바이브 코딩 실전 아키텍처

프런트엔드 · 백엔드 · 저장소

왜 나누었고, 무엇으로 만들고, 어떻게 한 번에 띄우는가

Streamlit TypeScript FastAPI Go n8n PostgreSQL InfluxDB MongoDB Prometheus Grafana Docker Compose

AI 애플리케이션 개발 과정 · 보충 강의

오늘의 흐름

다섯 단계로 따라가는 서비스 구조

01	**배경과 역사**	한 덩어리였던 웹은 왜 둘로 나뉘었나
02	**구성도**	Presentation · Domain Logic · Storage, 세 층의 책임
03	**구성 환경**	Streamlit · TypeScript / FastAPI · Go · n8n / PostgreSQL · InfluxDB · MongoDB
04	**모니터링**	Prometheus · InfluxDB · Grafana로 메트릭 수집과 시각화
05	**배포**	Docker Compose 한 파일로 전체 서비스 띄우기
06	**협업**	계약 · 규칙 파일 · Git 흐름으로 여럿이 함께 바이브 코딩

분리 이전: 한 파일에 모든 것이 섞여 있던 시대

서버가 HTML을 통째로 생성
CGI · PHP · ASP · JSP — 1990년대 ~ 2000년대 초

화면 · 규칙 · DB 접근이 한 파일에
디자이너와 개발자가 같은 파일을 고치며 충돌

클릭 한 번 = 페이지 전체 새로고침
작은 변화에도 HTML 전체를 다시 받아 그림

새 클라이언트 = 로직 재작성
모바일 앱이 생기면 같은 규칙을 또 구현

전형적인 PHP 페이지 (요약)

```php
<?php
$db = mysqli_connect("localhost", "root", "pw");
$rows = mysqli_query($db,
  "SELECT * FROM orders WHERE user=" . $_GET['u']);
?>
<table>
<?php while ($r = mysqli_fetch_assoc($rows)) { ?>
  <tr><td><?= $r['item'] ?></td>
      <td><?= $r['price'] * 1.1 ?></td></tr>
<?php } ?>
</table>
```

주황 = DB 접근 · 청록 = 업무 규칙(부가세 10%) · 나머지 = 화면
문자열을 이어 붙인 쿼리는 SQL 인젝션 위험까지 안고 있다

전환점: 무엇이 분리를 이끌었나

2000
REST
로이 필딩의 박사 논문 — 자원은 URL로, 동작은 HTTP 메서드로

2005
AJAX
페이지 이동 없이 데이터만 주고받는 방식에 이름이 붙음

2007
스마트폰
브라우저 밖 클라이언트 폭증 — 앱도 같은 데이터를 요청

2009
Node.js
JavaScript로 서버 작성 — JSON API가 표준처럼 확산

2010–14
SPA 프레임워크
AngularJS · React · Vue — 화면은 브라우저가 직접 그림

2013–14
Docker · Compose
층마다 컨테이너로 묶어 함께 실행하는 길이 열림

결과: 화면을 그리는 쪽(프런트엔드)과 데이터 · 규칙을 다루는 쪽(백엔드)이 **HTTP + JSON이라는 계약(API)**으로만 대화하게 되었다.

분리 이후: 무엇이 나아졌나

역할 분리
화면 담당과 로직 담당이 API 명세만 합의하고 동시에 개발

클라이언트 재사용
웹 · 모바일 · n8n · 다른 서버가 같은 API를 호출

독립 배포
화면 문구만 바꿔도 백엔드는 그대로 — 배포 단위가 작아짐

독립 확장
요청이 몰리는 백엔드만 컨테이너를 늘림

기술 선택의 자유
화면은 TypeScript, AI 로직은 Python, 수집기는 Go

대가도 있다
네트워크 왕복 · CORS · API 버전 관리 · 배포 대상 증가

그래서 **구성도(누가 무엇을 하나)**와 **배포 도구(어떻게 함께 띄우나)**가 필요해졌다.

02 · 구성도

세 개의 층, 두 개의 계약

Presentation Layer
프런트엔드

- 화면 그리기 · 입력 받기
- 입력 형식 1차 검사
- 결과 · 로딩 · 오류 표시

예: Streamlit, React(TS)

HTTP
JSON

Domain Logic
백엔드

- 업무 규칙 · 계산 · 권한
- 입력 재검증 · 트랜잭션
- LLM · 외부 API 호출

예: FastAPI, Go, n8n

SQL
드라이버

Storage
저장소

- 데이터 영속화
- 데이터 모양별 DB 선택
- 백업 · 보존 정책

예: PostgreSQL, InfluxDB, MongoDB

규칙 하나: 프런트엔드는 DB를 직접 만지지 않는다. 모든 데이터는 백엔드 API를 지난다.

02 · 구성도

층별 책임: 해야 할 일과 하지 말아야 할 일

층	해야 할 일	하지 말아야 할 일	AI에게 줄 한 줄
프런트엔드	화면 상태 관리, 입력 형식 검사, 로딩 · 오류 표시	DB 접속 정보 보관, 가격 · 권한 계산, API 키 노출	“DB에는 접근하지 말고 백엔드 API만 호출해”
백엔드	업무 규칙, 권한 확인, 입력 재검증, 트랜잭션, LLM 호출	화면 문구 · 레이아웃 결정, 프런트엔드 검증만 믿기	“엔드포인트별 요청 · 응답 스키마를 Pydantic으로 먼저 정의해”
저장소	제약조건으로 정합성 보장, 인덱스, 백업	업무 로직을 저장 프로시저에 몰아넣기, 비밀번호 평문 저장	“스키마와 마이그레이션 파일을 분리하고 초기 데이터를 넣어줘”

03 · 구성 환경 — 프런트엔드

Python Streamlit, 또는 TypeScript

Streamlit · Python

- Python 한 언어로 화면까지 — 백엔드 개발자가 바로 UI 작성
- 상호작용마다 스크립트를 위에서 아래로 다시 실행하는 단순한 모델
- 적합: 사내 대시보드 · AI 데모 · PoC / 한계: 세밀한 UI, 대규모 공개 서비스

```
import requests
import streamlit as st

q = st.text_input("질문")
if st.button("보내기"):
    r = requests.post(f"{API}/ask", json={"q": q})
    st.write(r.json()["answer"])
```

React · Next.js · TypeScript

- React + Vite(SPA) 또는 Next.js(서버 렌더링 포함)
- 컴포넌트 · 상태로 복잡한 화면을 쪼개고, 타입으로 API 응답 구조를 검사
- 적합: 대외 서비스 · 복잡한 UI / 한계: 빌드 도구 · 학습량

```
const res = await fetch(`${API}/ask`, {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ q }),
});
const { answer } = await res.json();
setAnswer(answer);
```

백엔드: 로직을 무엇으로 만들까

FastAPI

Python

강점 타입 힌트 → Pydantic 자동 검증, /docs에 OpenAPI 문서 자동 생성, async 지원

적합 AI · RAG · 데이터 처리 API — LangChain, LiteLLM 등 Python 생태계와 직결

주의 CPU를 오래 쓰는 작업은 워커 · 큐로 분리

Go

Gin · Echo · 표준 net/http

강점 단일 실행 파일 배포, 고루틴 기반 높은 동시성, 적은 메모리

적합 API 게이트웨이, 센서 · 로그 수집기, 트래픽 많은 API

주의 AI 라이브러리는 Python보다 적음 → AI 처리는 Python 서비스에 맡기기

n8n

AI 워크플로우

강점 노드 연결로 LLM · RAG · SaaS 연동, Webhook 노드로 API처럼 호출

적합 자주 바뀌는 AI 흐름, 알림 · 연동 자동화, 비개발자와 협업

주의 권한 · 결제 같은 핵심 로직은 코드 백엔드에 두기

조합이 일반적이다: **FastAPI가 관문**이 되고, 자주 바뀌는 AI 흐름은 n8n에, 고성능 수집은 Go에 맡긴다.

백엔드 로직 안에서 n8n AI 워크플로우 쓰기

① **프런트엔드** · POST /ask 요청

⬇

② **FastAPI** · 인증 · 입력 검증 · 비용 한도

⬇

③ **n8n Webhook 노드** · 요청 수신

⬇

④ **AI Agent 노드** · LLM + RAG + 도구

⬇

⑤ **Respond to Webhook** → FastAPI 저장 · 응답

```
@app.post("/ask")
async def ask(req: AskReq, user=Depends(auth)):
    async with httpx.AsyncClient(timeout=60) as c:
        r = await c.post(f"{N8N_URL}/webhook/ask",
            json={"q": req.q, "user": user.id})
    r.raise_for_status()
    result = r.json()
    await save_log(user.id, req.q, result)
    return result
```

왜 백엔드를 거칠까? n8n 주소와 LLM 키를 숨기고, 권한 · 비용 한도 · 로그를 한 곳에서 관리하기 위해서.

참고: 편집기 테스트 주소는 /webhook-test/…, 운영 주소 /webhook/…는 워크플로우를 활성화해야 동작

데이터의 모양과 쓰임이 다르면, 맞는 저장소도 다르다

DB 하나로 모든 것을 해결하려 하면 어딘가에서 성능이나 유연성을 잃는다 — 이를 폴리글랏 퍼시스턴스(Polyglot Persistence)라 부른다.

구조가 고정된 레코드

RDB · PostgreSQL

회원 · 설비 · 주문처럼 관계가 있고, 정확한 수정(CRUD)과 트랜잭션이 중요

시간 + 숫자

시계열 DB · InfluxDB

센서 · 메트릭처럼 초 단위로 계속 쌓이고, 기간별 평균 · 최대로 조회

자주 바뀌는 구조

NoSQL 문서 · MongoDB

설정 · 메타데이터처럼 키가 수시로 추가 · 변경되는 JSON 문서

세 저장소 비교: 무엇을, 왜

구분	PostgreSQL · RDB	InfluxDB · 시계열	MongoDB · 문서 NoSQL
데이터 모양	고정 스키마 테이블 + 관계(외래 키)	시간 + 태그 + 정수 · 실수 필드	JSON 문서, 문서마다 키가 달라도 됨
주된 작업	레코드 CRUD · JOIN · 트랜잭션	계속 추가(append) · 기간 집계	문서 단위 읽기 · 쓰기, 중첩 구조 저장
왜 이 DB인가	ACID 트랜잭션 · 제약조건으로 정합성	시간순 압축, 보존 정책으로 자동 삭제	ALTER 없이 새 키를 바로 추가
예시	설비 마스터 · 사용자 · 작업 지시	센서 온도 · 진동, API 응답 시간	장비별 임계값, 프롬프트 템플릿
주의	초당 수천 건 시계열 적재엔 부담	행 수정 · JOIN 같은 관계 연산에 약함	스키마 검증 책임이 앱으로 이동

설비 “press-01”을 세 저장소에 담으면

PostgreSQL · 설비 마스터

```
CREATE TABLE equipment (
  id   SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  line TEXT NOT NULL
);
INSERT INTO equipment
  (name, line)
VALUES ('press-01', 'A');
```

스키마가 먼저 — 규칙에 어긋난 값은 DB가 거절한다(NOT NULL)

InfluxDB · 센서 값 (Line Protocol)

```
# 측정이름,태그 필드 [타임스탬프]
machine,eq=press-01 temp=71.3,vib=0.42
machine,eq=press-01 temp=71.9,vib=0.45
machine,eq=press-01 temp=72.4,vib=0.51

# 최근 1시간, 5분 평균 (Flux 일부)
▷ range(start: -1h)
▷ aggregateWindow(every: 5m, fn: mean)
```

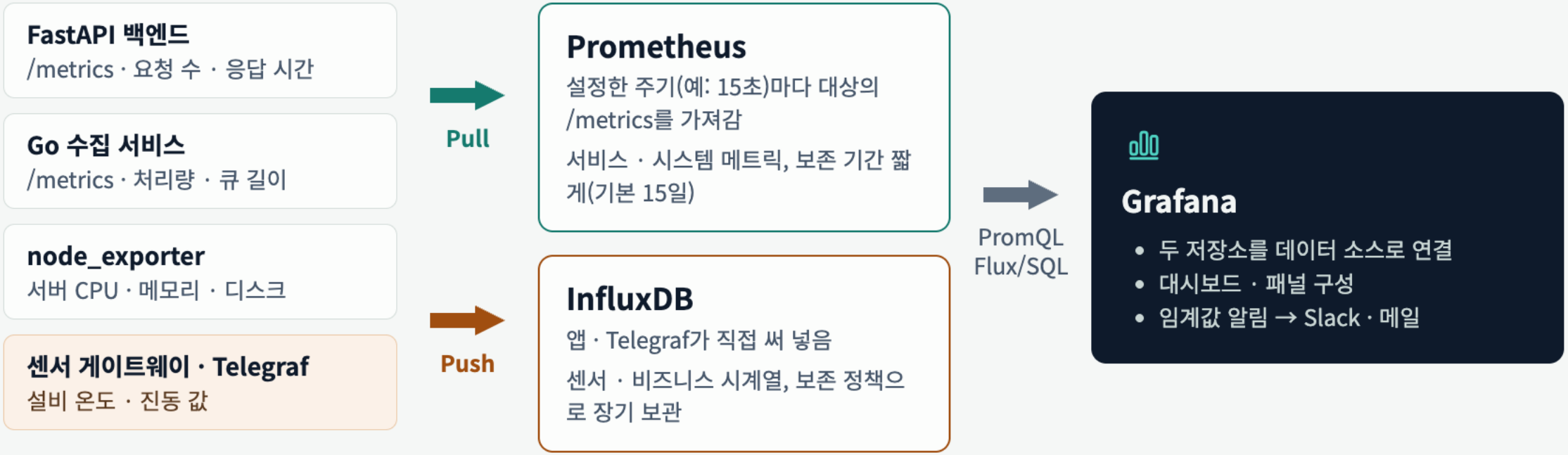
한 줄 = 한 시점의 측정값. 타임스탬프를 생략하면 서버 수신 시각이 들어간다

MongoDB · 장비 설정

```
db.config.insertOne({
  eq: "press-01",
  threshold: { temp: 85 },
  model: "lstm-v3",
  alerts: ["slack", "sms"]
})
```

문서마다 구조가 달라도 됨 — 새 키(alerts)는 그냥 넣으면 끝

수집 → 저장 → 시각화



Prometheus는 **"서버가 아픈가"**, InfluxDB는 **"설비가 아픈가"**를 본다 — Grafana는 둘을 한 화면에.

어떤 메트릭을 구할까: RED와 USE

RED · 서비스(API) 관점

Rate — 초당 요청 수
Errors — 오류 비율
Duration — 응답 시간(p95 · p99)

USE · 자원(서버) 관점

Utilization — CPU · 메모리 사용률
Saturation — 포화(대기열 · 부하)
Errors — 디스크 · 네트워크 오류

```
# 초당 요청 수 (최근 5분 기준)
sum(rate(http_requests_total[5m]))

# 5xx 오류 비율
sum(rate(http_requests_total{status=~"5.."}[5m]))
  / sum(rate(http_requests_total[5m]))

# p95 응답 시간(초)
histogram_quantile(0.95, sum by (le)
  (rate(http_request_duration_seconds_bucket[5m])))
```

FastAPI에 prometheus-fastapi-instrumentator를 붙이면 /metrics가 열리고 위 지표들이 자동으로 생긴다.

왜 Docker Compose인가

직접 하나씩 실행하면

- 서비스 8개를 각각 설치 · 실행
- 포트 · 환경변수 · 버전을 사람이 기억
- “내 PC에선 되는데 서버에선 안 돼요”

compose.yaml 하나로

- 서비스 · 네트워크 · 볼륨을 한 파일에 선언
- 노트북 · 사내 서버 어디서나 같은 결과
- 서비스 이름으로 서로 통신(내부 DNS)

`docker compose up -d`	전체를 백그라운드로 기동
`docker compose ps`	상태 · 헬스체크 확인
`docker compose logs -f backend`	백엔드 로그 따라가기
`docker compose up -d --build backend`	코드 수정 후 백엔드만 다시 빌드
`docker compose down`	중지 · 삭제 (볼륨 데이터는 유지)
`docker compose down -v`	**주의:** 볼륨까지 삭제 — 데이터 소멸

앱 서비스: 프런트엔드 · 백엔드 · n8n

```yaml
services:
  frontend:
    build: ./frontend          # Streamlit
    ports: ["8501:8501"]
    environment:
      API_URL: http://backend:8000
    depends_on: [backend]
  backend:
    build: ./backend           # FastAPI
    env_file: .env
    environment:
      N8N_URL: http://n8n:5678
    depends_on:
      postgres: { condition: service_healthy }
  n8n:
    image: n8nio/n8n
    ports: ["5678:5678"]
    volumes: [n8ndata:/home/node/.n8n]
```

① 서비스 이름 = 주소

컨테이너끼리는 localhost가 아니라 http://backend:8000처럼 이름으로 호출

② build와 image

내가 만든 코드는 build(Dockerfile), 남이 만든 소프트웨어는 image

③ depends_on + condition

DB 헬스체크가 통과한 뒤에야 백엔드가 시작

④ ports = 외부 공개

브라우저가 직접 들어오는 Streamlit · n8n 편집기만 연다

저장소와 모니터링 서비스

```
  postgres:
    image: postgres:17
    environment: { POSTGRES_PASSWORD: "${PG_PASSWORD}" }
    volumes: [pgdata:/var/lib/postgresql/data]
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
  mongo:    { image: mongo:8, volumes: [mongodata:/data/db] }
  influxdb:
    image: influxdb:2
    volumes: [influxdata:/var/lib/influxdb2]
  prometheus:
    image: prom/prometheus
    volumes: [./prometheus.yml:/etc/prometheus/prometheus.yml]
  grafana:
    image: grafana/grafana
    ports: ["3000:3000"]
volumes: {pgdata: {}, mongodata: {}, influxdata: {}, n8ndata: {}}
```

named volume

컨테이너를 지워도 데이터 유지 — down -v는 볼륨까지 삭제

${PG_PASSWORD}

값은 .env 파일에, .env는 .gitignore에

healthcheck

pg_isready 성공 = “DB 준비 완료” 신호

```
# prometheus.yml
scrape_configs:
  - job_name: backend
    static_configs:
      - targets: ["backend:8000"]
```

05 · 배포

바이브 코딩에서 자주 막히는 지점

증상	원인	해결
프런트엔드 → 백엔드 연결 거부	컨테이너 안의 localhost는 자기 자신	서비스 이름 사용: http://backend:8000
React 화면에서만 backend:8000 실패	브라우저는 도커 내부 이름을 모름	공개 포트 또는 리버스 프록시 주소 사용
백엔드가 뜨자마자 DB 연결 실패	DB가 준비되기 전에 백엔드 시작	healthcheck + service_healthy, 재시도
재시작했더니 데이터가 사라짐	볼륨 미선언, 또는 down -v 실행	named volume 선언, -v는 신중하게
브라우저 콘솔에 CORS 오류	다른 출처(포트)의 API 호출	FastAPI CORSMiddleware에 허용 출처 명시
API 키가 Git에 올라감	compose.yaml에 비밀값 직접 작성	.env로 분리 + .gitignore 등록

각자 AI에게 시키면 생기는 일

이름 불일치

프런트엔드는 userName, 백엔드는 user_name을 주고받는다

테이블 중복

같은 개념인데 users와 members 테이블이 따로 생긴다

환경 불일치

포트 8000과 8080, requests와 httpx가 섞인다

남의 파일 수정

AI가 "고치는 김에" 다른 사람 폴더까지 바꾼다

AI는 빠르지만, 각자 다른 방향으로 빠르다.

해법은 하나 — **코드보다 계약을 먼저 합의**하고, 모든 팀원의 AI가 같은 규칙과 같은 계약을 읽게 한다.

역할은 층을 따라, 소유는 폴더 단위로

역할	소유 폴더	책임	지키는 계약
프런트엔드	frontend/	Streamlit 화면, API 연동, 오류 표시	openapi.yaml (사용하는 쪽)
백엔드	backend/	FastAPI 엔드포인트, 업무 규칙, 권한	openapi.yaml (만드는 쪽)
데이터 · 인프라	contracts/db/, monitoring/, compose.yaml	스키마 · 마이그레이션, 배포, 대시보드	DB 스키마, compose.yaml
AI 워크플로우	workflows/	n8n 흐름, 프롬프트, RAG 문서	Webhook 입출력 JSON

통합 담당(팀장)

계약 변경 승인과 main 머지를 책임진다. 역할과 겸해도 된다.

1인 개발이라면

네 개의 모자를 순서대로 쓴다 — AI 세션도 역할별로 나눈다.

계약 우선: 하나의 저장소, 하나의 진실

```
project/
├─ AGENTS.md            # AI 공통 규칙
├─ docs/
│  ├─ architecture.md   # 오늘의 구성도
│  └─ adr/              # 결정 기록
├─ contracts/
│  ├─ openapi.yaml      # FE ↔ BE
│  └─ db/migrations/    # BE ↔ DB
├─ frontend/            # Streamlit
├─ backend/             # FastAPI
├─ workflows/           # n8n JSON export
├─ monitoring/          # prometheus.yml
├─ compose.yaml
└─ .env.example         # 키 이름만, 값은 X
```

openapi.yaml — 화면과 로직의 계약

엔드포인트와 요청 · 응답 필드. FastAPI가 만든 /openapi.json을 내보내 커밋해도 된다

db/migrations — 로직과 저장소의 계약

테이블 변경은 Alembic 같은 마이그레이션 파일로만

compose.yaml + .env.example

서비스 이름 · 포트 · 환경변수 키의 단일 기준

docs/adr — 왜 그렇게 정했나

짧은 결정 기록은 사람에게도, AI에게도 맥락이 된다

모든 팀원의 AI가 같은 규칙을 읽게

```
# AGENTS.md — 모든 AI 공통 규칙

## 구조
- frontend는 backend API만 호출한다 (DB 접근 금지)
- API를 바꾸려면 contracts/openapi.yaml부터 수정
- DB 변경은 마이그레이션 파일로만
## 저장소
- 마스터 PostgreSQL · 센서 InfluxDB · 설정 MongoDB
## 금지
- 비밀값 하드코딩 금지 → .env에서 읽기
- 작업 범위로 지정된 폴더 밖 파일 수정 금지
## 완료 조건
- docker compose up 후 /health 200
- 테스트 통과, 변경한 계약은 PR 본문에 명시
```

원본은 하나

도구마다 읽는 파일이 다르다. Codex는 AGENTS.md, Claude Code는 CLAUDE.md — CLAUDE.md에 @AGENTS.md 한 줄로 원본을 불러온다

실수는 규칙으로 쌓는다

AI가 같은 실수를 반복하면(예: 화면 코드에서 또 DB 직접 접속) 프롬프트로 매번 말하지 말고 AGENTS.md에 금지 규칙을 한 줄 적는다. 팀 전체 AI가 읽는 파일이므로 혼자 고치지 않고 PR로 올려 함께 확인한다

첫 프롬프트 습관

“AGENTS.md와 contracts/를 먼저 읽고, backend/ 안에서만 작업해”

작업 한 건의 흐름: 이슈에서 머지까지

① 이슈	② 브랜치	③ AI 구현	④ 로컬 검증	⑤ PR	⑥ 리뷰 · CI	⑦ 머지
작업 단위 하나	feat/be-ask	규칙 · 계약 참조	compose up · 테스트	템플릿 작성	사람 1명 + AI	main은 보호

PR 템플릿 예시

```
## 무엇을
- /ask 엔드포인트에 사용자별 비용 한도 추가
## 계약 변경
- [x] openapi.yaml  [ ] DB  [ ] compose
## AI 사용
- 도구 · 핵심 프롬프트 요약
## 확인
- [x] compose up 후 /health 200
- [x] pytest 통과
```

main 직접 push 금지 — 브랜치 보호 규칙으로 막는다

PR은 작게 — 하루 안에 리뷰할 수 있는 크기

설명할 수 있어야 머지 — AI가 쓴 코드도 작성자가 리뷰어에게 설명

CI 자동 실행 — PR마다 compose 빌드 + 테스트

CI/CD: 검사와 배포는 사람 대신 기계가

① **PR 생성** — push하면 시작 → ② **CI** — 빌드 · 테스트 → ③ **머지** — 통과 + 리뷰 후 → ④ **CD** — 서버에 배포 → ⑤ **관측** — Grafana · 롤백

CI — PR마다 자동 검사 (GitHub Actions 예시)

```yaml
# .github/workflows/ci.yml
on: [pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: echo "PG_PASSWORD=${{ secrets.PG_PASSWORD }}" > .env
      - run: docker compose up -d --build --wait
      - run: docker compose exec -T backend pytest
      - run: docker compose down -v
```

CD — main에 들어가면 배포

이미지를 빌드해 레지스트리에 올리고(태그 = 커밋 해시), 서버에서 compose pull 후 up -d

배포 뒤엔 관측

Grafana에서 오류율 · p95 확인, 이상하면 이전 태그로 롤백

시작은 CI부터

자동 테스트 없는 자동 배포는 위험하다. 사내 Gitea Actions도 거의 같은 문법

06 · 여럿이 함께 바이브 코딩

계약이 있으면 기다리지 않고 동시에 만든다

역할	Day 0 · 함께	구현 · 각자 병렬	매일 통합
프런트엔드	화면 목록, 필요한 API 정리	mock 서버로 화면 개발	실제 백엔드 주소로 전환
백엔드	openapi.yaml 초안	엔드포인트 구현, 계약 테스트	main 머지, /docs 확인
데이터 · 인프라	스키마, compose 골격	마이그레이션, 시드 데이터, 대시보드	compose up 통합 점검
AI 워크플로우	Webhook 입출력 정의	n8n 흐름, 프롬프트, RAG 문서	워크플로우 JSON export 커밋

백엔드 없이 화면부터 — mock 서버

```
npx @stoplight/prism-cli mock contracts/openapi.yaml
```

계약 변경 절차

제안 PR → 영향 담당자 승인 → 계약 먼저 머지 → 각자 구현 수정

정리

AI에게 구조를 먼저 말하면, 코드가 덜 흔들린다

1 **층을 나눈다**
화면 · 규칙 · 저장을 섞지 않는다

2 **데이터 모양으로 저장소를 고른다**
레코드 · 시계열 · 문서

3 **처음부터 관측하고, 한 파일로 띄운다**
/metrics + compose.yaml

프롬프트 예시

"프런트엔드는 Streamlit, 백엔드는 FastAPI로 분리해 줘. 프런트엔드는 백엔드 API만 호출한다. 설비 정보는 PostgreSQL, 센서 값은 InfluxDB, 장비 설정은 MongoDB에 저장한다. 백엔드는 /metrics를 노출하고, Prometheus · Grafana까지 포함한 compose.yaml을 만들어 줘. 비밀값은 .env로 분리해 줘."