

W I T H F O R C E

# MES 시스템 개선안

## Version 2.0

### 시스템 요구사항 명세서

작성일	2025년 11월 10일
작성자	Claude
문서 분류	기술 사양서

# Withforce MES 시스템 개선안 v2.0

문서 버전: v2.0

작성일: 2025-11-10

문서 유형: 시스템 요구사항 명세서

## 1. 프로젝트 개요

### 1.1 프로젝트 배경

현재 문제점:

- 생산 공정 전체에 대한 기록 및 추적 시스템 부재
- 제품 불량 발생 시 원인 추적 불가능
- 실시간 생산 현황 파악 불가
- 수작업 기반 관리로 인한 비효율성

개선 목표:

- 완벽한 제품 추적성(Traceability) 확보
- 실시간 생산 현황 모니터링
- 데이터 기반 품질 관리 체계 구축
- 생산 효율성 향상

### 1.2 프로젝트 범위

**Phase 1:**

- LOT 및 시리얼 번호 관리
- 공정별 착공/완공 관리
- 실시간 생산 현황 대시보드
- 기본 추적성 기능

**Phase 2:**

- 서버 이중화 (HA)
- 고급 분석 기능
- 모바일 대시보드

**1.3 주요 사용자**

사용자 유형	역할	주요 기능
생산 관리자	LOT 생성, 모니터링	React Dashboard
현장 작업자	착공/완공 처리	프론트엔드 앱 (PyQt5)
시스템 관리자	시스템 운영 관리	서버 관리, 백업

**2. 제품 및 공정 현황 (AS-IS)****2.1 제품 소개**

**제품명:** Withforce Wearable Robot

**용도:** 산업용/농업용 허리 보조 로봇

**핵심 구성품:** LMA (Linear Muscle Actuator)

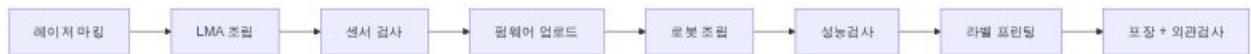
### LMA 구성:

- SMA (Shape Memory Alloy) 스프링
- 모션 케이블
- 제어 보드
- 센서 (온도, TOF)

## 2.2 주요 용어 정의

용어	설명
**LMA**	Linear Muscle Actuator - 로봇의 핵심 구동기
**SMA 스프링**	Shape Memory Alloy 스프링 - 온도로 구동
**모션**	SMA 스프링을 제조하는 원재료 케이블
**EOL 검사**	End Of Line 검사 - 최종 성능 검사
**TOF**	Time Of Flight 센서 - 거리 측정 센서
**LOT**	동일 조건으로 생산된 제품 묶음 (100개 단위)

## 2.3 현재 생산 공정 흐름



공정 개수: 8개

**일일 생산 목표:** 약 100대

**작업 방식:** LOT 단위 (100대), 작업대차로 이동

## 2.4 공정별 상세 설명

### 2.4.1 공정 1: 레이저 마킹

**작업 내용:**

- LMA에 LOT 번호 레이저 각인

### 2.4.2 공정 2: LMA 조립

**작업 내용:**

- 플라스틱 부품 + SMA 스프링 조립
- 조립 순서에 따른 수작업

**소요 시간:** 약 1시간

**품질 검사:** 작업자 육안 검사

### 2.4.3 공정 3: 센서 검사

**검사 항목:**

- 온도 센서 검사
- 검사 온도: 60°C

- 합격 기준:  $\pm 1^{\circ}\text{C}$  ( $59^{\circ}\text{C} \sim 61^{\circ}\text{C}$ )

- TOF 센서 검사
  - 검사 내용: I2C 통신 연결 확인

#### 2.4.4 공정 4: 펌웨어 업로드

##### 작업 내용:

- 대상: 제어 보드 MCU
- 펌웨어 제공: 서버 → Local App → 제어 보드
- 버전 관리: 서버에서 펌웨어 버전 관리

#### 2.4.5 공정 5: 로봇 조립

##### 작업 내용:

- LMA를 로봇 본체에 조립
- 케이블 연결 및 고정

#### 2.4.6 공정 6: 성능검사

##### 측정 항목:

- 온도 ( $^{\circ}\text{C}$ ): LMA 구동기 가열 온도
- 변위 (mm): LMA 구동기 이동 거리
- 힘 (kgf): LMA 구동기가 발생하는 힘

##### 검사 프로토콜 예시:

- 조건: 200mm 벤위에서 LMA 구동기를 52°C로 가열
- 측정: 발생하는 힘(kgf) 측정

#### 2.4.7 공정 7: 라벨 프린팅

작업 내용:

- 시리얼 번호 생성
- 바코드 라벨 출력 (Zebra 프린터)
- 로봇에 라벨 부착

시리얼 번호 형식: WF-KR-251110D-001-0001

#### 2.4.8 공정 8: 포장 + 외관검사

작업 내용:

- 로봇 외관 최종 검사 (스크래치, 오염 등)
- 비닐 박스에 포장

### 2.5 현재 문제점 및 개선 필요사항

#### 2.5.1 현재 생산 관리 방식

기록 방식: 없음 (전혀 기록하지 않음)

작업 지시: 구두 또는 화이트보드

**품질 관리:** 개별 작업자 판단

### 2.5.2 주요 문제점

#### 1. 추적성(Traceability) 부재

- 문제 발생 시 원인 추적 불가능
- 어떤 모선 케이블을 사용했는지 알 수 없음
- 어느 공정에서 불량이 발생했는지 파악 불가

#### 2. 실시간 현황 파악 불가

- 현재 생산 진행 상황을 알 수 없음
- 각 공정별 진행률 미파악
- 병목 구간 식별 불가

#### 3. 품질 데이터 미수집

- 온도, 변위, 힘 측정 데이터 저장 안 됨
- 통계 분석 불가능
- 개선 근거 데이터 부족

### 2.5.3 개선 목표

**정성적 목표:**

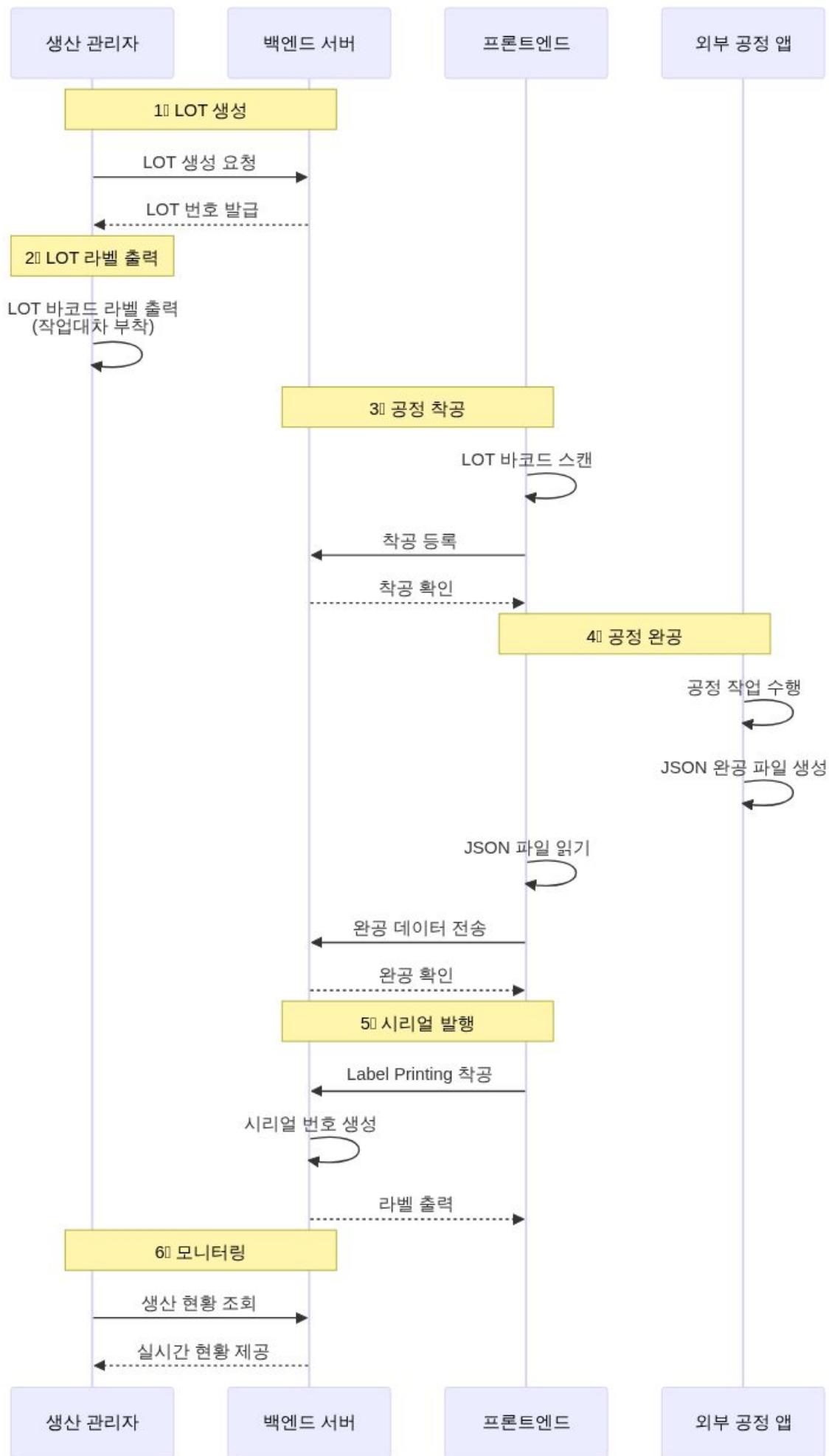
- **실시간 모니터링:** 언제 어디서나 생산 현황 파악
- **완벽한 추적성:** 시리얼 번호 → LOT → 공정 이력 → 부품 LOT 추적
- **데이터 기반 의사결정:** 측정 데이터 수집 및 분석
- **업무 효율성:** 자동화를 통한 작업 시간 단축

**정량적 목표:**

- LOT 생성 시간: 5분 → 30초
- 불량 원인 파악 시간: 불가능 → 5분 이내
- 재고 파악 시간: 1시간 → 실시간

### 3. MES 요구사항 (TO-BE)

#### 3.1 전체 프로세스 개요



## 3.2 핵심 기능 요구사항

### 3.2.1 LOT 관리

#### FR-LOT-001: LOT 생성

**생성 주체:** 생산 관리자

**생성 방법:**

- 관리자용 대시보드(React 웹 애플리케이션)에서 LOT 생성
- 입력 항목: 제품 모델, 목표 수량, 교대조
- 백엔드 서버에 LOT 생성 요청 전송

**처리 결과:**

- LOT 번호 자동 발급 (형식: WF-KR-251110D-001 )
- 생성 시 상태: CREATED
- LOT 정보 데이터베이스 저장

#### FR-LOT-002: LOT 라벨 출력

- LOT 바코드 라벨 1장 출력
- 작업대차에 부착하여 전 공정에서 식별

#### FR-LOT-003: LOT 상태 관리

- 상태: CREATED → IN\_PROGRESS → COMPLETED → CLOSED
- 각 상태 전환 이력 기록

### 3.2.2 시리얼 번호 관리

#### FR-SN-001: 시리얼 번호 생성

- Label Printing 공정 착공 시 자동 생성
- 형식: WF-KR-251110D-001-0001
- LOT당 순차적으로 발급

#### FR-SN-002: 시리얼 라벨 출력

- 시리얼 번호 바코드 라벨 자동 출력
- 라벨에 포함 정보: 시리얼, LOT, 제품명, 날짜

#### FR-SN-003: 라벨 재출력

- 라벨 손상/분실 시 재출력 기능
- 재출력 이력 기록 (사유, 작업자, 시간)

### 3.2.3 공정 관리

#### FR-PROC-001: 착공 처리

- LOT 바코드 스캔으로 착공 등록
- 실시간 검증 (이전 공정 완료 여부)
- 즉시 피드백 제공 (성공/실패)

#### FR-PROC-002: 완공 처리

- 외부 공정 앱에서 JSON 파일로 완공 데이터 전송
- 공정별 데이터 수집 (온도, 변위, 힘, 부품 LOT 등)
- 완공 검증 및 저장

#### FR-PROC-003: 공정 순서 제어

- 정의된 공정 순서 준수 강제
- 이전 공정 미완료 시 착공 불가

### 3.2.4 모니터링 및 대시보드

#### FR-DASH-001: 실시간 생산 현황

- 금일 생산 현황 (착공/완공/불량)
- LOT별 진행 상태
- 공정별 현황

#### FR-DASH-002: LOT 상세 조회

- LOT 기본 정보
- 공정별 진행 현황
- 시리얼 번호 목록

#### FR-DASH-003: 추적성 조회

- 시리얼 번호로 전체 이력 조회
- 공정별 작업 데이터
- 사용된 부품 LOT

### 3.3 비기능 요구사항

#### 3.3.1 성능

- 착공 API 응답 시간: 1초 이내
- 대시보드 로딩 시간: 3초 이내
- 동시 사용자: 20명

### 3.3.2 가용성

- 시스템 가동률: 99% (Phase 1)
- 서버 이중화 후: 99.9% (Phase 2)

### 3.3.3 데이터 보관

- 생산 데이터: 영구 보관
- 백업: 일일 전체 백업, 6시간 증분 백업
- 백업 보관: 30일

### 3.3.4 보안

- 사용자 인증: JWT 기반
- 역할 기반 접근 제어 (RBAC)
- 모든 변경 이력 감사 로그

## 3.4 데이터 인터페이스 요구사항

### 3.4.1 착공 인터페이스

**개요:** 현장 작업자가 프론트엔드 앱에서 LOT 바코드를 스캔하여 공정 착공을 등록하는 인터페이스

**통신 방식:** HTTP REST API (프론트엔드 ↔ 백엔드)

**API 엔드포인트:** `POST /api/v1/process/start`

**요청 (Request) 스키마:**

**JSON**

```
{\n    "lot_number": "WF-KR-251110D-001",\n    "process_id": "PROC-001",\n    "process_name":\n        "레이저 마킹",\n    "worker_id": "W001",\n    "start_time": "2025-01-10T09:00:00+09:00"\n}
```

**필드 설명:**

필드	타입	필수	설명
lot_number	string	Y	LOT 번호 (바코드 스캔 값)
process_id	string	Y	공정 ID (PROC-001 ~ PROC-008)
process_name	string	Y	공정명 (한글)
worker_id	string	Y	작업자 ID
start_time	string	Y	착공 시간 (ISO 8601)

**응답 (Response) - 성공:****JSON**

```
{\n    "status": "success",\n    "message": "착공이 등록되었습니다",\n    "data": {\n        "lot_number": "WF-KR-251110D-001",\n        "process_id": "PROC-001",\n        "process_name":\n            "레이저 마킹",\n        "worker_id": "W001",\n        "start_time": "2025-01-10T09:00:00+09:00",\n        "work_order_id": "WO-20250110-001",\n        "sequence_number": 1\n    }\n}
```

**응답 (Response) - 실패:****JSON**

```
{\n    "status": "error",\n    "message": "이전 공정이 완료되지 않았습니다",\n    "error_code":\n        "PREVIOUS_PROCESS_NOT_COMPLETED",\n    "data": {\n        "lot_number": "WF-KR-251110D-001",\n        "current_process": "센서 검사",\n        "previous_process": "LMA 조립",\n        "required_action": "LMA 조립 공정을 먼저 완료해주세요"\n    }\n}
```

### 주요 에러 코드:

에러 코드	설명	HTTP 상태
LOT_NOT_FOUND	LOT 번호가 존재하지 않음	404
PREVIOUS_PROCESS_NOT_COMPLETED	이전 공정 미완료	400
DUPLICATE_START	이미 착공된 공정	409
INVALID_PROCESS_SEQUENCE	잘못된 공정 순서	400
WORKER_NOT_FOUND	작업자 ID 없음	404

### 3.4.2 완공 데이터 인터페이스

**개요:** 외부 공정 앱에서 공정 작업 완료 후 완공 데이터를 MES 백엔드로 전송하는 인터페이스

**통신 방식:** JSON 파일 기반 (File Watcher)

#### 배경:

- 외부 업체가 개발한 공정 앱 (레이저 마킹, 센서 검사, 성능검사 등)
- 소스 코드 접근 불가, API 연동 불가
- 파일 기반 데이터 교환 방식 채택

#### 데이터 전송 프로세스:

- 외부 공정 앱이 작업 완료 후 JSON 파일 생성
- 지정된 디렉토리에 파일 저장: `C:\neurohub_work\pending\`
- 프론트엔드 앱(PyQt5)의 File Watcher가 파일 감지
- 프론트엔드가 JSON 파일을 읽고 HTTP POST로 백엔드에 전송
- 백엔드가 완공 데이터 파싱, 검증 및 데이터베이스 저장
- 프론트엔드가 처리 완료 파일을 `C:\neurohub_work\completed\`로 이동
- 전송 실패 시: `C:\neurohub_work\error\`로 이동하고 오류 로그 생성

## 기본 JSON 스키마:

### JSON

```
{\n    "lot_number": "WF-KR-251110D-001",\n    "process_id": "PROC-003",\n    "process_name":\n        "센서 검사",\n    "worker_id": "W001",\n    "start_time": "2025-01-10T09:00:00+09:00",\n    "complete_time": "2025-01-10T09:15:00+09:00",\n    "process_data": {\n        "공정별 데이터 (아래 참조)": "..."\n    }\n}
```

## 공정별 process\_data 상세 스키마:

### 공정 1: 레이저 마킹

### JSON

```
{\n    "process_data": {\n        "lot_number_engraved": "WF-KR-251110D-001",\n        "marking_result": "SUCCESS"\n    }\n}
```

### 공정 2: LMA 조립

### JSON

```
{\n    "process_data": {\n        "sma_spring_lot": "SPRING-2025011001",\n        "busbar_lot":\n            "BUSBAR-2025011001"\n    }\n}
```

### 공정 3: 센서 검사

### JSON

```
{\n    "process_data": {\n        "temp_sensor": {\n            "measured_temp": 60.2,\n            "target_temp": 60.0,\n            "tolerance": 1.0,\n            "result": "PASS"\n        },\n        "tof_sensor": {\n            "i2c_communication": true,\n            "result": "PASS"\n        },\n        "overall_result": "PASS"\n    }\n}
```

### 공정 4: 펌웨어 업로드

**JSON**

```
{\n  "process_data": {\n    "firmware_version": "v1.2.3",\n    "upload_result":\n      "SUCCESS"\n  }\n}
```

**공정 6: 성능검사****JSON**

```
{\n  "process_data": {\n    "test_results": [\n      {\n        "test_point_id":\n          "T38_P170",\n        "spec": {\n          "target_force": 12.8,\n          "tolerance": 1.0\n        },\n        "result":\n          "PASS"\n      },\n      {\n        "test_point_id": "T50_P180",\n        "spec": {\n          "target_force": 14.5,\n          "tolerance": 1.0\n        },\n        "result": "PASS"\n      },\n      {\n        "test_point_id": "T52_P200",\n        "spec": {\n          "target_force": 16.0,\n          "tolerance": 1.0\n        },\n        "result": "PASS"\n      }\n    ],\n    "overall_result": "PASS",\n    "test_duration_seconds": 45,\n    "tested_at": "2025-01-10T11:00:00+09:00"\n  }\n}
```

**공정 7: 라벨 프린팅****JSON**

```
{\n  "process_data": {\n    "label_printed": true,\n    "printer_id": "PRINTER-07",\n    "print_time": "2025-01-10T14:30:00+09:00",\n    "barcode_verified": true\n  }\n}
```

**공정 8: 포장 + 외관검사****JSON**

```
{\n  "process_data": {\n    "inspection_result": "PASS"\n  }\n}
```

**참고:** 향후 포장 라벨 출력이 필요한 경우 `packaging_label_printed`, `packaging_printer_id` 등의 필드를 추가할 수 있습니다.

## 파일명 규칙:

- 형식: `{LOT_NUMBER}_{PROCESS_ID}_{TIMESTAMP}.json`
- 예시: `WF-KR-251110D-001_PROC-003_20250110090000.json`

## 처리 결과:

- 성공: 파일을 `C:\neurohub_work\completed\`로 이동
- 실패: 파일을 `C:\neurohub_work\error\`로 이동하고 오류 로그 생성

### 3.4.3 라벨 출력 인터페이스

**개요:** LOT 바코드 라벨 및 시리얼 번호 바코드 라벨을 산업용 프린터로 출력하는 인터페이스

**통신 방식:** 직렬 통신 (Serial) 또는 네트워크 (TCP/IP)

## 지원 프린터:

- Zebra ZT410, ZT230
- TSC TTP-244 Pro, TTP-345
- Godex G500, G530
- 기타 ZPL 또는 ESC/POS 명령어 지원 프린터

## 출력 대상:

라벨 종류	출력 시점	포함 정보	관련 요구사항
LOT 바코드 라벨	LOT 생성 직후	LOT 번호, 제품명, 생성일, 목표 수량	FR-LOT-002
시리얼 번호 바코드 라벨	Label Printing 공정 착공 시	시리얼 번호, LOT 번호, 제품명, 생성일	FR-SN-002

## 라벨 출력 요청 API:

**API 엔드포인트:** POST /api/v1/label/print

### 요청 (Request) - LOT 라벨:

#### JSON

```
{\n    "label_type": "LOT",\n    "lot_number": "WF-KR-251110D-001",\n    "product_model":\n        "Withforce",\n    "target_quantity": 100,\n    "created_date": "2025-01-10",\n    "printer_id": "PRINTER-01"\n}
```

### 요청 (Request) - 시리얼 라벨:

#### JSON

```
{\n    "label_type": "SERIAL",\n    "serial_number": "WF-KR-251110D-001-0001",\n    "lot_number": "WF-KR-251110D-001",\n    "product_model": "Withforce",\n    "created_date":\n        "2025-01-10",\n    "printer_id": "PRINTER-07"\n}
```

### 응답 (Response) - 성공:

#### JSON

```
{\n    "status": "success",\n    "message": "라벨이 출력되었습니다.",\n    "data": {\n        "label_type": "SERIAL",\n        "serial_number": "WF-KR-251110D-001-0001",\n        "printer_id": "PRINTER-07",\n        "print_time": "2025-01-10T14:30:00+09:00"\n    }\n}
```

### 응답 (Response) - 실패:

#### JSON

```
{\n    "status": "error",\n    "message": "프린터 연결 실패",\n    "error_code":\n        "PRINTER_NOT_CONNECTED",\n    "data": {\n        "printer_id": "PRINTER-07",\n        "printer_status": "offline"\n    }\n}
```

### 주요 에러 코드:

에러 코드	설명	조치 방법
PRINTER_NOT_FOUND	프린터 ID가 등록되지 않음	프린터 설정 확인
PRINTER_NOT_CONNECTED	프린터 연결 끊김	프린터 전원 및 케이블 확인
PRINTER_OUT_OF_PAPER	용지 부족	라벨 용지 보충
INVALID_LABEL_DATA	라벨 데이터 형식 오류	요청 데이터 확인

### 3.4.4 펌웨어 배포 인터페이스

**개요:** 공정 4 (펌웨어 업로드) 착공 시 최신 펌웨어를 백엔드에서 프론트엔드로 배포하고, 외부 로컬 앱이 제어 보드에 업로드하는 인터페이스

#### 배경:

- 펌웨어는 백엔드 서버에서 중앙 관리
- 버전 불일치 방지를 위한 동기화 메커니즘 필요
- 로컬 앱이 항상 최신 펌웨어만 사용하도록 보장

#### 디렉토리 구조:

```
C:\neurohub_work\firmware\n   └── firmware_v1.2.3.bin      (펌웨어 바이너리 파일)\n   └── firmware_meta.json    (메타데이터 파일)
```

#### 펌웨어 동기화 프로세스:

- 착공 시 버전 확인 (프론트엔드)
  - 공정 4 착공 API 호출
  - 백엔드가 최신 펌웨어 정보를 응답에 포함
  - 프론트엔드가 로컬 `firmware_meta.json` 과 비교

- 펌웨어 다운로드 (버전 불일치 시)
  - 기존 `.bin` 파일 삭제
  - 백엔드에서 최신 펌웨어 다운로드
  - 새 펌웨어 파일 저장: `firmware_v{version}.bin`
  - `firmware_meta.json` 업데이트
- 로컬 앱 펌웨어 업로드
  - 로컬 앱이 `firmware_meta.json` 감시 (File Watcher)
  - 파일 변경 감지 시 메타데이터 읽기
  - `filename`에 지정된 `.bin` 파일 존재 확인
  - MD5 해시 검증
  - 제어 보드에 펌웨어 업로드
- 완공 보고
  - 업로드 결과를 JSON 파일로 생성
  - 프론트엔드가 백엔드로 완공 보고

#### 착공 API 응답 (공정 4 전용):

##### JSON

```
{
  "status": "success",
  "message": "착공이 등록되었습니다",
  "data": {
    "lot_number": "WF-KR-251110D-001",
    "process_id": "PROC-004",
    "process_name": "펌웨어 업로드",
    "worker_id": "W001",
    "start_time": "2025-01-10T10:00:00+09:00",
    "work_order_id": "W0-20250110-004",
    "firmware_info": {
      "version": "v1.2.3",
      "filename": "firmware_v1.2.3.bin",
      "file_size": 65536,
      "md5_hash": "5d41402abc4b2a76b9719d911017c592",
      "download_url": "/api/v1/firmware/download/v1.2.3"
    }
  }
}
```

#### 펌웨어 다운로드 API:

API 엔드포인트: `GET /api/v1/firmware/download/{version}`

요청 예시:

```
GET /api/v1/firmware/download/v1.2.3\nAuthorization: Bearer {JWT_TOKEN}
```

### 응답:

- Content-Type: `application/octet-stream`
- Content-Disposition: `attachment; filename="firmware_v1.2.3.bin"`
- Binary file stream

### `firmware_meta.json` 스키마:

#### JSON

```
{\n  "version": "v1.2.3",\n  "filename": "firmware_v1.2.3.bin",\n  "file_size":\n    65536,\n  "md5_hash": "5d41402abc4b2a76b9719d911017c592",\n  "downloaded_at":\n    "2025-01-10T10:00:15+09:00",\n  "status": "READY",\n  "target_mcu": "STM32F103"\n}
```

### 필드 설명:

필드	타입	설명
version	string	펌웨어 버전 (예: v1.2.3)
filename	string	펌웨어 파일명
file_size	integer	파일 크기 (bytes)
md5_hash	string	MD5 체크섬 (무결성 검증용)
downloaded_at	string	다운로드 완료 시간 (ISO 8601)
status	string	상태 (READY, UPLOADING, uploaded, ERROR)
target_mcu	string	대상 MCU 정보

### 로컬 앱 연동 가이드:

로컬 앱은 다음 로직을 구현해야 합니다:

- **파일 감시:** `firmware_meta.json` 변경 감지
- **메타데이터 읽기:** JSON 파싱
- **파일 검증:**
  - `filename` 필드의 파일 존재 확인
  - MD5 해시 계산 및 `md5_hash` 와 비교
- **펌웨어 업로드:**
  - 제어 보드에 시리얼/USB로 업로드
  - `status` 를 "UPLOADING"으로 업데이트 (선택사항)
- **결과 보고:** 완공 JSON 파일 생성

완공 데이터 (공정 4: 펌웨어 업로드) 업데이트:

#### JSON

```
{\n  "process_data": {\n    "firmware_version": "v1.2.3",\n    "upload_result":\n      "SUCCESS"\n  }\n}
```

에러 처리:

상황	처리 방법
다운로드 실패	프론트엔드가 재시도 (최대 3회), 실패 시 작업자에게 알림
MD5 불일치	로컬 앱이 펌웨어 업로드 중단, 완공 JSON에 오류 기록
업로드 실패	로컬 앱이 재시도 (최대 3회), 완공 JSON에 실패 기록
구버전 펌웨어 감지	프론트엔드가 자동으로 최신 버전 다운로드

## 3.5 기능 검수 항목

이 섹션은 MES 시스템의 각 기능 요구사항이 올바르게 구현되었는지 검증하기 위한 인수 검수 (Acceptance Test) 항목을 정의합니다.

### 3.5.1 LOT 관리 검수

검수 ID	검수 항목	검수 기준	검수 방법
TC-LOT-001	LOT 생성 기능	관리자 대시보드에서 LOT 생성 시 형식 준수 ('WF-KR-YYMMDD{D/N}-XXX')	대시보드에서 LOT 생성 후 데이터베이스 확인
TC-LOT-002	LOT 번호 자동 발급	당일 순번이 자동으로 증가 (001, 002, ...)	동일 날짜에 여러 LOT 생성 후 순번 확인
TC-LOT-003	LOT 라벨 출력	LOT 생성 시 바코드 라벨 자동 출력 성공	Zebra 프린터로 라벨 출력 확인
TC-LOT-004	중복 LOT 방지	동일한 LOT 번호를 재생성 시도 시 오류 반환	동일 LOT 번호로 생성 시도 후 오류 메시지 확인
TC-LOT-005	LOT 상태 관리	LOT 생성 시 CREATED 상태, 모든 공정 완료 시 COMPLETED	각 상태 변경 시점에 DB 확인
TC-LOT-006	LOT 조회 기능	대시보드에서 LOT 번호로 상세 정보 조회 가능	LOT 검색 후 기본 정보 및 공정 현황 표시 확인

### 3.5.2 시리얼 번호 관리 검수

검수 ID	검수 항 목	검수 기준	검수 방법
TC-SN-001	시리얼 번호 생성	라벨 프린팅 공정 착공 시 자동 생성, 형식 준수 ('WF-KR-YYMMDD{D/N}-XXX-YYYY')	라벨 프린팅 착공 후 DB에서 시리얼 번호 확인
TC-SN-002	시리얼 순차 발급	LOT당 0001부터 순차 증가	동일 LOT 내 여러 제품의 시리얼 번호 확인
TC-SN-003	시리얼 라벨 출력	시리얼 번호 생성 시 바코드 라벨 자동 출력	Zebra 프린터로 라벨 출력 및 내용 확인
TC-SN-004	라벨 재 출력 기능	라벨 손상/분실 시 재출력 가능, 이력 기록됨	재출력 요청 후 이력 테이블 확인 (사유, 작업자, 시간)
TC-SN-005	시리얼 중복 방지	동일 시리얼 번호 재발급 불가	동일 시리얼 생성 시도 시 오류 확인

### 3.5.3 공정 관리 검수

검수 ID	검수 항목	검수 기준	검수 방법
TC-PROC-001	착공 등록	LOT 바코드 스캔 시 착공 API 정상 호출 및 성공 응답	프론트엔드에서 바코드 스캔 후 HTTP 200 응답 확인
TC-PROC-002	착공 검증	이전 공정 미완료 시 착공 불가, 적절한 오류 메시지 반환	공정 순서 위반 시도 후 에러 코드 확인 ('PREVIOUS_PROCESS_NOT_COMPLETED')
TC-PROC-003	완공 등록	JSON 파일 생성 시 백엔드가 자동 감지 및 처리	File Watcher가 JSON 파일 감지, DB 저장 확인
TC-PROC-004	완공 데이터 검증	공정별 필수 데이터 누락 시 오류 처리	필수 필드 누락 JSON 전송 후 오류 로그 확인
TC-PROC-005	완공 파일 이동	처리 성공 시 'completed/' , 실패 시 'error/'로 이동	각 경우의 파일 이동 경로 확인
TC-PROC-006	공정 순서 제어	정의된 순서대로만 진행 가능	순서 위반 시도 시 착공 차단 확인
TC-PROC-007	작업자 기록	각 공정 착공/완공 시 작업자 ID 기록	DB에서 작업자 정보 확인
TC-PROC-008	시간 기록	착공/완공 시간 정확히 기록	DB 타임스탬프와 실제 시간 비교 (오차 1초 이내)

### 3.5.4 실시간 모니터링 검수

검수 ID	검수 항목	검수 기준	검수 방법
TC-DASH-001	금일 생산 현황	착공/완공/불량 수량 정확히 표시	DB 집계값과 대시보드 표시값 비교
TC-DASH-002	LOT별 진행 상태	각 LOT의 현재 공정 및 진행률 표시	대시보드 표시값과 실제 DB 데이터 비교
TC-DASH-003	실시간 업데이트	데이터 변경 시 5초 이내 대시보드 반영	착공/완공 후 대시보드 업데이트 시간 측정
TC-DASH-004	대시보드 로딩 시간	초기 로드 시간 3초 이내	브라우저 개발자 도구로 로딩 시간 측정

### 3.5.5 추적성 검수

검수 ID	검수 항목	검수 기준	검수 방법
TC-TRACE-001	LOT 이력 조회	LOT 번호로 전체 공정 이력 조회 가능	대시보드에서 LOT 검색 후 8개 공정 이력 확인
TC-TRACE-002	시리얼 이력 조회	시리얼 번호로 개별 제품의 상세 이력 조회	시리얼 검색 후 공정별 데이터 확인
TC-TRACE-003	부품 LOT 추적	사용된 부품 LOT 정보 기록 및 조회 가능	조립 공정 데이터에서 부품 LOT 확인 (모선, 링크 등)
TC-TRACE-004	불량 추적	불량 발생 시 해당 공정 및 원인 추적 가능	불량 데이터 입력 후 이력 조회로 추적
TC-TRACE-005	작업자 추적	각 공정 작업자 정보 조회 가능	이력 조회 시 작업자 ID 표시 확인
TC-TRACE-006	데이터 무결성	모든 공정 데이터 누락 없이 저장	샘플 LOT의 전체 공정 데이터 완전성 확인

## 4. 시스템 아키텍처

## 4.0 시스템 규모 및 성능 요구사항

### 설계 목표 규모:

항목	목표 값	비고
**동시 접속자**	100명	작업자 + 관리자 + 외부 시스템
**작업 PC**	7-20대	공정별 확장 가능
**관리자 대시보드**	10-30명	생산 관리자, 품질 관리자, 경영진
**일일 트랜잭션**	~50,000건	착공/완공 각 800건 + 조회 48,000건
**피크 TPS**	~20 TPS	오전 9시, 오후 1시 작업 시작 시간대
**API 응답 시간**	< 200ms (P95)	착공/완공 API 기준
**대시보드 로딩**	< 2초	초기 로드 시간
**WebSocket 연결**	100+	실시간 모니터링
**데이터 보관**	3년	법적 요구사항 준수
**시스템 가동률**	99.5%	월 최대 다운타임 3.6시간

### 성능 산출 근거:

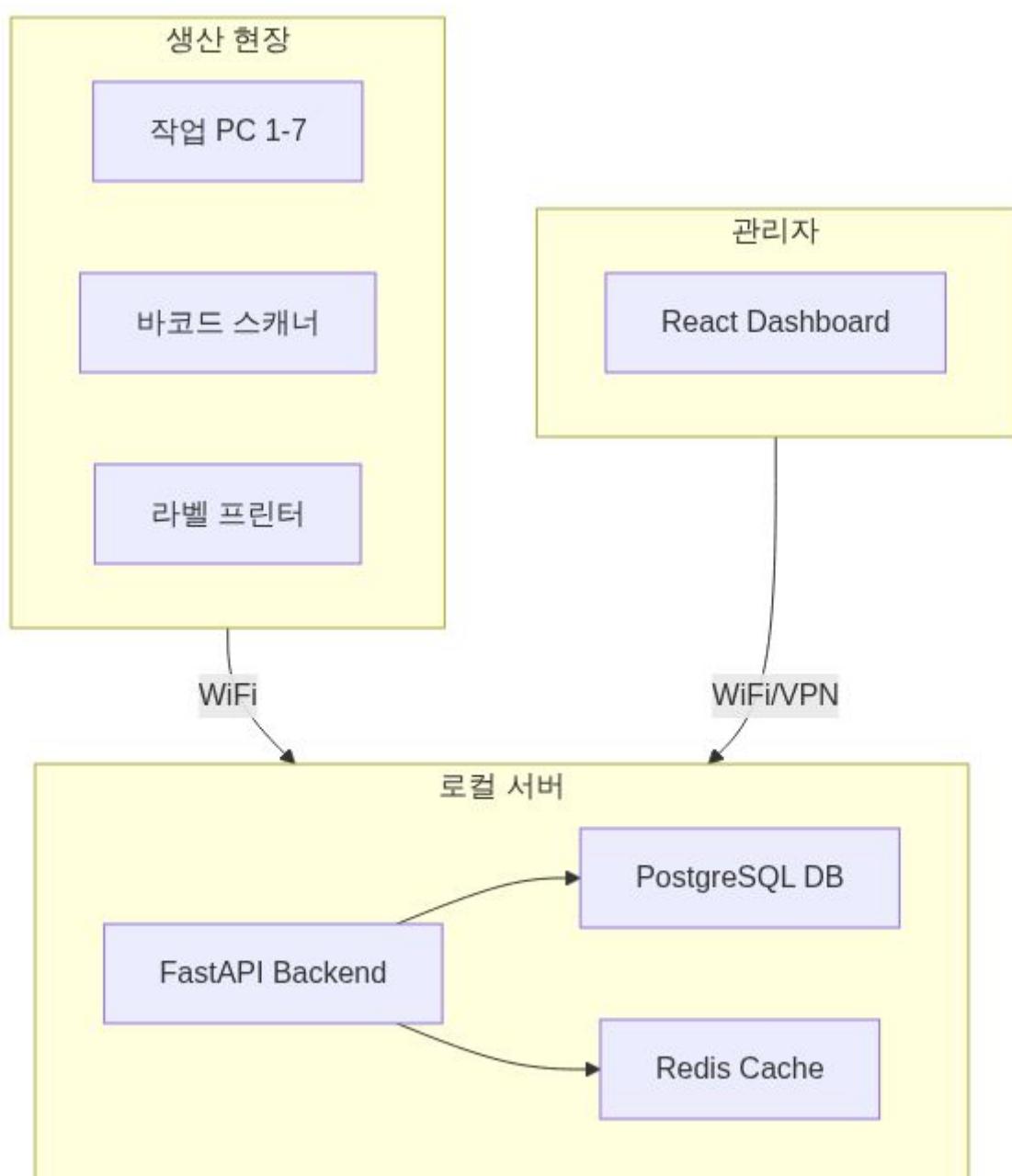
일일 생산량: 100대  
공정 수: 8개  
작업일: 주 5일  
착공/완공 트랜잭션: 100대 × 8공정 × 2(착공 +완공) = 1,600건/일  
대시보드 조회: 30명 × 평균 50회/일 = 1,500건/일  
모니터링 API: 10명 × 10초마다 × 8시간 = 28,800건/일  
기타 조회: 약 20,000건/일  
총합: 약 50,000건/일  
피크 시간(2시간): 10,000건 / 7,200초 ≈ 1.4 TPS (여유율 고려 20 TPS 설계)

### 확장성 요구사항:

- 향후 2년 내 동시 접속자 **200명**까지 확장 가능해야 함
- 일일 생산량 **500대** 규모까지 대응 가능해야 함

- 수평 확장(Scale-out) 가능한 아키텍처 설계

## 4.1 전체 구성도



## 4.2 기술 스택

**설계 기준:** 동시 접속자 100명 규모

### 4.2.1 Backend

**언어 및 프레임워크:**

- 언어: Python 3.11+
- 프레임워크: FastAPI
- ASGI 서버: Uvicorn (multi-worker)
- 프로세스 관리: Gunicorn + Uvicorn workers
- ORM: SQLAlchemy 2.0 (async)
- 마이그레이션: Alembic

**성능 최적화:**

- Worker 프로세스: CPU 코어 수 × 2 (예: 4코어 → 8 workers)
- Connection Pool:
  - 최소 연결: 10
  - 최대 연결: 50
  - Overflow: 20
  - 비동기 I/O: asyncio, httpx

### 4.2.2 Database

**Primary Database:**

- DBMS: PostgreSQL 15+
- Connection Pool: asyncpg
- pool\_size: 20-50 (동시 접속자에 따라 조정)
- max\_overflow: 10
- 인덱싱 전략:
  - lot\_number, process\_id, worker\_id에 인덱스
  - 복합 인덱스: (lot\_number, process\_id, created\_at)
  - 파티셔닝: 3개월 단위 날짜 파티셔닝 (선택사항)

**Caching:**

- L1 Cache: 애플리케이션 인메모리 (LRU, max 1000 items)
- L2 Cache: Redis 7.0+
- 용도: 세션, 임시 데이터, API 캐시
- TTL: 5분~1시간 (데이터 특성별)
- 메모리: 4GB 권장
- Persistence: AOF (Append Only File)

**4.2.3 Load Balancing & Reverse Proxy****Reverse Proxy:**

- Nginx 1.24+
- HTTP/2 지원
- SSL/TLS 종단 처리
- Static file 서빙 (React build)
- Gzip 압축
- Rate Limiting: 100 req/min per IP

**Load Balancing:**

- 알고리즘: Least Connections
- Health Check: 10초마다 /health 엔드포인트 확인
- Timeout: 30초
- 백엔드 서버: 2대 (Phase 2)

**4.2.4 Frontend (작업 PC)****프레임워크:**

- PyQt5 (Python 3.11+)
- 통신: httpx (async HTTP client)
- 파일 감시: watchdog
- 프린터 통신: pyserial
- 바코드 스캐너: USB HID

#### 로컬 저장소:

- SQLite (오프라인 작업 지원)
- 디렉토리: C:\neurohub\_work\

### 4.2.5 Dashboard (관리자)

#### Frontend:

- 프레임워크: React 18+
- 상태 관리: Zustand 또는 Redux Toolkit
- UI 라이브러리: Material-UI (MUI) 또는 Ant Design
- 차트: Chart.js 또는 Recharts
- 테이블: TanStack Table (React Table)
- HTTP Client: Axios
- 빌드 도구: Vite

#### 실시간 통신:

- WebSocket (Socket.io 또는 native WebSocket)
- Redis Pub/Sub (멀티 서버 환경 대응)
- 재연결 로직: 자동 재연결 (exponential backoff)

### 4.2.6 Infrastructure

#### 운영 체제:

- OS: Ubuntu 22.04 LTS
- 컨테이너: Docker + Docker Compose (권장)
- 오케스트레이션: Docker Swarm (선택사항)

#### 고가용성 (Phase 2):

- HA Solution: Keepalived
- Virtual IP: 192.168.1.100
- Failover: 자동 (30초 이내)

#### 백업:

- Database:
  - 일일 백업: pg\_dump (cron, 새벽 2시)
  - 증분 백업: WAL archiving (선택사항)
  - 보관 주기: 30일
- 파일 시스템:
  - 완공 JSON: 90일 보관
  - 펌웨어: 버전별 영구 보관
  - 백업 위치: NAS 또는 외부 스토리지

#### 4.2.7 모니터링 & 로깅

##### 로깅:

- 구조화 로깅: structlog (JSON 포맷)
- 로그 레벨: INFO (production), DEBUG (development)
- 로그 저장:
  - 로컬 파일: /var/log/neurohub/ (30일 rotation)
  - 중앙 집중: Loki (선택사항)

##### 모니터링:

- 메트릭 수집: Prometheus
- 수집 간격: 15초
- 보관 주기: 90일
- 시각화: Grafana
- 대시보드: 시스템 상태, API 성능, 비즈니스 메트릭
- 알림: Alertmanager → Slack/Email
- CPU > 80% (5분 이상)
- 메모리 > 85%
- API 응답 시간 > 500ms (P95)
- 에러율 > 1%

##### APM (선택사항):

- OpenTelemetry (분산 추적)
- Jaeger (트레이싱 시각화)

## 4.2.8 보안

### 인증 & 인가:

- JWT (Access Token + Refresh Token)
- Access Token TTL: 15분
- Refresh Token TTL: 7일
- RBAC (Role-Based Access Control)
- 역할: 관리자, 생산관리자, 작업자, 뷰어

### 네트워크 보안:

- HTTPS: Nginx + Let's Encrypt (또는 자체 인증서)
- 방화벽: UFW (Ubuntu Firewall)
- 허용 포트: 22 (SSH), 80 (HTTP), 443 (HTTPS)
- 내부망만 허용: PostgreSQL (5432), Redis (6379)
- Rate Limiting: slowapi (FastAPI middleware)

### 데이터 보안:

- 암호화: bcrypt (비밀번호 해싱)
- 감사 로그: 모든 CRUD 작업 기록
- SQL Injection 방지: ORM (SQLAlchemy parameterized queries)

## 4.2.9 테스트

### Backend:

- 단위 테스트: pytest
- API 테스트: pytest + httpx
- 부하 테스트: Locust (목표: 20 TPS, 100 동시 사용자)

### Frontend:

- React: Jest + React Testing Library
- E2E: Playwright (선택사항)

#### 4.2.10 개발 도구

##### 버전 관리:

- Git + GitHub/GitLab
- 브랜치 전략: Git Flow

##### CI/CD (Phase 2):

- GitLab CI 또는 GitHub Actions
- 자동 테스트 → 빌드 → 배포

##### API 문서:

- FastAPI 자동 생성 (Swagger UI, ReDoc)
- 엔드포인트: /docs, /redoc

#### 4.2.11 기술 스택 요약표

계층	기술	목적	우선순위
**Load Balancer**	Nginx	부하 분산, SSL 종단	High
**Backend**	FastAPI + Uvicorn	REST API 서버	High
**Database**	PostgreSQL 15+	관계형 데이터 저장	High
**Cache**	Redis	세션, API 캐시	High
**ORM**	SQLAlchemy 2.0	데이터베이스 추상화	High
**Frontend (작업 PC)**	PyQt5	데스크톱 앱	High
**Dashboard**	React 18	관리자 웹	High
**WebSocket**	FastAPI WebSocket	실시간 통신	High
**모니터링**	Prometheus + Grafana	시스템 모니터링	High
**로깅**	structlog	구조화 로그	Medium
**백업**	pg_dump + Cron	데이터 백업	High
**컨테이너**	Docker	배포 환경 관리	Medium
**HA**	Keepalived	고가용성 (Phase 2)	Low
**APM**	OpenTelemetry	분산 추적 (선택)	Low

## 4.3 배포 아키텍처

**설계 목표:** 동시 접속자 100명 지원, 향후 200명까지 확장 가능

### 4.3.1 Phase 1: 단일 서버 구성 (현재)

아키텍처:



### 서버 스펙 권장:

- CPU: 8 코어 (Intel Xeon 또는 AMD EPYC)
- RAM: 32GB
- SSD: 500GB (NVMe)
- 네트워크: 1Gbps
- OS: Ubuntu 22.04 LTS

### 컴포넌트 배치:

컴포넌트	포트	프로세스 수	메모리
Nginx	80, 443	1	512MB
FastAPI (Uvicorn)	8000	8 workers	4GB (총)
PostgreSQL	5432	1	8GB
Redis	6379	1	4GB
Prometheus	9090	1	2GB
Grafana	3000	1	512MB

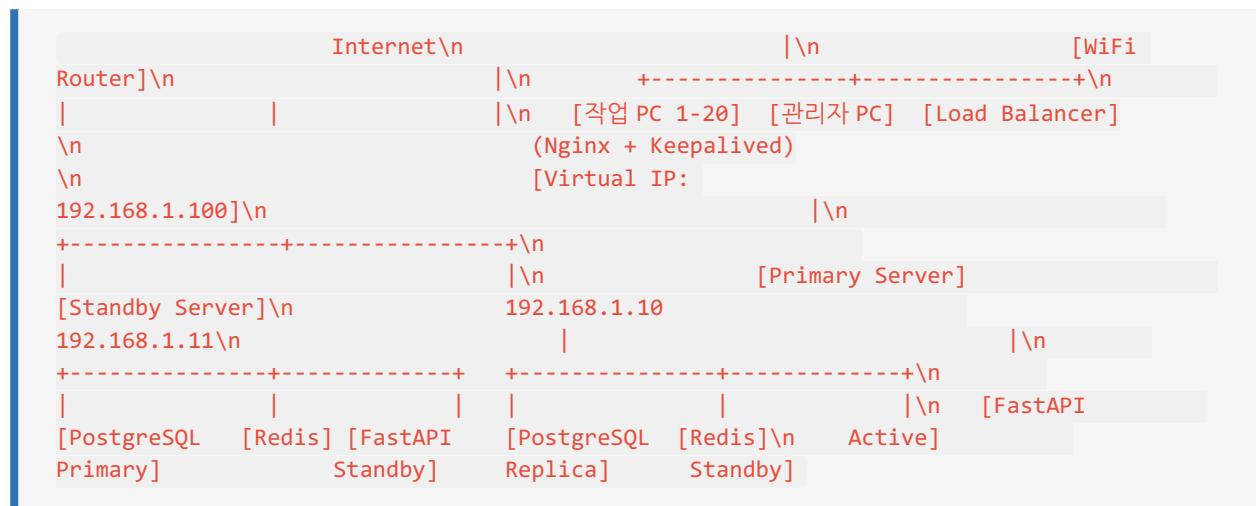
### 연결 흐름:

- 클라이언트 → Nginx (443) → SSL 종단
- Nginx → FastAPI (8000) → 8개 worker에 분산
- FastAPI → PostgreSQL (5432) → Connection Pool (50 connections)

- FastAPI → Redis (6379) → 캐시 조회/저장

### 4.3.2 Phase 2: 고가용성 구성 (6개월 후)

아키텍처:



서버 스펙 권장 (각 서버):

- CPU: 16 코어
- RAM: 64GB
- SSD: 1TB (NVMe)
- 네트워크: 10Gbps
- OS: Ubuntu 22.04 LTS

고가용성 구성:

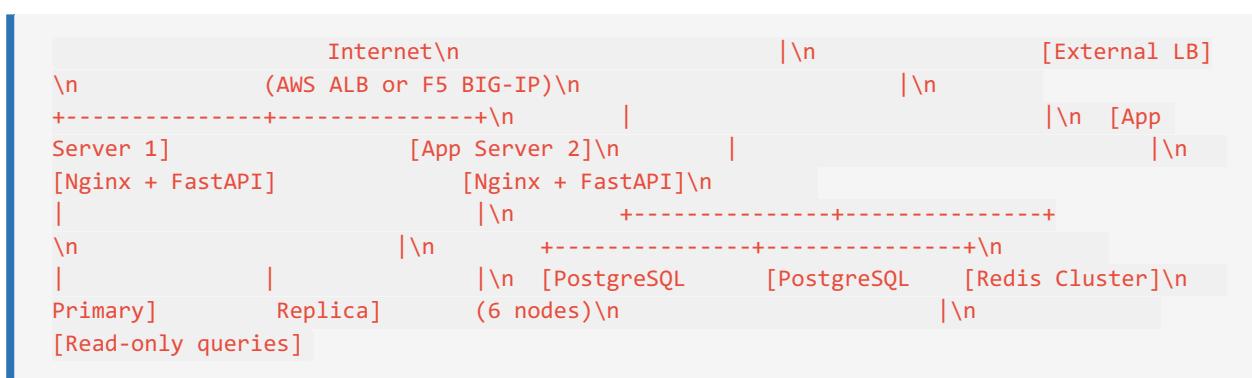
컴포넌트	Primary	Standby	Failover 방식
Nginx	Active	Active	Keepalived (VIP)
FastAPI	Active (8 workers)	Active (8 workers)	Nginx Load Balancing
PostgreSQL	Primary	Hot Standby	Streaming Replication
Redis	Master	Replica	Sentinel (자동 Failover)

### 데이터 복제:

- PostgreSQL: Streaming Replication (비동기)
- 복제 지연: < 1초
- WAL 아카이빙 활성화
- Redis: Master-Replica (비동기)
- 복제 지연: < 100ms

### 4.3.3 Phase 3: 대규모 확장 구성 (1년+ 후, 200명 이상)

#### 아키텍처:



#### 추가 요구사항:

- Message Queue: RabbitMQ 또는 Redis Streams
- CDN: CloudFlare (React Dashboard 정적 파일)

- Object Storage: MinIO (펌웨어, 백업 파일)
- Kubernetes: 컨테이너 오케스트레이션 (선택사항)

## 4.4 네트워크 구성

### 4.4.1 Phase 1: 단일 서버

구분	IP 주소	용도	포트
WiFi 공유기	192.168.1.1	Gateway	-
Primary 서버	192.168.1.10	MES 서버	80, 443, 5432, 6379
작업 PC 1-7	192.168.1.101-107	공정별	-
관리자 PC	192.168.1.201-230	대시보드 접속	-
프린터	192.168.1.50-57	라벨 프린터	9100 (TCP)

### 4.4.2 Phase 2: 고가용성

구분	IP 주소	용도	포트
Virtual IP	192.168.1.100	서비스 접속 (HA)	80, 443
Primary 서버	192.168.1.10	주 서버	80, 443, 5432, 6379
Standby 서버	192.168.1.11	대기 서버	80, 443, 5432, 6379
Monitoring 서버	192.168.1.20	Prometheus + Grafana	9090, 3000
작업 PC 1-20	192.168.1.101-120	공정별 확장	-
관리자 PC	192.168.1.201-230	대시보드 접속	-

#### 네트워크 대역폭 산정:

동시 접속자: 100명  
 평균 API 요청 크기: 5KB (Request) + 20KB (Response) = 25KB  
 요청 빈도: 10초당 1회 (0.1 req/s per user)  
 총 대역폭 = 100명 × 0.1 req/s × 25KB = 250KB/s = 2Mbps  
 피크 시간대 (×5 배수): 10Mbps  
 권장 네트워크: 1Gbps (여유율 100배)

## 4.5 AWS 클라우드 마이그레이션 전략

### 4.5.1 코드 수정 최소화 원칙

본 시스템은 온프레미스 환경에서 개발되지만, 향후 AWS로의 마이그레이션을 고려하여 다음 원칙을 따라 설계합니다:

#### 핵심 설계 패턴:

- **12 Factor App 패턴**
- 모든 환경별 설정을 환경 변수로 관리
- 하드코딩된 설정 금지 (IP, 포트, 경로 등)
- 개발/스테이징/프로덕션/AWS 환경 동일 코드 사용

- Storage Adapter 패턴

- 파일 저장소 추상화 (로컬 vs S3)
- 인터페이스 기반 설계로 구현체만 교체
- 펌웨어, 백업 파일 등에 적용

- Docker 컨테이너화

- 환경 독립적인 배포 패키지
- 온프레미스와 AWS에서 동일 이미지 사용
- 환경 변수만 변경하여 배포

**코드 수정 목표:** 0-10 라인 (환경 변수 파일만 수정)

#### 4.5.2 환경별 설정 관리

**config.py 구조:**

```
PYTHON
# backend/app/core/config.py\nimport os\nfrom typing import Optional\n\nclass Settings:\n    """환경 변수 기반 설정 관리"""\n    # 데이터베이스\n    DATABASE_URL: str = os.getenv("DATABASE_URL", "postgresql://neurohub:password@localhost:5432/neurohub")\n    # Redis 캐시\n    REDIS_URL: str = os.getenv("REDIS_URL", "redis://localhost:6379/0")\n    # 스토리지 설정\n    USE_S3: bool = os.getenv("USE_S3", "false").lower() == "true"\n    S3_BUCKET: Optional[str] = os.getenv("S3_BUCKET", None)\n    AWS_REGION: str = os.getenv("AWS_REGION", "ap-northeast-2")\n    # 로컬 파일 경로 (온프레미스 전용)\n    FIRMWARE_DIR: str = os.getenv("FIRMWARE_DIR", "/app/data/firmware")\n    BACKUP_DIR: str = os.getenv("BACKUP_DIR", "/app/data/backups")\n    # 애플리케이션\n    API_HOST: str = os.getenv("API_HOST", "0.0.0.0")\n    API_PORT: int = int(os.getenv("API_PORT", "8000"))\n    # 보안\n    SECRET_KEY: str = os.getenv("SECRET_KEY", "dev-secret-key-change-in-production")\n    JWT_ALGORITHM: str = "HS256"\n    ACCESS_TOKEN_EXPIRE_MINUTES: int = int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES", "30"))\n\nsettings = Settings()
```

**환경별 설정 파일:**

온프레미스 환경 (`.env.onprem`):

**BASH**

```
DATABASE_URL=postgresql://neurohub:password@192.168.1.10:5432/  
neurohub\nREDIS_URL=redis://192.168.1.10:6379/0\nUSE_S3=false\nFIRMWARE_DIR=/app/data/  
firmware\nBACKUP_DIR=/app/data/  
backups\nAPI_HOST=0.0.0.0\nAPI_PORT=8000\nSECRET_KEY=your-production-secret-key
```

AWS 환경 (`.env.aws`):

**BASH**

```
DATABASE_URL=postgresql://admin:password@neurohub.abc123.ap-  
northeast-2.rds.amazonaws.com:5432/neurohub\nREDIS_URL=redis://  
neurohub.abc123.apn2.cache.amazonaws.com:6379/0\nUSE_S3=true\nS3_BUCKET=neurohub-  
firmware-prod\nAWS_REGION=ap-  
northeast-2\nAPI_HOST=0.0.0.0\nAPI_PORT=8000\nSECRET_KEY=your-aws-production-secret-key
```

#### 4.5.3 Storage Adapter 구현

스토리지 추상화 인터페이스:

**PYTHON**

```

# backend/app/services/storage.py\nfrom abc import ABC, abstractmethod\nfrom pathlib\nimport Path\nfrom typing import BinaryIO\n\nclass StorageAdapter(ABC):\n    """파일 저장소\n    추상 인터페이스"""\n\n    @abstractmethod\n    def upload(self, file_path: str, key: str) -> str:\n        """파일 업로드"""\n        pass\n\n    @abstractmethod\n    def download(self, key: str, file_path: str) -> None:\n        """파일 다운로드"""\n        pass\n\n    @abstractmethod\n    def exists(self, key: str) -> bool:\n        """파일 존재 여부 확인"""\n        pass\n\n    @abstractmethod\n    def delete(self, key: str) -> None:\n        """파일 삭제"""\n        pass\n\n    @abstractmethod\n    def get_url(self, key: str) -> str:\n        """파일 URL 반환"""\n        pass\n\nclass LocalStorage(StorageAdapter):\n    """로컬 파일 시스템 저장소 (온프레미스)"""\n\n    def __init__(self, base_dir: str):\n        self.base_dir = Path(base_dir)\n\n    def upload(self, file_path: str, key: str) -> str:\n        import shutil\n        dest = self.base_dir / key\n\n        dest.parent.mkdir(parents=True, exist_ok=True)\n\n        shutil.copy(file_path, dest)\n\n        return str(dest)\n\n    def download(self, key: str, file_path: str) -> None:\n        import shutil\n        src = self.base_dir / key\n\n        shutil.copy(src, file_path)\n\n    def exists(self, key: str) -> bool:\n        return (self.base_dir / key).exists()\n\n    def delete(self, key: str) -> None:\n        (self.base_dir / key).unlink(missing_ok=True)\n\n    def get_url(self, key: str) -> str:\n        return f"/api/v1/files/{key}"\n\nclass S3Storage(StorageAdapter):\n    """AWS S3 저장소 (클라우드)"""\n\n    def __init__(self, bucket: str, region: str):\n        import boto3\n\n        self.bucket = bucket\n        self.region = region\n        self.s3 = boto3.client('s3', region_name=region)\n\n    def upload(self, file_path: str, key: str) -> str:\n        self.s3.upload_file(file_path, self.bucket, key)\n\n        return f"s3://{self.bucket}/{key}"\n\n    def download(self, key: str, file_path: str) -> None:\n        self.s3.download_file(self.bucket, key, file_path)\n\n    def exists(self, key: str) -> bool:\n        try:\n            self.s3.head_object(Bucket=self.bucket, Key=key)\n            return True\n        except:\n            return False\n\n    def delete(self, key: str) -> None:\n        self.s3.delete_object(Bucket=self.bucket, Key=key)\n\n    def get_url(self, key: str) -> str:\n        # Presigned URL (7일 유효)\n\n        return self.s3.generate_presigned_url(\n            'get_object',\n            Params={'Bucket': self.bucket, 'Key': key},\n            ExpiresIn=604800 # 7 days\n        )\n\n    # Factory: 환경에 따라 자동 선택\n    def get_storage() -> StorageAdapter:\n        """환경별 스토리지 어댑터 생성"""\n        from app.core.config import settings\n\n        if settings.USE_S3:\n            return S3Storage(settings.S3_BUCKET, settings.AWS_REGION)\n        else:\n            return LocalStorage(settings.FIRMWARE_DIR)

```

사용 예시 (펌웨어 관리):

**PYTHON**

```
# backend/app/api/v1/endpoints/firmware.py\nfrom app.services.storage import\nget_storage\n\n@router.post("/upload")\nasync def upload_firmware(file: UploadFile):\n    storage = get_storage() # 환경에 따라 자동 선택\n    # 임시 파일로 저장\n    temp_path = f"/tmp/{file.filename}"\n    with open(temp_path, "wb") as f:\n        f.write(await file.read())\n    # 스토리지에 업로드 (로컬 or S3)\n    key = f"firmware/{file.filename}"\n    storage.upload(temp_path, key)\n    return {"message": "Uploaded", "key": key}\n\n@router.get("/download/{version}")\nasync def download_firmware(version: str):\n    storage = get_storage()\n    # 다운로드 URL 생성\n    (로컬: /api/v1/files/xxx, S3: presigned URL)\n    key = f"firmware/firmware_{version}.bin"\n    url = storage.get_url(key)\n    return {"download_url": url}
```

**4.5.4 AWS 마이그레이션 옵션****Option 1: Lift & Shift (EC2 기반)**

항목	내용
**난이도**	★★★ (쉬움)
**마이그레이션 시간**	1-2일
**코드 수정**	0-10 라인 (환경 변수만)
**예상 비용**	~\$252/월
**장점**	빠른 마이그레이션, 최소 변경
**단점**	운영 부담 높음, HA 수동 구성

**AWS 구성:**

- EC2 t3.large (2개): primary + standby
- PostgreSQL: EC2에 설치
- Redis: EC2에 설치
- 로드밸런서: Nginx (EC2 내부)
- 파일 스토리지: EBS 볼륨

## 배포 방법:

### BASH

```
# 1. Docker Compose로 배포\ndocker-compose -f docker-compose.aws.yml up -d\n\n# 2. 환경 변수만 변경\nexport DATABASE_URL=postgresql://...\\nexport REDIS_URL=redis://...\\nexport USE_S3=false
```

## Option 2: Managed Services (권장)

항목	내용
**난이도**	★★★ (보통)
**마이그레이션 시간**	3-4주
**코드 수정**	0 라인 (환경 변수만)
**예상 비용**	~\$427/월
**장점**	자동 백업, 자동 HA, 모니터링 통합
**단점**	초기 설정 복잡, 비용 높음

## AWS 서비스 매핑:

온프레미스	AWS 서비스	설정
PostgreSQL 15	RDS PostgreSQL 15	db.t3.large (Multi-AZ)
Redis 7.x	ElastiCache Redis	cache.t3.medium (Cluster 모드)
Nginx	Application Load Balancer (ALB)	HTTP/HTTPS 리스너
파일 저장소	S3	Standard Storage Class
백업	S3 + RDS Snapshot	자동 백업 (7일 보관)
모니터링	CloudWatch	RDS/EC2 통합 메트릭
로그	CloudWatch Logs	중앙 집중식 로그

#### 인프라 구성 (Terraform):

```
HCL
# infrastructure/aws/main.tf
resource "aws_db_instance" "neurohub" {
  identifier      = "neurohub-postgres"
  engine          = "postgres"
  engine_version  = "15.4"
  instance_class   = "db.t3.large"
  allocated_storage = 100
  storage_type     = "gp3"
  multi_az         =
  true # 고가용성
  backup_retention_period = 7
  db_name          = "neurohub"
  username          = "admin"
  password          = var.db_password
  vpc_security_group_ids = [aws_security_group.db.id]
  nresource
  "aws_elasticache_cluster" "neurohub" {
    cluster_id      = "neurohub-redis"
    engine          = "redis"
    engine_version  = "7.0"
    node_type       =
    "cache.t3.medium"
    num_cache_nodes = 2
    parameter_group_name =
    "default.redis7"
    subnet_group_name =
    aws_elasticache_subnet_group.neurohub.name
    security_group_ids =
    [aws_security_group.cache.id]
    nresource "aws_s3_bucket" "firmware" {
      bucket =
      "neurohub-firmware-prod"
      versioning {
        enabled = true
      }
      lifecycle_rule {
        enabled = true
        expiration {
          days = 90 # 90일 후 자동 삭제
        }
      }
    }
  }
}
```

#### 4.5.5 단계별 마이그레이션 절차

##### Phase 1: 준비 단계 (1주)

- AWS 계정 설정

- AWS 계정 생성
- IAM 사용자 및 역할 설정
- VPC 및 서브넷 구성 (Private/Public)

#### • 코드 검증

- 환경 변수 기반 설정 확인
- Storage Adapter 테스트
- Docker 이미지 빌드 및 테스트

#### • 데이터 마이그레이션 계획

- PostgreSQL 덤프 생성
- 데이터 크기 산정
- 마이그레이션 다운타임 계획

### Phase 2: 인프라 구축 (1-2주)

#### • AWS 리소스 프로비저닝

```
`bash
cd infrastructure/aws
terraform init
terraform plan
terraform apply`
```

#### • 데이터베이스 마이그레이션

```
`bash
# 온프레미스에서 덤프 생성
pg_dump -h 192.168.1.10 -U neurohub neurohub > neurohub_backup.sql`
```

```
# AWS RDS로 복원
```

```
psql -h neurohub.abc123.ap-northeast-2.rds.amazonaws.com \
-U admin -d neurohub < neurohub_backup.sql`
```

- 파일 마이그레이션 (펌웨어)

```
`bash
# AWS CLI로 S3 동기화
aws s3 sync /app/data/firmware/ s3://neurohub-firmware-prod/firmware/
`
```

### Phase 3: 배포 및 전환 (3-5일)

- 애플리케이션 배포

```
`bash
# Docker 이미지 ECR에 푸시
docker tag neurohub-backend:latest 123456789.dkr.ecr.ap-northeast-2.amazonaws.com/neurohub-
backend:latest
docker push 123456789.dkr.ecr.ap-northeast-2.amazonaws.com/neurohub-backend:latest

# EC2에서 실행
docker-compose -f docker-compose.aws.yml up -d
`
```

- 환경 변수 설정

```
`bash
# .env.aws 파일 생성
DATABASE_URL=postgresql://admin:password@neurohub.abc123.ap-
northeast-2.rds.amazonaws.com:5432/neurohub
REDIS_URL=redis://neurohub.abc123.apn2.cache.amazonaws.com:6379/0
USE_S3=true
S3_BUCKET=neurohub-firmware-prod
AWS_REGION=ap-northeast-2
`
```

- DNS 전환

- Route 53에 도메인 등록
- ALB DNS를 A 레코드로 연결
- 점진적 트래픽 전환 (0% → 10% → 50% → 100%)

#### Phase 4: 검증 및 최적화 (1주)

- 성능 테스트

- API 응답 시간 측정
- 데이터베이스 쿼리 최적화
- 캐시 히트율 확인

- 모니터링 설정

- CloudWatch 대시보드 구성
- 알람 설정 (CPU, 메모리, 디스크, API 에러율)
- 로그 확인 및 분석

- 백업 검증

- RDS 자동 스냅샷 확인
- S3 버킷 버전 관리 테스트
- 복원 절차 문서화

#### 4.5.6 비용 산정

온프레미스 vs AWS 비용 비교 (월간):

구분	온프레미스	AWS (Lift & Shift)	AWS (Managed)
**서버**	~\$3,000 (초기 투자)	EC2 t3.large × 2: \$120	EC2 t3.large × 2: \$120
**데이터베이스**	포함	포함 (EC2)	RDS db.t3.large (Multi-AZ): \$240
**캐시**	포함	포함 (EC2)	ElastiCache t3.medium: \$50
**스토리지**	HDD 2TB: \$100	EBS 100GB: \$10	S3 50GB: \$1.15
**네트워크**	사내망 무료	데이터 전송: \$20	ALB: \$25 + 데이터 전송: \$20
**백업**	외장 HDD: \$50	EBS 스냅샷: \$5	포함 (RDS 자동 백업)
**인건비**	시스템 관리자 필요	부분 관리 필요	최소 관리
**총 월간 비용**	~\$150 + 인건비	~\$252	~\$427

### 3년 총 소유 비용 (TCO):

항목	온프레미스	AWS (Managed)
초기 투자	\$3,000 (서버 구매)	\$0
운영 비용 (3년)	\$5,400 (\$150×36)	\$15,372 (\$427×36)
인건비 (3년)	\$36,000 (시스템 관리자)	\$12,000 (부분 관리)
**총계**	**\$44,400**	**\$27,372**

결론: AWS Managed Services가 3년 기준 약 \$17,000 절감 (38% 저렴)

### 4.5.7 롤백 계획

AWS 마이그레이션 실패 시 온프레미스로 복귀하는 절차:

- **DNS 복원** (5분)
  - Route 53에서 온프레미스 IP로 A 레코드 변경
  - TTL이 짧으므로 5분 내 전환

- **데이터 동기화** (1-2시간)

```
`bash
```

```
# AWS RDS에서 최신 데이터 덤프
pg_dump -h neurohub.abc123.ap-northeast-2.rds.amazonaws.com \
-U admin neurohub > latest_backup.sql
```

```
# 온프레미스 DB 복원
```

```
psql -h 192.168.1.10 -U neurohub neurohub < latest_backup.sql
```

- **환경 변수 복원**

```
`bash
```

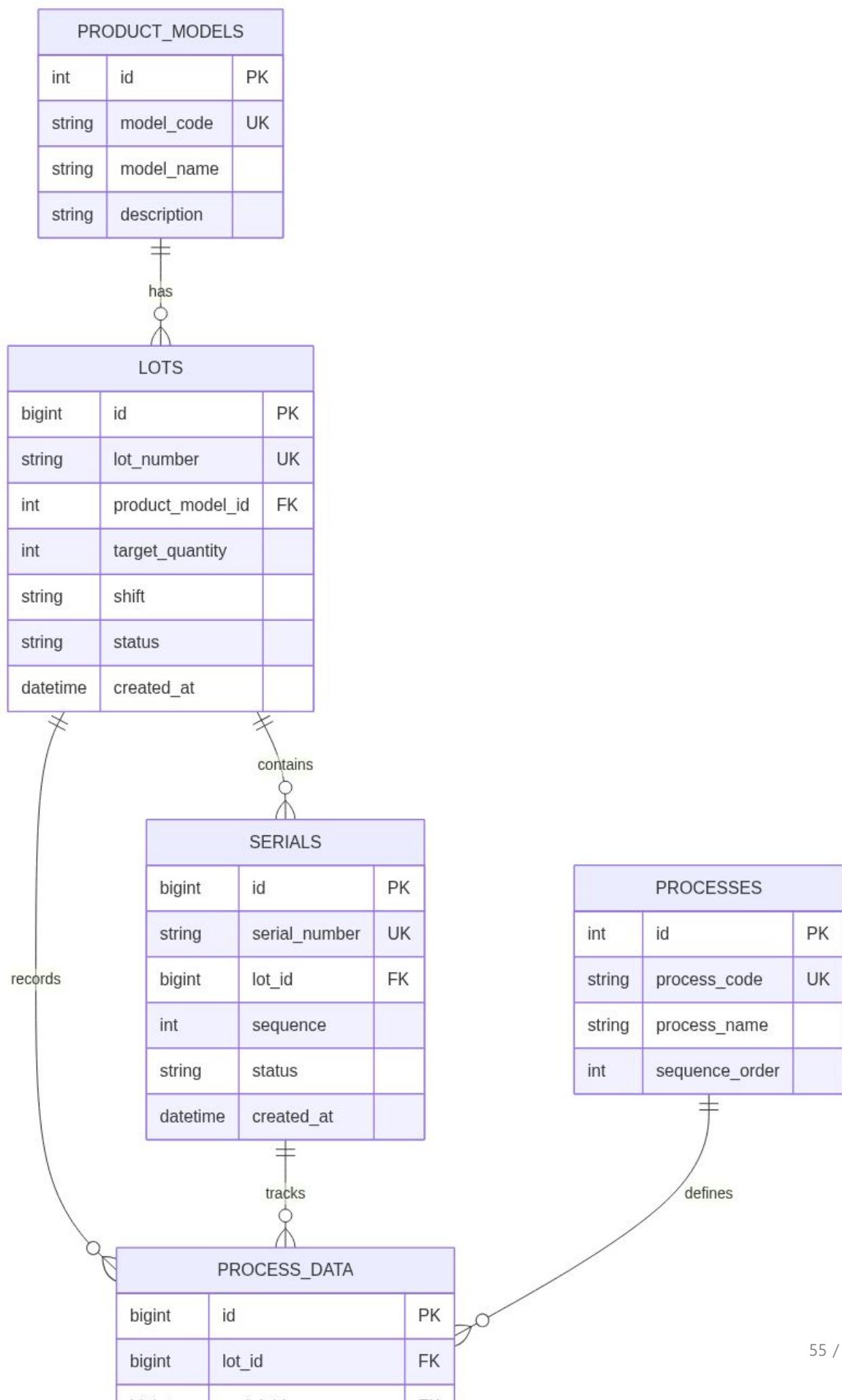
```
# .env.onprem 파일 재적용
docker-compose -f docker-compose.onprem.yml up -d
```

- **검증**
  - API 헬스체크 확인
  - 주요 기능 테스트 (LOT 생성, 공정 완공 등)

**최대 다운타임:** 2-3시간 (데이터 동기화 시간 포함)

## 5. 데이터 설계

## 5.1 핵심 엔티티 관계도 (ERD)



## 5.2 주요 테이블 설명

### 5.2.1 제품 모델 (product\_models)

용도: 제품 유형 관리

주요 필드:

- `model_code`: 제품 모델 코드 (예: NH-F2X-001)
- `model_name`: 제품명
- `description`: 설명

### 5.2.2 LOT (lots)

용도: LOT 정보 관리

주요 필드:

- `lot_number`: LOT 번호 (UK)
- `product_model_id`: 제품 모델 FK
- `target_quantity`: 목표 수량 (100)
- `shift`: 교대 (D/N)
- `status`: CREATED, IN\_PROGRESS, COMPLETED, CLOSED

### 5.2.3 시리얼 번호 (serials)

용도: 개별 제품 추적

### 주요 필드:

- `serial_number` : 시리얼 번호 (UK)
- `lot_id` : 소속 LOT FK
- `sequence` : LOT 내 순번 (1~100)
- `status` : CREATED, IN\_PROGRESS, PASSED, FAILED

### 5.2.4 공정 데이터 (process\_data)

**용도:** 공정별 작업 데이터 저장

### 주요 필드:

- `lot_id` : LOT FK
- `serial_id` : 시리얼 FK (nullable)
- `process_id` : 공정 FK
- `process_specific_data` : 공정별 데이터 (JSONB)
- `start_time` : 착공 시간
- `complete_time` : 완공 시간

### JSONB 예시:

#### JSON

```
{\n  "온도": 60.5,\n  "변위": 198.3,\n  "힘": 15.2,\n  "모선_lot": "MS-2025-100",\n  "검사\n결과": "PASS"\n}
```

## 5.3 LOT 및 시리얼 번호 체계

### 5.3.1 LOT 번호 형식

형식: {제품코드}-{공장코드}-{YYMMDD}{교대}-{일련번호}

예시: WF-KR-251110D-001

요소	설명	예시
제품코드	제품 식별 코드	WF
공장코드	생산 공장	KR
YYMMDD	생산 날짜	251110
교대	Day/Night	D
일련번호	당일 순번	001

### 5.3.2 시리얼 번호 형식

형식: {LOT번호}-{순번}

예시: WF-KR-251110D-001-0001

- LOT당 0001~0100 (100개)
- Label Printing 공정에서 순차 생성

## 6. 개발 및 투자 계획

## 6.1 개발 일정

### Phase 1: MVP (8주)

주차	작업 내용
1주	환경 설정, DB 설계, API 설계
2-3주	Backend API 개발 (LOT, 시리얼, 공정)
4-5주	Frontend App 개발 (작업 PC)
6주	Dashboard 개발 (관리자)
7주	통합 테스트
8주	현장 파일럿 및 피드백

### Phase 2: 안정화 (1개월)

- 서버 이중화 구축
- 성능 최적화
- 추가 기능 개발

## 6.2 팀 구성

역할	인원	책임
Backend 개발자	1-2명	FastAPI, DB, API
Frontend 개발자	1명	PyQt5, React
DevOps	1명	서버, 배포, 모니터링

**최소 인원:** 2-3명

**권장 인원:** 3-4명

## 6.3 투자 계획

### 6.3.1 하드웨어 (Phase 1)

항목	수량	단가 (만원)	금액 (만원)
Primary 서버	1	150	150
작업 PC	7	80	560
WiFi 공유기	1	30	30
바코드 스캐너	7	15	105
라벨 프린터	2	150	300
NAS (백업용)	1	160	160
<b>**소계**</b>			<b>**1,305**</b>

### 6.3.2 소프트웨어 및 개발

항목	금액 (만원)
개발 인건비 (2개월)	3,200
기타 SW 라이선스	100
예비비 (10%)	380
<b>**소계**</b>	<b>**3,680**</b>

### 6.3.3 총 투자 요약

**Phase 1 (필수):** 4,985만원

**Phase 2 (권장):** 900만원 (Standby 서버, 개발 1개월)

**연간 운영 비용:** 약 1,360만원/년

- 시스템 운영자 (600만원)
- 전기/인터넷 (600만원)
- 클라우드 백업 (60만원)
- 기타 소모품 (100만원)

### 6.4 ROI 분석

**기대 효과 (연간):**

항목	절감/효과	금액 (만원/년)
불량 원인 파악 시간 단축	월 20시간 절감	600
재고 관리 효율화	과잉 재고 감소	1,000
생산 계획 최적화	가동률 5% 향상	2,000
**합계**		**3,600**

회수 기간: 약 1.4년

## 7. 부록

### 7.1 용어 정의

용어	설명
**LOT**	동일 조건으로 생산된 제품 묶음 단위 (통상 100개)
**시리얼 번호**	개별 제품 고유 식별 번호 (Serial Number)
**작업대차**	LOT 단위로 제품을 이동/보관하는 이동식 작업대
**착공**	공정 작업 시작
**완공**	공정 작업 완료
**Traceability**	추적성 - 제품 이력 추적 능력
**MES**	Manufacturing Execution System - 제조 실행 시스템
**LMA**	Linear Muscle Actuator - 로봇의 핵심 구동기
**SMA 스프링**	Shape Memory Alloy 스프링 - 온도로 형상 기억하여 구동
**모선**	SMA 스프링을 제조하는 원재료 케이블
**EOL 검사**	End Of Line 검사 - 최종 성능 검사
**TOF 센서**	Time Of Flight 센서 - 거리 측정 센서
**교대조**	작업 교대 (D: Day, N: Night)
**바코드 스캔**	LOT 번호/시리얼 번호 자동 인식 방식
**File Watcher**	특정 디렉토리의 파일 생성/변경을 자동 감시하는 기능
**JSONB**	PostgreSQL의 JSON 이진 저장 타입 - 검색/인덱싱 최적화
**API**	Application Programming Interface - 시스템 간 데이터 통신 인터페이스
**JWT**	JSON Web Token - 사용자 인증용 토큰
**RBAC**	Role-Based Access Control - 역할 기반 접근 제어
**HA**	High Availability - 고가용성 (시스템 이중화)
**VIP**	Virtual IP - 가상 IP 주소 (이중화 환경용)
**Connection Pool**	데이터베이스 연결 재사용 메커니즘 (성능 최적화)

용어	설명
**WebSocket**	실시간 양방향 통신 프로토콜
**TPS**	Transactions Per Second - 초당 트랜잭션 처리량
**TTL**	Time To Live - 데이터 유효 시간
**Presigned URL**	임시 접근 권한이 부여된 URL (S3 다운로드용)
**Failover**	주 시스템 장애 시 대기 시스템으로 자동 전환
**MD5 Hash**	파일 무결성 검증용 체크섬
**Streaming Replication**	PostgreSQL 실시간 데이터 복제 방식

## 7.2 문서 버전 이력

버전	날짜	변경 내용	작성자
v1.0	2025.11.09	초기 버전	-
v1.6	2025.11.09	투자 목록 추가	-
**v2.0**	**2025.11.10**	**전면 재작성 - 사양서 중심**	**Claude**

## 7.3 참고 자료

- FastAPI 공식 문서: <https://fastapi.tiangolo.com>
- PostgreSQL 공식 문서: <https://www.postgresql.org/docs/>
- React 공식 문서: <https://react.dev>

문서 끝