    }  
    return true;  
}  
  
if (xTaskCreatePinnedToCore(init_wifi_task, "init_wifi", 4096, task_args, tskIDLE_PRIORITY  
    + 6, nullptr, 0) != pdPASS) {  
    Serial.println("[coffee/network_task.cpp][error] failed to create init_wifi task");  
    delete task_args;  
}
```

FreeRTOS 에서는 태스크를 생성할 때 xTaskCreatePinnedToCore() API 를 통해 특정 코어에 고정(Pinning)시킬 수 있다. 만약 태스크를 특정 코어에 고정시키지 않을 경우에는

---

Unpinned 상태로 Ready 큐에 삽입되며, 우선 순위에 따라 스케줄러가 적절한 코어를 배정한다[1].

ESP32-S3의 두 코어는 성능적으로 동일하지만, ESP 플랫폼의 Wi-Fi / Bluetooth 스택 및 esp\_timer 등 일부 시스템 태스크는 Core 0에 고정된다. 따라서 GUI 태스크같이 지속적으로 실행되는 고부하 태스크는 일반적으로 Core 1에 배치하는 것이 권장된다.

또한 동일한 코어에 태스크를 여러 개 고정시키는 경우에는 태스크의 우선순위를 적절하게 설정하는 것이 중요하다. FreeRTOS는 프로세스의 기아 상태를 방지하는 메커니즘이 포함되어 있지 않으므로[2], 같이 장시간 실행되어야 하는 UI 태스크와 같은 작업에 높은 우선 순위를 부여하는 것은 피해야 한다.

### 3.5.2.2. 전반적 설계 특징

네트워크 처리와 UI 처리를 루프 함수 기반의 베어메탈 MCU 구조에서 동시에 수행하는 것은 구조적으로 어려움이 크다. 이를 극복하기 위해 본 연구의 펌웨어는 FreeRTOS를 활용하여 태스크를 분산 배치함으로써 실시간성을 보장하였다. 또한 안정적인 시스템 운용을 위해 각 태스크는 독립적으로 실행되면서도 큐(Queue), 뮉텍스(Mutex / Semaphore) 등 FreeRTOS의 IPC(Inter-Process Communication) 기능을 통해 동기화 및 이벤트 제어를 수행하는 것이 특징이다.

### 3.5.2.3. 주요 태스크 소개

임베디드 펌웨어에서 실행하는 태스크는 다음과 같다.

#### - UI 태스크

```
static void ui_task(void* task_param) {  
    char buf[COFFEE_MAX_STR_BUF] = { 0 };  
    while (true) {  
        if (dbg_overlay_label && queue_poll(dbg_overlay_q, buf)) {
```

```

        lv_label_set_text(dbg_overlay_label, buf);
    }

    lv_timer_handler();

    vTaskDelay(pdMS_TO_TICKS(10));
}

vTaskDelete(NULL);
}

```

LVGL 기반 GUI를 구동하며, 장치 실행 동안 상시 동작한다. UI 자원을 안정적으로 관리하기 위해 Core 1에 고정하여 실행한다. UI의 원활한 운영을 위하여 그 외 다른 작업들은 모두 코어 0에서 실행시킨다.

#### - 네트워크 관련 태스크

Wi-Fi 및 MQTT 클라이언트 초기화, MQTT 이벤트 수신 대기 및 파일 다운로드 등을 처리한다. 이러한 작업은 대부분 고부하 작업이지만 발생하는 빈도가 낮아 모두 코어 0에서 처리되어도 큰 문제가 없다.

UI 태스크 외에 지속적인 작업을 수행해야 하는 태스크들도 존재하는데, 구체적으로 다음과 같은 태스크들이 있다.

#### - 다운로드 태스크

다운로드 태스크는 큐(download\_q)에서 요청을 수신하여 파일 다운로드를 수행한다. 초기에는 서버의 요청마다 별도의 태스크를 생성했으나, 메모리 부족과 스택 오버플로우 문제가 발생하여 하나의 다운로드 태스크가 배치 방식으로 순차 처리하도록 개선하였다. 이를 통해 성능 일부를 희생하는 대신 안정적인 운용을 확보하였다.

#### - 시스템 상태 보고 태스크

주기적으로 시스템의 상태를 전송하는 상태 보고 태스크 또한 주요 작업을 처리하는 태스크

---

중 하나다. 이 태스크는 1 분 간격으로 시스템 상태를 MQTT 를 통해 서버에 송신한다. vTaskDelay 를 활용하여 Blocked 상태로 대기[2]하다가 주기적으로 실행됨으로써, 다른 태스크 수행에 지장을 주지 않도록 구현하였다.

#### 3.5.2.4. 태스크 간 동기화 및 통신(IPC)

멀티코어 환경에서는 자원에 대한 동시 접근 문제가 필연적으로 발생한다. 특히 UI 태스크가 관리하는 LVGL 자원에 다른 코어에서 직접 접근하면 경쟁 상태(Race Condition)가 발생하여 데이터 무결성과 시스템 안정성이 훼손될 수 있다. 이를 방지하기 위해 UI 갱신과 관련된 모든 작업은 반드시 UI 태스크를 통해서만 처리할 수 있도록 설계하였다.

이를 위해 본 연구의 펌웨어는 FreeRTOS 큐를 활용하여, 코어 0 의 타 작업에서 실행된 결과를 큐에 전송하면 코어 1 의 UI 태스크가 수신하여 이를 UI 에 반영하는 형태로 구현하였다. 또한 빈번하게 발생하는 태스크 간 통신을 효율적으로 처리하기 위해 별도의 IPC 모듈을 설계하여 큐를 통합 관리하였다.

```
void queue_printf(QueueHandle_t queue, const std::string& tag, bool serial_print, const char* fmt, ...) {  
    // ...  
    char msg[COFFEE_MAX_STR_BUF] = { 0 };  
    strncpy(msg, res.c_str(), (res.size() < COFFEE_MAX_STR_BUF) ? res.size():  
COFFEE_MAX_STR_BUF - 1);  
    msg[COFFEE_MAX_STR_BUF - 1] = 0;  
    xQueueSend(queue, msg, portMAX_DELAY);  
}  
  
bool queue_poll(QueueHandle_t queue, char* msg_out) {  
    if (!queue) {  
        return false;  
    }  
}
```



```
return (xQueueReceive(queue, msg_out, 0) == pdPASS);  
}
```

SD 카드, 네트워크 객체, 공통 전역 변수 등 공유 자원 접근에서도 경쟁 상태를 방지해야 한다. 이 경우에는 FreeRTOS의 뮤텝스를 활용하여 자원을 보호하였다. 뮤텝스 역시 IPC와 유사하게 Event Control 모듈을 통해 통합 관리하도록 설계하였다.

```
bool lock_mtx(SemaphoreHandle_t& mtx, std::size_t wait_time) {  
    if (!mtx) {  
        return false;  
    }  
    if (wait_time == portMAX_DELAY) {  
        return (xSemaphoreTake(mtx, portMAX_DELAY) == pdTRUE);  
    } else {  
        return (xSemaphoreTake(mtx, pdMS_TO_TICKS(wait_time)) == pdTRUE);  
    }  
}  
  
void unlock_mtx(SemaphoreHandle_t& mtx) {  
    if (!mtx) {  
        return;  
    }  
    xSemaphoreGive(mtx);  
}
```

---

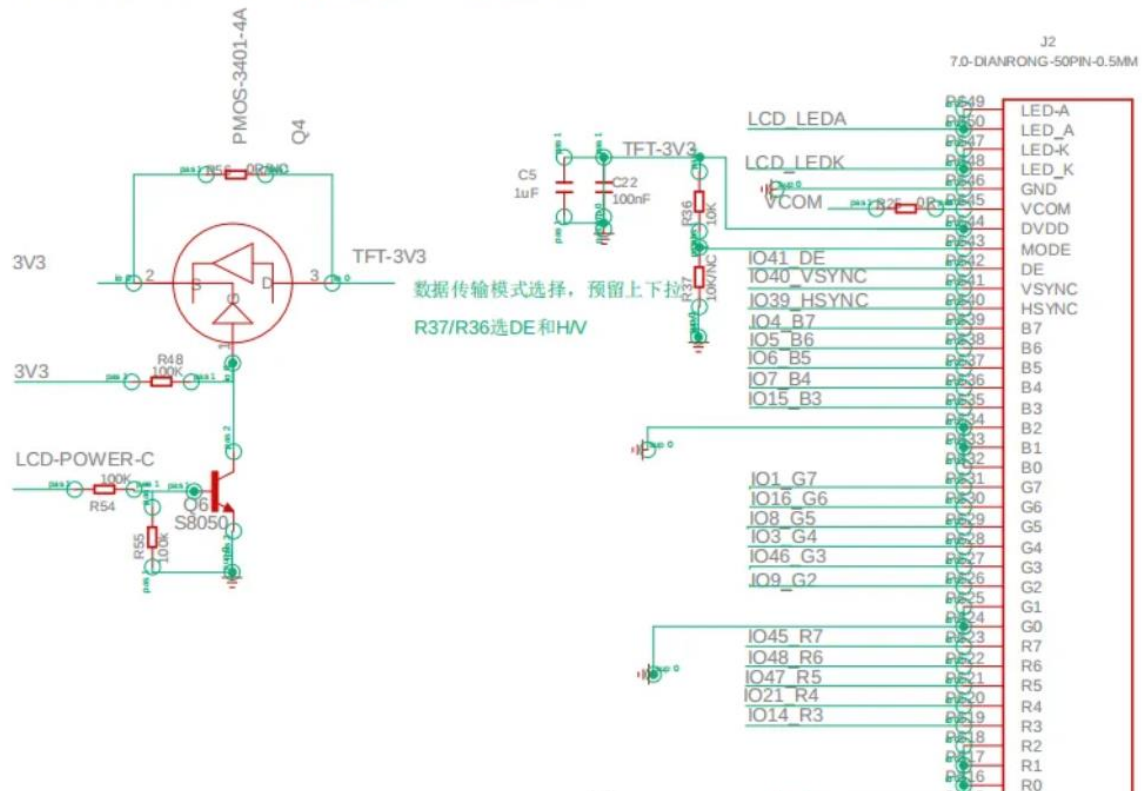
### 3.5.3. 하드웨어 드라이버 및 UI

#### 3.5.3.1. 드라이버 계층 설계

본 연구의 임베디드 하드웨어는 ESP32-S3 를 기반으로 하며, TFT-LCD 디스플레이와 결합된 형태의 보드를 활용한다. 이러한 하드웨어는 단순 MCU 동작을 넘어 디스플레이 출력과 터치 입력을 포함하므로, 이를 제어하기 위한 전용 드라이버가 필요하다. 본 연구에서는 디스플레이 드라이버와 터치 드라이버를 중심으로 구현하였으며, 개발 편의성과 확장성을 고려하여 Wi-Fi, SD 카드 접근, RTC 초기화 등 유틸리티 드라이버 또한 추가하였다. 드라이버 코드는 메인 프로젝트와 분리된 독립 저장소에서 관리하여 해당 보드를 활용하는 다른 개발자들이 활용할 수 있도록 공개하였다[3].

디스플레이 드라이버는 LVGL 과 LovyanGFX 를 연결하는 중간 계층으로 구현되었다. LVGL 은 GUI 프레임워크로서 버튼, 라벨 등 다양한 위젯을 제공하며, 이벤트 처리와 사용자 입력을 GUI 동작으로 변환한다. LovyanGFX 또한 TFT-LCD 디스플레이 상에 간단한 도형은 그릴 수 있는 그래픽 라이브러리이지만 본 연구의 하드웨어 드라이버에서는 LVGL 이 생성한 픽셀 데이터를 실제 하드웨어에 출력하는 역할을 수행하는 방식으로만 활용한다. LovyanGFX 를 활용하여 ESP32-S3 와 LCD 모듈 간 연결을 처리하였고, 제조사가 제공한 스키마틱에 따라 RGB565(16 비트) 컬러 포맷에 맞춰 핀 매핑을 직접 설정하였다[4].

## LCD+TP Interface



터치 드라이버는 GT911 칩을 기반으로 하며, 공개된 오픈소스 라이브러리[5]를 활용하여 구현하였다. 드라이버 초기화 과정에서 LVGL의 입력 장치 드라이버와 연결하여 터치 좌표를 GUI 이벤트로 전달한다. 다만 본 보드의 핀 수가 제한적이어서, ESP32-S3의 I2C 포트를 터치 드라이버와 외부 디바이스가 공유할 경우 간섭이 발생할 수 있다는 점을 고려해야 한다.

- 유틸리티 드라이버

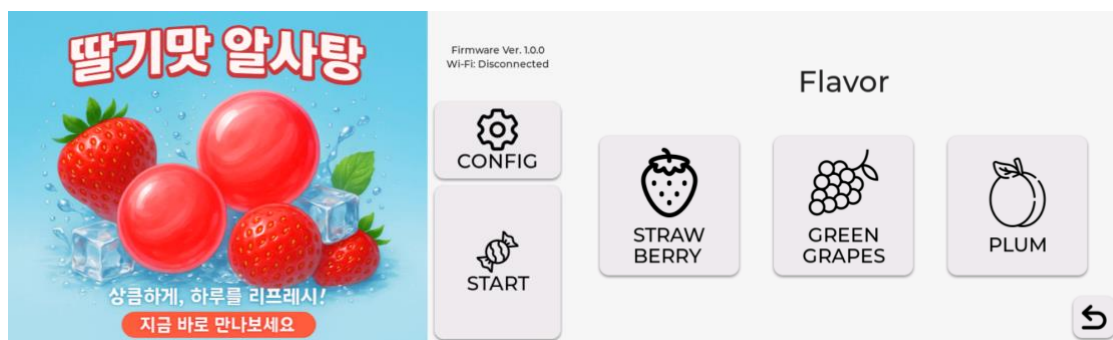
유틸리티 드라이버는 저수준 하드웨어 접근보다는 고수준 인터페이스 제공에 중점을 두었다. ESP-IDF[6] 환경에 Arduino as a component 방식을 적용하여 Arduino 스타일의 함수를 그대로 사용할 수 있도록 구성하였으며[7], 이를 통해 Wi-Fi 연결, SD 카드 파일 입출력, RTC 초기화를 손쉽게 사용할 수 있도록 하였다. 예를 들어, LVGL 이 이미지 리소스를 표시할 때는 펌웨어에 직접 포함하지 않고 SD 카드에서 동적으로 읽어오도록 하였으며, 이를 위해 LVGL 파일 시스템 인터페이스를 SD 드라이버와 연동하였다. RTC 초기화의 경우 mTLS 기반 보안 통신에 필요한 서버 시간 동기화를 위해 esp\_sntp API 를 사용하여 NTP 서버(pool.ntp.org)와 시간을 동기화하도록 설계하였다.

### 3.5.3.2. GUI 라이브러리 활용

GUI 는 LVGL 을 기반으로 구현하였다. LVGL 은 전체 화면을 매 프레임 다시 그리지 않고, 변경된 영역만 갱신하는 방식으로 동작한다. 이를 위해 LVGL 이 갱신한 내용을 담을 픽셀 버퍼가 필요하며, 본 연구에서는 초기에는 외부 RAM 을 활용하였다. 하지만 테스트를 진행하면서 눈에 띄는 성능 저하가 발생하는 것을 확인하였고, 이후 테스트와 조정 과정을 거쳐 내부 DMA 가능 메모리에 버퍼를 할당하는 방식으로 변경하였다. 내부 메모리 용량의 제약을 고려해 전체 화면 크기의 버퍼를 두는 대신, LVGL 이 제공하는 더블 버퍼링 기법을 활용하여 두 개의 작은 DMA 버퍼를 운용하였다.

화면 출력 과정은 LVGL 이 생성한 픽셀 버퍼를 LinyanGFX 가 DMA 를 통해 LCD 모듈로 전송하는 방식으로 수행된다. DMA 를 이용함으로써 CPU 는 그래픽 데이터 전송 과정에 묶이지 않고 다른 태스크를 처리할 수 있으며, DMA 전송 과정이 끝나기를 대기한 후 LVGL 에 다음 화면 내용을 보내라는 콜백을 반환함으로써 버퍼 경쟁 상태를 방지한다.

### 3.5.3.3. UI 구현 방향



본 연구의 UI 는 OTA 및 콘텐츠 업데이트 기능을 검증할 수 있도록 설계되었다. 광고 콘텐츠를 포함한 디지털 캔디 머신 목업 프로그램으로 구현하였으며, 간단한 기능과 직관적인 UI 를 제공하여 무인 판매기 환경에 익숙하지 않은 사용자도 쉽게 사용할 수 있도록 하였다.

### 3.5.4. 네트워크 및 보안 통신

임베디드 시스템의 네트워크 계층은 기기와 클라우드 서버 간의 안정적이고 안전한 데이터 교환을 보장하는 데에 노력을 기울였다. 본 연구에서는 Wi-Fi 기반의 인터넷 연결을 바탕으로

MQTT(Message Queuing Telemetry Transport) 프로토콜을 활용하여 클라우드 서버와 통신하며, 보안성과 신뢰성을 확보하기 위해 mTLS(Mutual TLS)를 적용하였다. 또한 펌웨어 및 콘텐츠의 무결성 검증을 통해 OTA 과정에서 발생할 수 있는 위협을 차단하고, 장기적인 운영을 고려한 인증서 갱신 기능을 포함하였다.

### 3.5.4.1. Wi-Fi 및 MQTT 초기화

시스템이 부팅되면 우선적으로 Wi-Fi 초기화를 수행하며, 동시에 RTC 동기화를 진행한다. RTC 동기화는 mTLS 연결에서 필수적인 인증서 유효성 검증에 필요하므로 초기 단계에서 수행된다. Wi-Fi 연결이 성공하면 기기는 사전에 저장된 인증서를 기반으로 클라우드 서버와 mTLS MQTT 연결을 수행한다.

MQTT 연결이 확립되면, 기기는 사전에 정의된 주제(Topic)를 구독한다. OTA 요청, 콘텐츠 배포 요청, 인증서 갱신 요청 등 모든 관리 메시지는 이 토픽을 통해 전달된다. MQTT 이벤트 처리를 위한 태스크는 별도로 분리하여 설계함으로써 이벤트 수신 시 관련 로직이 독립적으로 동작하여 다른 기능과의 간섭을 최소화하였다.

업데이트 요청 메시지를 수신하면, 기기는 메시지에 포함된 Signed URL 을 사용하여 지정된 파일을 다운로드한다. 다운로드 과정은 청크 단위로 수행되며, 수신한 데이터를 순차적으로 SD 카드에 기록하면서 병렬적으로 해시 계산을 수행한다.

### 3.5.4.2. 보안 통신 적용

```
(Top) → Component config → mbedTLS
Espressif IoT Development Framework Configuration
Memory allocation strategy (External SPIRAM) --->
[*] Asymmetric in/out fragment length
(16384) TLS maximum incoming fragment length
(8192) TLS maximum outgoing fragment length
[*] Using dynamic TX/RX buffer
[ ] Free SSL peer certificate after its usage
[ ] Free private key and DHM data after its usage
[ ] Enable mbedTLS debugging
mbedTLS v2.28.x related --->
Certificate Bundle --->
[ ] Enable mbedTLS ecsp restartable
[ ] Enable CMAC mode for block ciphers
[*] Enable hardware AES acceleration
[*] Use interrupt for long AES operations
[*] Enable hardware MPI (bignum) acceleration
[*] Enable hardware SHA acceleration
```

임베디드 시스템 파트에서는 TLS 통신 구현을 위해 ESP32 프레임워크에 기본 포함된 mbedTLS 라이브러리를 활용하였다. 일반적인 비보안 통신과 달리 TLS 기반 보안 통신은 암호화 / 복호화 과정에서 상당한 연산량과 메모리를 요구하기 때문에, 제한된 자원을 가진 MCU 환경에서는 mbedTLS 내부 버퍼 크기와 메모리 사용량을 세심하게 조정하는 과정이

---

필요하다. 본 연구에서는 이러한 제약을 고려하여 mbedTLS 설정을 최적화하였으며, 특히 파일 전송 과정에서 사용하는 청크 크기를 mbedTLS 의 입력 버퍼 크기에 맞추어 설계함으로써 기기가 안정적인 동작을 할 수 있도록 설계하였다.

모든 MQTT 통신 구간은 mTLS 기반으로 암호화된다. 이를 통해 네트워크 구간에서 발생할 수 있는 도청이나 중간자 공격을 방지하며, 서버와 기기가 상호 인증을 통해 신뢰할 수 있는 상태임을 보장한다. OTA 파일 다운로드 과정에서도 보안을 강화하기 위해, 청크 단위로 데이터를 수신할 때마다 지속적으로 해시를 갱신한다. 최종적으로 계산된 해시 값은 MQTT 메시지를 통해 전달받은 레퍼런스 해시와 비교하여 파일 무결성을 검증한다. 이 과정을 통과한 파일만이 실제 시스템 업데이트에 사용되므로, 저자원 임베디드 환경에서 악성 소프트웨어 업데이트 방지를 보장할 수 있다.

#### **3.5.4.3. 인증 및 무결성 관리**

장기 운영 환경에서도 안전하게 보안 통신을 유지할 수 있도록, MQTT 송신 태스크에는 장기적인 운영을 위한 클라이언트 인증서 관리 체계도 포함하였다. MQTT 초기화 단계에서 인증서의 유효 기간을 확인하며, 남은 기간이 한 달 이하일 경우 자동으로 CSR(Certificate Signing Request)을 생성하여 서버로 갱신 요청을 전송한다. 서버로부터 새로운 인증서를 수신하면, 기기는 해당 인증서를 SD 카드에 저장하고 기존 인증서를 교체한다.

#### **3.5.5. OTA 업데이트 및 롤백**

##### **3.5.5.1. OTA 프로세스 구조**

본 연구에서 구현한 OTA(Over-the-Air) 프로세스는 큐 기반의 태스크 구조로 동작한다. 새로운 펌웨어 다운로드가 완료되면, 해당 정보가 OTA 큐에 삽입되고 이를 대기 중이던 OTA 태스크가 수신하여 플래시에 기록을 수행한다. 최초 테스트 과정에서 플래시 내 대용량 데이터 쓰기 작업이 디스플레이 갱신과 충돌하여 화면 출력이 불안정해지는 문제를 확인하였고, 이를 해결하기 위해 플래시에 쓰기 작업이 진행되는 동안 일시적으로 화면을 비활성화하고, 펌웨어를 청크 단위로 나누어 순차적으로 기록하는 방식을 채택하였다. 기록이 완료되면 시스템은 5 초 대기 후 자동으로 재부팅된다.

재부팅 이후에는 새로운 펌웨어가 정상적으로 실행되는지를 확인한다. 정상 부팅에 실패하면 프로그램이 크래시되며 자동으로 재부팅되고, ESP-IDF 의 OTA 롤백 기능에 의해 이전

---

펌웨어가 복원된다. 반면 정상 부팅에 성공하면 부팅 직후 정상 부팅 플래그가 설정되어, 불필요하게 이전 펌웨어로 롤백되지 않도록 한다.

#### 3.5.4.2. 파티션 구조 활용

ESP-IDF 는 OTA 업데이트를 지원하기 위한 전용 파티션 구조를 제공한다. 본 연구에서는 이 구조를 활용하여 2 개의 OTA 파티션을 구성하고, 하나의 파티션에서 실행되는 동안 다른 파티션에 새 펌웨어를 기록하는 방식으로 롤백이 가능한 OTA 메커니즘을 구축하였다. 또한 ESP-IDF 가 제공하는 롤백 플래그를 관리하여, 부팅 실패 시 자동으로 이전 펌웨어가 복구되도록 하였다. 이 구조는 OTA 과정에서 발생할 수 있는 시스템 불안정을 최소화하고, 기기의 안전성을 높이는 핵심 요소로 작용한다.

#### 3.5.4.3. 무결성 검증

OTA 업데이트의 신뢰성을 확보하기 위해 다단계 무결성 검증 절차를 구현하였다. 파일 다운로드 단계에서는 MQTT 메시지로 전달받은 SHA-256 해시를 기준으로 수신한 파일의 무결성을 검증한다. 또한 다운로드한 파일의 크기를 사전에 정의된 값과 비교하여 불완전한 다운로드를 방지하였다. 마지막으로 OTA 를 실제로 수행하기 전, 펌웨어 파일 내에 포함된 ESP-IDF 고유의 매직 넘버를 확인하여 펌웨어 형식 자체가 올바른지를 검증하였다. 이와 같은 단계적 검증 절차를 통해 오류 가능성을 최소화할 수 있다.

#### 3.5.4.4. 롤백 메커니즘

업데이트 이후 새로운 펌웨어가 정상적으로 부팅되지 못할 경우, 시스템은 자동으로 롤백하여 이전 펌웨어를 복구한다. 본 연구에서는 듀얼 OTA 파티션 방식을 활용하여 롤백 메커니즘을 구현한다. 이는 플래시를 두 개의 OTA 파티션으로 분할하여, 하나의 펌웨어가 실행 중일 때 다른 파티션에 새로운 펌웨어를 기록하고, 기존 펌웨어를 플래시에 그대로 보존하여 업데이트 실패 시 해당 펌웨어로 돌아가는 방식이다. 연구에서 개발한 펌웨어는 충분히 경량화되어 있어 이 방법을 효과적으로 적용할 수 있었다.

일반적으로 롤백 메커니즘에는 두 가지 방법이 존재한다. 첫째는 본 연구에서 채택한 듀얼

---

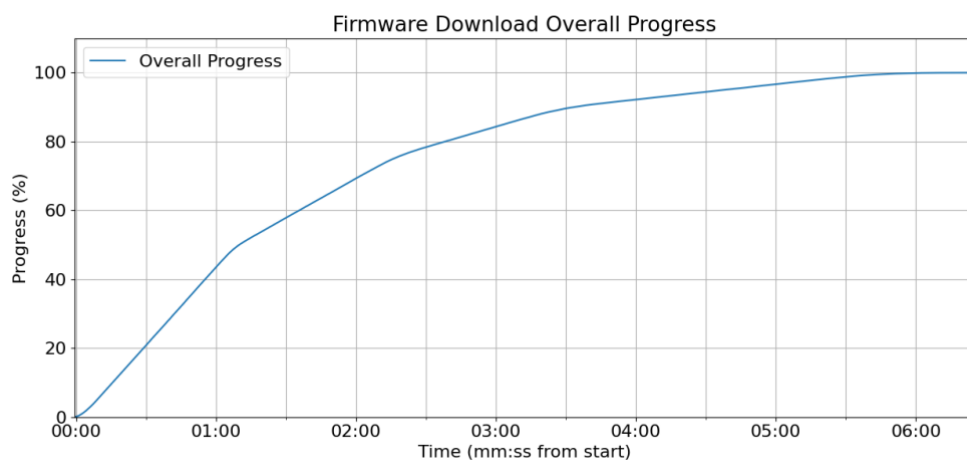
OTA 파티션 방식, 둘째는 플래시에 작은 부트로더 영역과 실행 펌웨어 영역을 구분하고, 필요 시 SD 카드에 보관된 이전 펌웨어를 다시 로드하는 방식이다. 두 번째 방법은 저장 장치 활용이 가능할 때 장점이 있으나, 본 연구의 범위에서는 경량 듀얼 파티션 방식이 더 효율적인 방법이다.

## 4. 연구 결과 분석 및 평가

본 프로젝트에서는 대규모 IoT 환경을 가정하여 시스템의 성능과 안정성을 검증하기 위해 다양한 시뮬레이션 실험을 수행하였다. 가상의 시뮬레이터 1,000 대를 대상으로 펌웨어 배포 테스트를 진행하였고, 또한 해외 거점(프랑스)에서 S3 Presigned URL 과 CloudFront Signed URL 을 비교하여 콘텐츠 전송 성능을 측정하였다. 그 결과, 제안한 아키텍처는 고부하 상황에서도 안정적으로 동작하며, 글로벌 서비스 확장 가능성을 충분히 확인할 수 있었다.

### 4.1. 펌웨어 배포 성능 실험

1,000 대 기기 시뮬레이터를 대상으로 약 1.5MB 크기의 펌웨어 파일을 동시에 배포하였다. 실험 결과, 전체 배포 성공률은 99.9%로 매우 높게 나타났으며, 전체 기기 배포 완료까지 평균 6 분 30 초가 소요되었다.



이 과정에서 MQTT Handler 의 병렬 메시지 처리 구조와 Redis-QuestDB-MySQL 기반의 이벤트 관리 구조가 안정적인 성능을 뒷받침하였다. 특히, 기기별 이벤트가 독립적으로

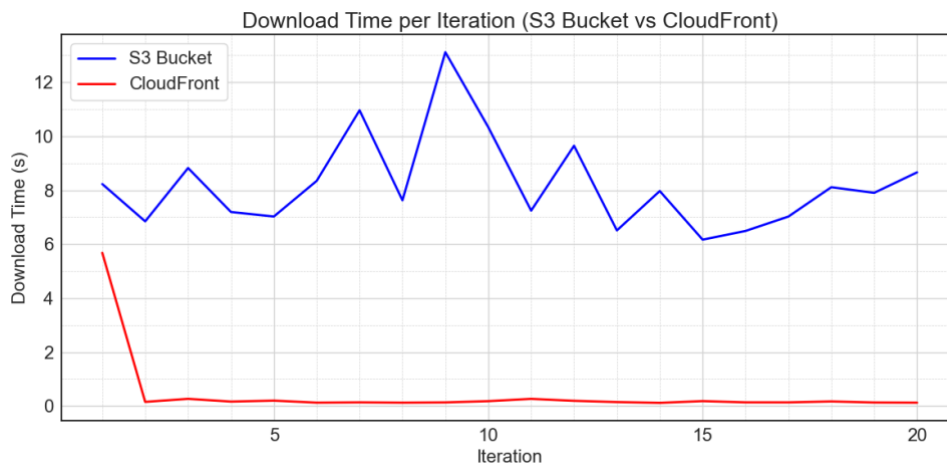


처리되면서 메시지 유실이나 연결 끊김과 같은 문제는 발생하지 않았으며, 이는 대규모 동시 배포 환경에서도 시스템이 충분히 확장 가능성을 입증하였다.

## 4.2. CloudFront(CDN) 기반 콘텐츠 전송 성능 실험

글로벌 서비스 확장성을 검증하기 위해, 프랑스 리전에 위치한 EC2 인스턴스를 활용하여 50MB 크기의 콘텐츠 파일을 반복 다운로드하는 테스트를 수행하였다. 테스트 방식은 동일한 콘텐츠를 S3 Presigned URL 과 CloudFront Signed URL 을 통해 각각 20 회 다운로드하여 평균 소요 시간을 비교하는 방식으로 진행하였다.

실험 결과, CloudFront(CDN)을 통한 다운로드 속도는 기존 S3 Presigned URL 대비 현저히 개선되었다. 구체적으로, 평균 다운로드 소요 시간이 약 8 초에서 0.2 초로 단축되었으며, 이는 글로벌 CDN 캐싱 효과가 유효하게 작동했음을 보여준다. 이러한 성능 개선은 해외 거점에서도 콘텐츠를 지연 없이 배포할 수 있음을 의미하며, 향후 글로벌 확장 시 서비스 품질을 보장하는 핵심 근거가 된다.



이상의 실험 결과를 종합하면, 본 시스템은 대규모 동시 배포 환경에서도 안정적으로 동작하며, 글로벌 사용자 환경에서도 빠른 콘텐츠 전송 성능을 제공할 수 있음을 입증하였다.

---

## 5. 결론 및 향후 연구 방향

본 연구는 대규모로 운영되는 MCU 기반 임베디드 장치 환경을 대상으로 클라우드 기반 OTA 관리 기술을 제안하고 구현하였다. 이를 통해 기존의 물리적 접근에 의존하던 유지보수 방식이 가진 비효율성을 극복하고, 수많은 장치를 중앙에서 통합적으로 관리할 수 있는 체계를 제시하였다. 클라우드 서버를 활용한 펌웨어 및 콘텐츠 배포와 장치 상태 모니터링 기능을 포함하여 대규모 IoT 장치의 운영의 안정성과 효율성을 확보하였으며, 롤백과 무결성 검증, mTLS 기반 보안 통신을 결합함으로써 신뢰성과 보안성을 강화하여 실제 산업 현장에 적용 가능한 수준의 관리 체계를 실증하였다.

이번 연구는 산업적 실용성 측면에서도 의미가 크다. 유지보수 비용 절감, 관리 효율성 향상, 보안 위협 대응 등 산업 환경에서 요구하는 핵심 요소와 안전성을 충족하였다. 특히 자원이 제한된 MCU 기반 장치에서도 OTA 기술을 안정적으로 구현하고, 시뮬레이션을 통해 대규모 환경을 검증함으로써, 단순한 기술 구현을 넘어 실질적 도입 가능성을 확인했다는 점에서 그 가치가 크다.

그러나 졸업과제의 범위상 한계점도 존재한다. 시뮬레이션을 통해 가상 환경 검증은 수행하였으나, 실제 대규모 하드웨어 환경에서의 테스트는 미흡하였다. 또한 시리얼 넘버 및 보안 키 관리와 같은 체계적 관리 방식의 고도화가 부족하였으며, 하드웨어 저수준 API 의존도가 높아 다양한 산업용 MCU 기반 장치와의 호환성을 충분히 검증하기 어려웠다. 초기 계획에 포함되었던 모니터링 기능 고도화나 콘텐츠 배포 스케줄링 등 일부 기능은 우선순위 조정 과정에서 축소되었다.

향후 연구에서는 이러한 한계를 보완하며 연구를 확장해나갈 것이다. 장치에서 송신되는 데이터를 수집-가공-처리-활용하는 파이프라인을 구축하여 모니터링의 활용성을 높이고, 시기별, 지역별로 최적화된 콘텐츠 배포가 가능하도록 관리 플랫폼을 고도화할 것이다. 더불어 실제 산업 현장에서 사용되는 다양한 이기종 MCU 기반 장치 환경에 적용할 수 있는 체계로 발전시켜, 장기적으로는 표준적인 관리 프로토콜로 자리잡을 수 있는 가능성을 찾고자 한다. 또한 실제 대규모 하드웨어 환경에서의 테스트를 통해 성능과 안정성을 검증하고, 메모리 누수 탐지 및 보안 취약점 점검 자동화와 같은 보안 기능을 고도화한다면,

---

본 연구 성과는 더욱 실질적인 가치와 학술적 기여를 동시에 확보할 수 있을 것이다.

결론적으로 본 연구는 클라우드 인프라와 임베디드 OTA 기술을 결합하여 대규모 IoT 장치 관리의 새로운 가능성을 제시했다는 점에서 의미가 크다. 앞으로 확장성과 보안성을 지속적으로 강화해 나간다면, 본 연구는 산업 현장에서 실질적으로 활용 가능한 수준을 넘어, 차세대 IoT 관리의 표준 체계로 발전할 수 있을 것으로 기대된다.

## 6. 참고 문헌

- [1] Prakash Chandrasekaran, K. B. Shibu Kumar, Remish L. Minz, Deepak D'Souza and Lomesh Meshram, "A multi-core version of FreeRTOS verified for data race and deadlock freedom", ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), pp. 62-71, Oct. 2014
- [2] Nicolas Melot, "Study of an operating system : FreeRTOS",  
[http://wiki.csie.ncku.edu.tw/embedded/FreeRTOS\\_Melot.pdf](http://wiki.csie.ncku.edu.tw/embedded/FreeRTOS_Melot.pdf)
- [3] coffee-is-essential, "coffee-driver", <https://github.com/coffee-is-essential/coffee-driver>
- [4] lovyano03, "LovyanGFX", <https://github.com/lovyano03/LovyanGFX>
- [5] TAMCTec, "gt911-arduino", <https://github.com/TAMCTec/gt911-arduino>
- [6] Espressif Systems, "ESP-IDF Getting Started", <https://idf.espressif.com/>
- [7] espressif, "arduino-esp32", <https://github.com/espressif/arduino-esp32>