「FastAPI로 그게 됩니까?」— 됩니다

엔터프라이즈 백엔드를 지탱하는 모듈러 모놀리스 1년 회고

윤상현 · 예이린 사회적협동조합 CTO · (주)예이린 대표이사

발표자 소개

윤상현 — 예이린 사회적협동조합 CTO · (주)예이린 대표이사

예이린

(주)예이린
IT & TECHNOLOGY

기술로 돌봄을 새롭게, 우리의 내일을 따뜻하게 — 아동 돌봄 Tech-Non Profit
보건복지부·과학기술정보통신부 소관 사회적협동조합의 기술 전반을 총괄합니다

오늘은 성공담이 아니라 회고입니다

잘한 결정과 잘못 판 구덩이를 같은 비중으로 담아 왔습니다
모든 수치·코드·삽질은 실제 프로덕션 코드베이스와 git log에서 실측했습니다

이 발표가 마주하려는 통념 두 가지

이 발표가 마주하려는 통념 두 가지

FastAPI

통념 ① — '빠르게 API 하나' 만드는 도구

이름 그대로 **Fast한 API 서빙**을 내세워 출시되었고,
커뮤니티의 시선도 오랫동안 비슷했습니다.

빠른 프로토타이핑 · 마이크로서비스의 한 조각 ·
"가볍게 쓰기 좋은 도구"

spring NestJS

통념 ② — 대규모 시스템의 정설

수십 개의 도메인, 여러 계층, DI 컨테이너 —
엔터프라이즈 설계의 어휘는 JVM과 TS 진영의 것이라는
정설이 자리 잡고 있었습니다.
저 역시 그 정설 위에서 NestJS를 골랐던 사람입니다

오늘의 질문 — **이 두 통념은 2026년에도 유효할까요?**
저희는 프로덕션에서 돌아가는 **105,594줄의 실측**으로 답해보려 합니다.

오늘의 여정

01 우리가 있던 곳

1인이 운영한 폴리글랏 MSA

레거시 전체 지도

프론트엔드 4개 (Vercel) yeirin-admin · yeirin-guardian · soul-e-frontend · myfriend-console

yeirin-backend
NestJS 11 + TypeORM
EC2 #1 · :3000 · PM2
47,936줄 · yarn berry

soul-e
FastAPI · Py 3.12
EC2 #2 · :8000/:8002 · PM2
35,973줄 · uv

yeirin-ai
FastAPI · Py 3.11
EC2 #3 · :8001 · PM2
15,879줄 · uv

myfriend
FastAPI · Py 3.13
ECS+ECR · 5유닛
32,675줄

DB 4개
PG 15 · 16 혼재
yeirin_dev / soul_e /
soul_e_bif / myfriend_dev

레포 6개 · 배포 유닛 9개 · CI 5벌 · 패키지 매니저 3종 · Python 3개 버전 + TS — 그리고 개발자 1명

레포 6개 · 배포 유닛 9개 · DB 4개 · 패키지 매니저 3종 — 그리고 개발자 1명이었습니다

잠깐, MSA가 뭐였죠? (입문)

마이크로서비스 아키텍처(MSA)

기능별로 서비스를 쪼개 각자 배포·확장하는 구조 — 큰 조직에선 팀 경계와 일치시켜 위력을 냅니다

우리의 현실

· 서비스는 쪼개졌지만 팀은 1명 — 경계마다 운영 비용만 치르고 있었습니다

· Auto Scaling 미적용 — 마이크로서비스의 비용은 다 내고, 탄력성 이점은 0

· 서비스끼리 HTTP로 대화하면서, 몰래 서로의 DB도 읽고 있었습니다

2GiB 서버에서 런타임 3개가 유휴 메모리 660MB 점유 — '프로세스 3개로 존재하는 비용'

어느 금요일 저녁의 체크리스트

2026년 2월의 어느 금요일, 18:00

"배포 하나 나가려면"

- ☐ EC2 3대에 각각 SSH 접속
- ☐ PM2 앱 4벌 + ECS 5유닛 상태 확인
- ☐ 기동 순서 암기: postgres → ai → backend → soul-e
- ☐ .env 키 121개 × 4서비스, 어긋난 것 없는지
- ☐ 내부 시크릿 헤더 3종 아직 유효한지
- ☐ t3.small에서 yarn build — 메모리 버텨줄지
- ☐ 단일 인스턴스 — 빌드하는 동안 **다운타임**
- ☐ 롤백 플랜: **없음** · CI 테스트: **없음**

진짜 리스크는 다운타임이 아니었습니다 — 이 체크리스트 전체를 아는 사람이 조직에 저 하나뿐이라는 사실이었습니다

서비스간 결합의 실체

서비스 사이 배관 코드 ≈ 3,185줄 · 내부 인증 헤더 3종이 섞여 있었습니다

가장 무서운 결합 — 남의 DB를 읽는다

soul-e-backend/src/soul_e/infrastructure/yeirin_db/models.py (727줄 중 발췌) — MSA 최악의 결합

```python
class ChildProfileReadModel(Base):
    """Read-only model for child_profiles (아동 프로필).

    NOTE: guardianId 제거됨. 모든 아동은 Institution에 직접 연결됩니다.
    """

    __tablename__ = "child_profiles"      # ← 남의 서비스(NestJS)의 테이블을

    id: Mapped[UUID] = mapped_column(PGUUID(as_uuid=True), primary_key=True)
    name: Mapped[str] = mapped_column(String(30))
    birthDate: Mapped[date] = mapped_column(Date)          # ← camelCase 컬럼명까지
    gender: Mapped[Gender] = mapped_column(                #    TypeORM 관례를 미러링
        Enum(Gender, name="child_profiles_gender_enum")    # ← PG enum 타입 이름까지
    )                                                      #    손으로 복사
    childType: Mapped[ChildType] = mapped_column(
        Enum(ChildType, name="child_profiles_childtype_enum")
    )
    careFacilityId: Mapped[UUID | None] = mapped_column(
        PGUUID(as_uuid=True),
        ForeignKey("care_facilities.id", ondelete="CASCADE"),
        nullable=True,
    )
    medicalInfo: Mapped[str | None] = mapped_column(Text, nullable=True)

# yeirin-backend가 synchronize:true로 스키마를 바꾸면 soul-e는 런타임에 조용히 깨진다.
# API를 우회해 남의 DB를 읽는 727줄 — 모놀리스 전환 후 0줄 (import 한 줄로 대체).
```

TypeORM 테이블을 파이썬으로 727줄 손 복사 — synchronize:true 한 번이면 조용히 무너집니다

성능은 한 번도
문제였던 적이 없습니다.
문제는 금요일 밤의 저였습니다.

공식 성능테스트 12/12 통과 · API p95 1,340ms · 30일 가동률 99.87%
— 그리고 그 MSA를 운영하는 개발자는 1명이었습니다

결정의 순간들

2026년 3월, 세 갈래 길 앞에서

결정 #1 — 계속 갈 수 있는가

2026년 3월 · 결정의 순간 #1

"이 시스템을, 이 인원으로, 계속 운영할 수 있는가?"

레거시 운영 4개월차 · 레포 6개 · 매주 금요일 밤마다 스스로에게 돌아오던 질문입니다

✗

A. MSA를 제대로 재정비

컨테이너화 완성 · 관측성 추가 · ASG 도입

운영 비용의 **원인(경계의 수)**은 그대로.
1인 팀에게는 경계 하나하나가 매달 나가는 월세였습니다

✗

B. NestJS 단일 모놀리스

이미 절반의 코드가 파이썬(FastAPI 3개).

TS DI 세리머니를 전체 코드로
확장하는 방향 — 저희의 흉터가 더 많은 쪽이었습니다

✓

C. FastAPI 모듈러 모놀리스

경계는 **패키지**로 남기고 운영은 하나로.

나중에 진짜 쪼개야 할 날이 오면
패키지째 떼어낼 수 있도록 구조를 남겨 두기로 했습니다

모놀리스는 MSA의 반대말이 아니라, MSA를 나중에 저렴하게 시작하는 방법이었습니다

결정 #2 — 왜 FastAPI인가

결정의 순간 #2

"파이썬으로 간다면 — Django냐 FastAPI냐"

프레임워크 우열 논쟁이 아니라, 저희 조직의 흉터와 관성에 대한 질문이었습니다

✗

Django

admin과 ORM은 정말 매력적이었습니다.

하지만 전면 async + LLM 스트리밍
파이프라인과 결이 다르고,
도메인 분리에는 앱 구조의 관성이 아쉬웠습니다

✓

FastAPI

이미 서비스 3개가 FastAPI — 검증된 관성.
타입 힌트가 곧 스펙 → OpenAPI-first로
프론트 5개가 codegen 소비.
async 네이티브 + Depends의 단순함

✗

NestJS 유지

가장 큰 흉터들이 이쪽에 있었습니다.

forwardRef 10곳 · DTO 780줄
· 문자열 토큰 14회 재등록

"어떤 프레임워크가 좋은가"가 아니라 "우리의 흉터가 어느 쪽에 더 적은가"를 물었습니다

결정 #3 — 빅뱅이냐 스트랭글러냐

결정의 순간 #3

"서비스는 멈출 수 없다. 그리고 나는 혼자다."

이 두 가지 제약이 마이그레이션 전략을 대신 정해 주었습니다

✗

빅뱅 전환

전부 새로 지어 한 번에 전환하는 길입니다.

리스크가 한 지점에 집중되고,
중간 검증이 어렵습니다.
실패하면 복구도 혼자 감당해야 합니다

✓

도메인 단위 스트랭글러

하루 1~2개 도메인씩, 같은 리듬의 3연타 커밋:
도메인 모델 → Command/Query → 인프라

레거시 API 계약(camelCase flat DTO)은 새 정규화 모델의 **파생 필드**로 보존했습니다 — 프론트엔드는 변화를 느끼지 못합니다

완성하지 못한 부분은 숨기지 않고 **스텁 8개**로 표시한 채 먼저 출하했습니다

13일간 도메인만 쌓고, 14일째에 앱을 조립했습니다 — 계획서가 아니라 git log에 남아 있는 사실입니다

02 NestJS를 파이썬으로 번역하기

Spring 스타일 개념 ↔ 파이썬 관용구 · 입문 환영

개념 번역 지도

SPRING 스타일 바이블		FastAPI 모듈러 모놀리스
`@Module` 선언문 + providers 배열	→	파이썬 `import` + `pyproject.toml`
`@Injectable()` + DI 컨테이너	→	`Depends()` 함수 체인 — 컨테이너 없음
class-validator 데코레이터 4개/필드	→	Pydantic 타입 힌트 1개 = 검증+문서+타입
TypeORM `@Entity` · PG enum	→	SQLAlchemy 2.0 `Mapped[]` · String+StrEnum
Guard 전역 default-on + `@Public()`	→	`user: CurrentUser` 타이핑으로 opt-in
`forwardRef(() ⇒ X)` 순환 우회	→	함수 지역 import — 파이썬 관용구 그대로

왼쪽을 몰라도 괜찮습니다 — 오른쪽은 이미 여러분이 쓰고 계신 파이썬입니다

NestJS 요청의 일생 (입문)

레이어 자체는 훌륭한 교과서입니다 — 문제는 전부 '수동 등록'이라는 점이었습니다

모듈 선언 — 95줄 vs 13줄

Before · NestJS @Module

yeirin-backend/src/presentation/counsel-request/counsel-request.module.ts (95줄)

```ts
// ↑ 상단에 import 문 37줄 생략

@Module({
  imports: [
    TypeOrmModule.forFeature([
      CounselRequestEntity,
      CounselRequestRecommendationEntity,
      VoucherLinkageEntity,
      BImpactVoucherInstitutionEntity,
      CommonVoucherInstitutionEntity,
      ChildProfileEntity,
    ]),
    ConfigModule,
    ChildModule,     // ChildRepository 사용 (권한 검증)
    MatchingModule, // Matching Domain Service 사용 (DDD 패턴)
    UploadModule,    // S3Service 사용 (통합 보고서 Presigned URL)
  ],
  controllers: [CounselRequestController],
  providers: [
    {
      provide: 'CounselRequestRepository',       // ← 문자열 토큰 (레포 전체 14회 재등록)
      useClass: CounselRequestRepositoryImpl,
    },
    {
      provide: VOUCHER_LINKAGE_REPOSITORY,
      useClass: VoucherLinkageRepositoryImpl,
    },
    OpenAIClient,
    SoulEClient,     // Soul-E MSA 클라이언트
    YeirinAIClient,  // Yeirin-AI MSA 클라이언트
    CreateCounselRequestUseCase,
    CreateCounselRequestFromSouliUseCase,
    GetCounselRequestUseCase,
    // … UseCase 14개 더, 전부 손으로 나열 …
    SelectVoucherInstitutionUseCase,
  ],
  exports: ['CounselRequestRepository', 'CounselRequestRecommendationRepository'],
})
export class CounselRequestModule {}
```

After · pyproject.toml + include_router

yeirin-platform/packages/counseling/pyproject.toml (13줄 전체)

```toml
[project]
name = "yeirin-counseling"
version = "2.0.0"
description = "상담기관 연계, B-IMPACT 인증 도메인"
requires-python = "≥3.12"
dependencies = ["yeirin-core", "openai"]   # FastAPI 없음 — 도메인은 프레임워크를 모른다

[tool.uv.sources]
yeirin-core = { workspace = true }

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

파이썬의 import 그래프가 곧 의존성 그래프 — 등록할 배열이 없습니다

DTO — 데코레이터 4개 vs 타입 힌트 1개

Before · class-validator 780줄

yeirin-backend/src/application/counsel-request/dto/create-counsel-request.dto.ts (총 780줄 중 발췌)

```ts
export class RequestDateDto {
  @ApiProperty({ description: '년', example: 2024 })
  @IsInt()
  year: number;

  @ApiProperty({ description: '월', example: 11, minimum: 1, maximum: 12 })
  @IsInt()
  @Min(1)
  @Max(12)
  month: number;                          // 필드 하나에 데코레이터 4개:
                                          // 문서(@ApiProperty) + 검증(@IsInt/@Min/@Max)
  @ApiProperty({ description: '일', example: 18, minimum: 1, maximum: 31 })
  @IsInt()
  @Min(1)
  @Max(31)
  day: number;
}

export class CoverInfoDto {
  @ApiProperty({ description: '의뢰 일자' })
  @ValidateNested()
  @Type(() => RequestDateDto)             // TS 타입은 런타임에 지워지므로
  requestDate: RequestDateDto;            // 데코레이터로 타입을 한 번 더 알려줘야 함

  @ApiProperty({ description: '센터명', example: '서울아동발달센터' })
  @IsString()
  @IsNotEmpty()
  centerName: string;
}

// … 이런 클래스가 780줄 동안 계속된다 (레포에서 3번째로 큰 파일이 DTO) …
```

After · Pydantic ~34줄

yeirin-platform/packages/app_guardian/.../routers/counsel_requests.py

```python
class CreateCounselRequestRequest(CamelModel):
    child_id: UUID = Field(description="대상 아동 ID")
    center_name: str = Field("", description="센터 이름")
    care_type: str = Field("GENERAL", description="돌봄 유형 (GENERAL, SPECIAL 등)")
    linkage_project: str | None = Field(None, description="연계사업 (2026_BUSAN_SOCIAL 등)")
    cover_info: dict[str, Any] = Field(default={}, description="표지 정보")
    basic_info: dict[str, Any] = Field(default={}, description="기본 정보")
    psychological_info: dict[str, Any] = Field(default={}, description="심리 정보")
    request_motivation: dict[str, Any] = Field(default={}, description="의뢰 동기")
    consent: str = Field("AGREED", description="동의 여부 (AGREED)")
    guardian_info: dict[str, Any] | None = Field(None, description="보호자 정보")

# 타입 힌트 하나 = 런타임 검증 + OpenAPI 스키마 + 정적 타입 (3역할)
# str | None 이 @IsString() + @IsOptional() 을 대체
```

타입 힌트 하나가 런타임 검증 + OpenAPI 문서 + 정적 타입, 세 몫을 합니다

의존성 주입 — 컨테이너 vs 함수 인자

Before · 생성자 주입 17개

yeirin-backend/src/presentation/counsel-request/counsel-request.controller.ts

```typescript
@ApiTags('상담의뢰지')
@Controller('api/v1/counsel-requests')
@UseGuards(JwtAuthGuard)
@ApiBearerAuth()
export class CounselRequestController {
  constructor(
    private readonly createCounselRequestUseCase: CreateCounselRequestUseCase,
    private readonly createCounselRequestFromSouliUseCase: CreateCounselRequestFromSouliUseCase,
    private readonly getCounselRequestUseCase: GetCounselRequestUseCase,
    private readonly getCounselRequestsByChildUseCase: GetCounselRequestsByChildUseCase,
    private readonly getCounselRequestsByInstitutionUseCase: GetCounselRequestsByInstitutionUseCase,
    private readonly getCounselRequestsPaginatedUseCase: GetCounselRequestsPaginatedUseCase,
    private readonly updateCounselRequestUseCase: UpdateCounselRequestUseCase,
    private readonly deleteCounselRequestUseCase: DeleteCounselRequestUseCase,
    private readonly requestRecommendationUseCase: RequestCounselRequestRecommendationUseCase,
    private readonly getRecommendationsUseCase: GetCounselRequestRecommendationsUseCase,
    private readonly selectInstitutionUseCase: SelectRecommendedInstitutionUseCase,
    private readonly startCounselingUseCase: StartCounselingUseCase,
    private readonly completeCounselingUseCase: CompleteCounselingUseCase,
    private readonly getChildAssessmentResultsUseCase: GetChildAssessmentResultsUseCase,
    private readonly submitLinkageInfoUseCase: SubmitLinkageInfoUseCase,
    private readonly recommendVoucherInstitutionsUseCase: RecommendVoucherInstitutionsUseCase,
    private readonly selectVoucherInstitutionUseCase: SelectVoucherInstitutionUseCase,
  ) {}  // ← DELETE 요청 하나에도 17개 전부 인스턴스화된다
```

After · Depends 함수 체인

yeirin-platform/packages/app_guardian/.../dependencies.py (984줄 중 발췌)

```python
"""FastAPI DI 의존성 체인.

모든 Repository → Handler DI를 여기서 조합한다.
"""

DBSession = Annotated[AsyncSession, Depends(get_db_session)]


def get_child_repo(session: DBSession):
    from yeirin_child.infrastructure.repositories import SqlAlchemyChildRepository

    return SqlAlchemyChildRepository(session)


def get_create_child_handler(repo=Depends(get_child_repo)):
    from yeirin_child.application.commands import CreateChildHandler

    return CreateChildHandler(repo)


# 컨테이너 없음 · 데코레이터 없음 · 문자열 토큰 없음 — 전부 grep 되는 함수.
# 엔드포인트는 필요한 핸들러 하나만 받는다:
#   async def create_child(body, user: CurrentUser,
#                          handler=Depends(get_create_child_handler)) → ChildDto:
```

DELETE 한 번에도 17개를 전부 생성 vs 엔드포인트가 쓰는 것만 조립합니다

결정 #4 — DI 컨테이너를 쓸 것인가

결정의 순간 #4 · 이건 지금도 가끔 흔들린다

"DI 컨테이너 라이브러리를 쓸 것인가?"

dependency-injector · punq · wireup ··· 후보는 많았습니다

✗

컨테이너 도입

등록·스코프·모듈 개념이 다시 생겨납니다.

그 '등록이라는 세리머니'가
바로 저희가 NestJS를 떠나며
내려놓고 싶었던 것이었습니다

✓

순수 Depends + 손 팩토리

비용을 **알면서** 선택했습니다:
provider 함수 452개 · 약 3,900줄을 직접 씁니다.

대신 마법이 하나도 없습니다 — 전부 grep으로 찾을 수 있는 평범한 함수라서, 온보딩과 디버깅의 부담이 가볍습니다

공짜는 없었습니다 — 다만 빚의 형태를 골랐을 뿐입니다 (프레임워크 마법 대신, 눈에 보이는 보일러플레이트)

DI 순환의 실측

yeirin-backend — DI 순환의 스케일을 재보자

```
$ grep -rl 'forwardRef' src --include='*.module.ts'
src/presentation/admin-api/settings/admin-settings.module.ts
src/presentation/admin-api/audit-log/admin-audit-log.module.ts
src/presentation/admin-api/risk-management/admin-risk-management.module.ts
src/presentation/admin-api/institution/admin-institution.module.ts
src/presentation/admin-api/dashboard/admin-dashboard.module.ts
src/presentation/admin-api/consent/admin-consent.module.ts
src/presentation/admin-api/review/admin-review.module.ts
src/presentation/admin-api/sessions/admin-session.module.ts
src/presentation/admin-api/counsel-request/admin-counsel-request.module.ts
src/presentation/admin-api/children/admin-children.module.ts

# admin 서브모듈 12개 중 10개 — 전부 forwardRef(() ⇒ AdminAuthModule)
# 원인은 하나: 모든 서브모듈이 AdminAuthModule의 가드가 필요한데, 부모가 그걸 export

$ grep -rn 'QueryRunner\|@Transaction\|manager.transaction' src | wc -l
       0
# 그리고 트랜잭션 경계는 코드베이스 어디에도 없었습니다
```

admin 모듈 12개 중 10개가 forwardRef — 그리고 트랜잭션 경계는 0건이었습니다

ORM — TypeORM vs SQLAlchemy 2.0

Before · @Entity + PG native enum

yeirin-backend/src/infrastructure/persistence/typeorm/entity/counsel-request.entity.ts

```typescript
@Entity('counsel_requests')
@Index('idx_counsel_requests_status', ['status'])
@Index('idx_counsel_requests_status_created', ['status', 'createdAt'])
export class CounselRequestEntity {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'uuid', name: 'child_id' })
  childId: string;

  @ManyToOne(() => ChildProfileEntity, { nullable: false })
  @JoinColumn({ name: 'child_id' })
  child: ChildProfileEntity;            // ← 자동 관계 로딩 = N+1 지뢰

  @Column({
    type: 'enum',                       // ← PG 네이티브 enum:
    enum: CounselRequestStatus,         //     값 하나 추가할 때마다 ALTER TYPE
    default: CounselRequestStatus.PENDING,
  })
  status: CounselRequestStatus;

  @Column({ type: 'jsonb', name: 'form_data' })
  formData: CounselRequestFormData;

  @Column({ type: 'enum', enum: CareType, nullable: true, name: 'care_type' })
  careType: CareType;
}
// 스키마 관리: synchronize: isDevelopment — 마이그레이션 파일 0개
// (src/scripts/fix-enum-names.ts 수리 스크립트가 흉터로 남아 있다)
```

After · Mapped[] + String + StrEnum

yeirin-platform/packages/counseling/.../infrastructure/counsel_request/models.py

```python
class CounselRequestModel(UUIDPrimaryKeyMixin, TimestampMixin, SoftDeleteMixin, Base):
    """상담의뢰 테이블."""

    __tablename__ = "counsel_requests"

    child_id: Mapped[str] = mapped_column(
        UUID(as_uuid=True),
        nullable=False,
        comment="아동 ID (child_profiles.id)",   # 논리 FK — 패키지 경계 넘는 FK 제약 없음
    )
    institution_id: Mapped[str | None] = mapped_column(   # Mapped[str | None] 이
        UUID(as_uuid=True),                               # mypy가 검사하는 바로 그 어노테이션
        nullable=True,
        comment="의뢰 기관 ID (institutions.id)",
    )
    status: Mapped[str] = mapped_column(
        String(20),                       # ← enum은 코드 레벨 StrEnum, DB는 String
        nullable=False,                   #    값 추가 = 마이그레이션 0건 (전사 규칙, 156개)
        default="PENDING",
        comment="상태 (PENDING, RECOMMENDED, MATCHED, IN_PROGRESS, COMPLETED, REJECTED)",
    )
    form_data: Mapped[dict] = mapped_column(
        JSONB,
        nullable=False,
        default=dict,
        comment="상담의뢰서 양식 데이터",
    )
# comment= 는 tbls로 DB 스키마 문서 사이트가 된다. 관계(@ManyToOne) 없음 — 조인은 명시적.
```

PG enum에 데인 흉터가 'DB에는 String만' 규칙이 되었습니다

삽질 로그 #1

Enum 하나 추가했을 뿐인데

상황

TypeORM `type:'enum'`이 만든 PG 네이티브 enum. 상태값 하나를 추가할 때마다 **ALTER TYPE 마이그레이션**이 필요했고, 타입 이름은 자꾸 꼬였습니다

원인

DB가 enum을 알게 되면, enum의 진화가 곧 **DDL 이벤트**가 됩니다.
급기야 `fix-enum-names.ts` 수리 스크립트까지 만들었습니다

남긴 것

신규 플랫폼 절대규칙 —
"Enum은 코드(StrEnum)에서만, DB는 String"
현재 StrEnum 156개 · PG enum 0개 — 규칙에는 흉터가 담겨 있습니다

인증 — default-on vs opt-in

Before · 전역 가드 + @Public() 탈출구

yeirin-backend/src/infrastructure/auth/guards/jwt-auth.guard.ts (31줄 전체 — 가드 3벌 중 1벌)

```typescript
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {
    // Public 데코레이터 확인
    const isPublic = this.reflector.getAllAndOverride<boolean>('isPublic', [
      context.getHandler(),
      context.getClass(),
    ]);

    if (isPublic) {
      return true;
    }

    // Admin API는 별도 AdminJwtAuthGuard 사용 - 글로벌 가드 스킵
    const request = context.switchToHttp().getRequest<Request>();
    if (request.path.startsWith('/admin')) {    // ← 제2 인증스택을 위한
      return true;                              //     경로 문자열 우회 해킹
    }

    return super.canActivate(context);
  }
}
// 전역 default-on + @Public() 탈출구 방식. RolesGuard에도 같은 /admin 우회가 또 있다.
```

After · user: CurrentUser 타이핑

yeirin-platform/packages/app_guardian/.../middleware/auth.py (30줄 전체)

```python
def _extract_token(authorization: str | None = Header(default=None)) -> str:
    """Authorization 헤더에서 Bearer 토큰을 추출한다."""
    if not authorization:
        raise AuthenticationError("인증 토큰이 필요합니다")
    if not authorization.startswith("Bearer "):
        raise AuthenticationError("올바른 인증 형식이 아닙니다 (Bearer)")
    return authorization[7:]


def get_current_user(token: str = Depends(_extract_token)) -> TokenPayload:
    """현재 사용자 토큰 페이로드를 반환한다."""
    payload = get_token_payload(token)
    if payload.token_type ≠ "access":
        raise AuthenticationError("액세스 토큰이 아닙니다")
    return payload


CurrentUser = Annotated[TokenPayload, Depends(get_current_user)]

# 사용: async def list_counsel_requests(user: CurrentUser, ...) → ...:
# default-off, 시그니처에 타이핑해서 opt-in — 인증 요구가 함수 서명에 보인다
```

인증 요구가 함수 시그니처에 보입니다 — 가드 3벌 337줄 → 30줄

테스트 — 컨테이너 조립 vs 그냥 생성자

Before · Test.createTestingModule 30줄 준비

yeirin-backend/src/.../create-counsel-request.usecase.spec.ts — 단언문 전 30줄의 준비

```typescript
beforeEach(async () => {
  mockRepository = {
    save: jest.fn(),
    findById: jest.fn(),
    findByChildId: jest.fn(),
    findByStatus: jest.fn(),
    findByInstitutionId: jest.fn(),
    findByCounselorId: jest.fn(),
    findAll: jest.fn(),
    delete: jest.fn(),
  };                                        // ← 메서드 8개 전부 수동 스텁

  mockYeirinAIClient = {
    requestIntegratedReport: jest.fn().mockResolvedValue(undefined),
  } as unknown as jest.Mocked<YeirinAIClient>;    // ← TS 달래기용 이중 캐스팅

  mockSoulEClient = {
    getLatestAssessmentResult: jest.fn().mockResolvedValue(null),
  } as unknown as jest.Mocked<SoulEClient>;       // ← 행 하나 만드는 테스트에
                                                  //     MSA 클라이언트 2개 목킹 필수
  const module: TestingModule = await Test.createTestingModule({
    providers: [
      CreateCounselRequestUseCase,
      { provide: 'CounselRequestRepository', useValue: mockRepository },
      { provide: YeirinAIClient, useValue: mockYeirinAIClient },
      { provide: SoulEClient, useValue: mockSoulEClient },
    ],
  }).compile();                             // ← 테스트마다 DI 컨테이너 재조립

  useCase = module.get<CreateCounselRequestUseCase>(CreateCounselRequestUseCase);
});
```

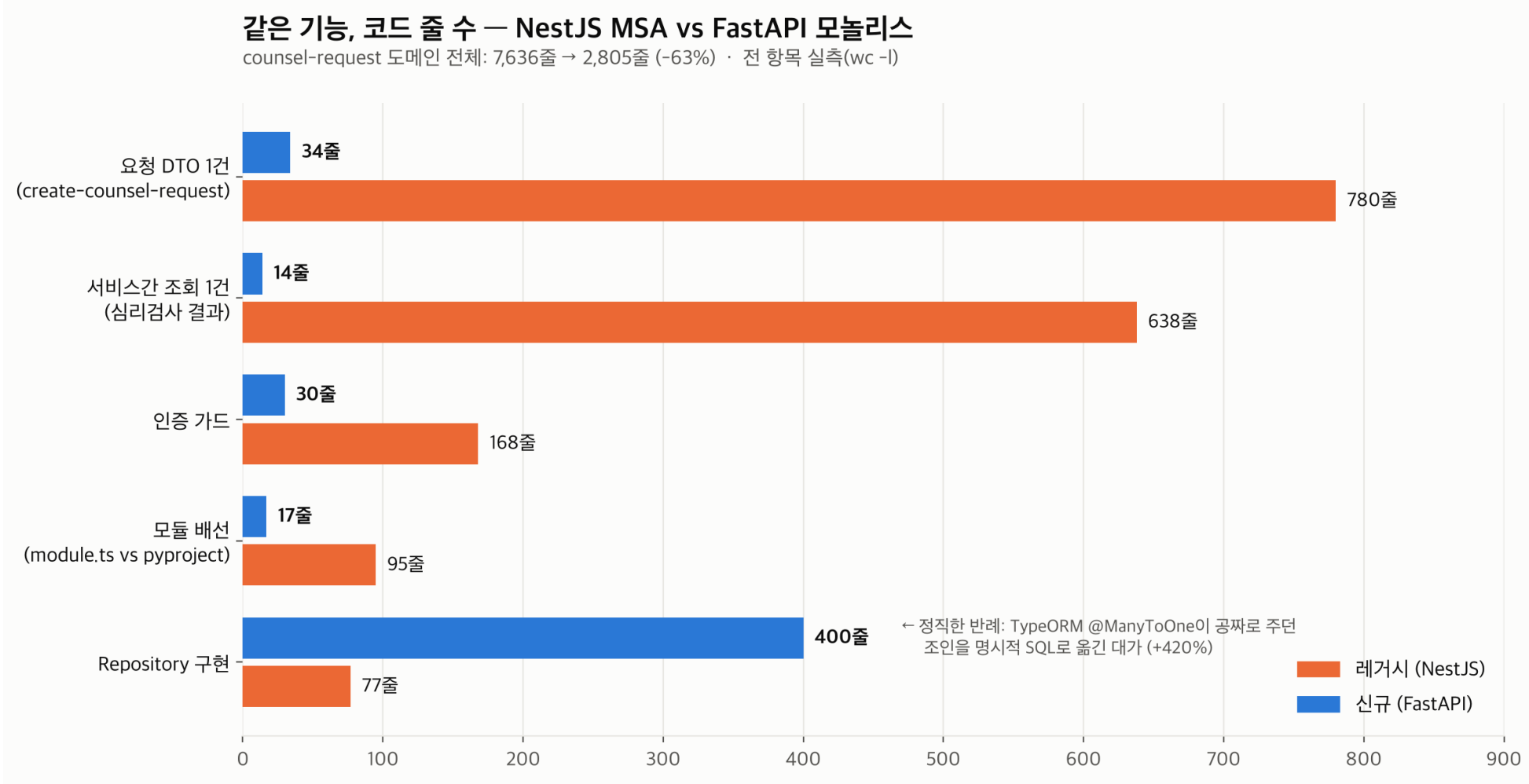
After · Handler(mock_repo) + AsyncMock

yeirin-platform/packages/counseling/tests/ — conftest.py + 테스트 본문

```python
# ── conftest.py ──────────────────────────────────
def _make_mock_repo(save_returns_input: bool = True) -> AsyncMock:
    """save 호출 시 입력 엔티티를 그대로 반환하는 mock repo 생성."""
    repo = AsyncMock()                      # ← 모든 메서드 자동 스텁
    if save_returns_input:
        repo.save = AsyncMock(side_effect=lambda e: e)
    return repo


@pytest.fixture()
def mock_counsel_request_repo() -> AsyncMock:
    return _make_mock_repo()


# ── test_counsel_request_handlers.py ─────────────
class TestCreateCounselRequestHandler:
    async def test_creates_with_correct_fields(self, mock_counsel_request_repo: AsyncMock) -> None:
        handler = CreateCounselRequestHandler(mock_counsel_request_repo)  # ← 그냥 생성자.
                                                                          #    컨테이너 없음.
        result = await handler.handle(
            CreateCounselRequestCommand(
                child_id=uuid4(),
                center_name="테스트센터",
                care_type="PRIORITY",
                form_data={"cover_info": {"name": "홍길동"}},
            )
        )

        assert result.status == "PENDING"
        events = _saved_entity(mock_counsel_request_repo).collect_events()
        assert isinstance(events[0], CounselRequestCreated)
```

DI가 함수 인자라서 테스트에 프레임워크가 필요 없습니다

같은 기능, 코드량 실측



도메인 전체 −63% · 다만 Repository는 +420% — 청구서는 6장에서 보여드립니다

03 모듈러 모놀리스 해부

uv workspace가 경계의 뼈대

새 플랫폼 전체 지도

앱 패키지 6개 — 조립 전담 (라우터 · 인증 · DTO만)

- **guardian** :11000 · 91
- **admin** :11001 · 185
- **myfriend** :11002 · 18
- **plobi-school** :11003 · **285**
- **cida** :11004 · 84

↓

도메인 패키지 19개 — 비즈니스 로직 소유 (FastAPI 의존성 없음)

child 8.8k · counseling 8.0k · field_record 6.9k · myfriend_engine 5.9k · voucher 5.6k · hr 4.8k · assessment 4.2k · care_center · dispatch · visit · consent · + 8개 더

↓

core — 1,179줄 (전체의 1.1%)

Base·믹스인 · get_db_session · Settings · auth · 예외 핸들러 · CamelDto

PostgreSQL 16

테이블 102개 · 단일 스키마

alembic 72리비전 · head 1개

패키지 25개(도메인 19 + 앱 6) · core는 전체의 1.1% · 테이블 102개입니다

모노레포의 뼈대 — uv workspace

yeirin-platform/pyproject.toml — 모노레포의 뼈대

```toml
[project]
name = "yeirin-platform"
version = "2.0.0"
description = "예이린 통합 연계시스템 v2.0 — 모듈러 모놀리스 플랫폼"
requires-python = "≥3.12"
dependencies = [
    "gunicorn≥25.3.0",
]

[tool.uv.workspace]
members = ["packages/*"]        # ← 패키지 25개 (도메인 19 + 앱 6)

[tool.uv]
package = false                 # 루트는 설치 대상이 아님 — 워크스페이스 홀더

[tool.uv.sources]
yeirin-core = { workspace = true }
yeirin-counseling = { workspace = true }
yeirin-voucher = { workspace = true }
# … 워크스페이스 소스 19개 …

# 락파일 1개(uv.lock, 174 deps)가 yarn.lock + uv.lock×3 을 대체.
# CI: uv sync --all-packages --frozen
```

락파일 1개가 yarn.lock + uv.lock×3을 대신합니다 · CI는 uv sync --all-packages --frozen

선언이 곧 경계

앱 — 필요한 도메인 + fastapi 선언

app_guardian
도메인 9개

app_admin
도메인 9개

app_plobi_school
도메인 15개

↓

도메인 — `dependencies = ["yeirin-core"]` 만

child

counseling

voucher

…16개

↓

core
pydantic · sqlalchemy · asyncpg — FastAPI 없음

선언이 곧 경계입니다.

앱은 **선언한 도메인만** import할 수 있고,

도메인 pyproject에는 FastAPI가 없어
프레임워크 import가 **물리적으로 불가능**합니다.

도메인 pyproject에 FastAPI가 없습니다 — 프레임워크 import가 물리적으로 불가능합니다

도메인 패키지 내부

packages/counseling — 76파일 · 8,049줄 · pyproject 13줄

domain/
순수 파이썬 + Pydantic
entities — `AggregateRoot`
repositories — `typing.Protocol`
value_objects — `StrEnum` (전사 156개)
events — 92클래스

←

application/
CQRS-lite
Command / Query
+ Handler (전사 497개)
DTO — `CamelDto` 상속
생성자 주입 클래스

←

infrastructure/
SQLAlchemy 2.0
models — `Mapped[]` + 믹스인
mappers — StrEnum 검증 지점
repositories — commit 안 함
(세션이 소유)

이 패키지에 없는 것: `router.py` ✗ `service.py` ✗ `schemas.py` ✗ — 전부 앱 패키지 소유

router.py도 service.py도 없습니다 — Protocol + Command/Query Handler + Mapper

요청 한 개의 DI 여행

세션 → repo → handler → 커밋 — 프레임워크 메타데이터가 없습니다

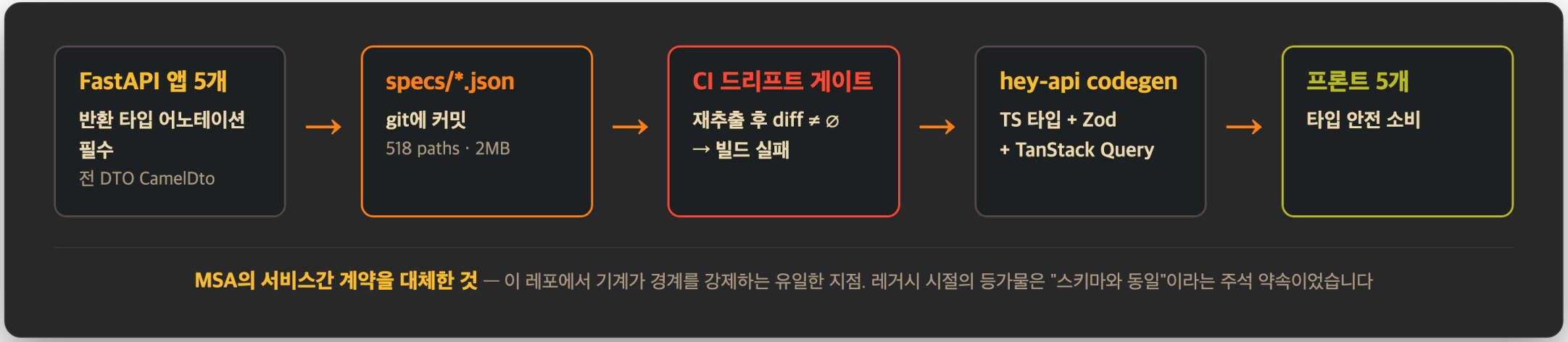
요청 = 트랜잭션

yeirin-platform/packages/core/src/yeirin_core/infrastructure/database.py

```python
async def get_db_session() -> AsyncGenerator[AsyncSession, None]:
    """FastAPI Depends용 비동기 DB 세션 제공자.

    SQLAlchemy 예외를 구조화된 도메인 예외로 변환한다.
    """
    session = async_session_factory()
    try:
        yield session
        await session.commit()          # ← 요청 = 작업 단위 (커밋은 여기 한 곳)
    except IntegrityError as exc:
        await session.rollback()
        raise DatabaseIntegrityError(_extract_integrity_detail(exc)) from exc
    except OperationalError as exc:
        await session.rollback()
        raise DatabaseConnectionError(
            "데이터베이스 연결에 실패했습니다. 잠시 후 다시 시도해 주세요.",
        ) from exc
    except DomainError:
        # 도메인 예외는 그대로 전파 (비즈니스 로직 실패 시 롤백)
        await session.rollback()
        raise
    except Exception:
        await session.rollback()
        raise
    finally:
        await session.close()

# 레거시 전체에서 QueryRunner/@Transaction grep 결과: 0건 — 트랜잭션 경계가 없었다.
# 지금은 요청 안의 모든 repo가 같은 세션 → 도메인 경계를 넘는 원자성이 공짜.
```

yield 제너레이터 하나가 커밋/롤백을 소유합니다 — 도메인 경계를 넘는 원자성이 따라옵니다

OpenAPI가 계약이다



스펙 518 paths를 git에 커밋하고, CI가 드리프트를 잡고, 프론트 5개가 codegen으로 소비합니다

04 HTTP에서 import로

13일의 전환, 그리고 그 리듬

같은 기능 — 638줄 vs 14줄

Before · axios 클라이언트 593줄 + 리매핑 45줄

yeirin-backend/src/infrastructure/external/soul-e.client.ts (총 593줄 중 메서드 1개)

```ts
async getAssessmentResults(childId: string): Promise<SoulEAssessmentResultSummary[]> {
  this.logger.log(`Soul-E 검사 결과 조회 요청 - childId: ${childId}`);

  try {
    const response = await this.client.get<SoulEAssessmentResultSummary[]>(
      `/api/v1/assessment/children/${childId}/results`,
    );
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      const axiosError = error as AxiosError;
      const status = axiosError.response?.status || HttpStatus.INTERNAL_SERVER_ERROR;

      // 404는 결과가 없는 경우 - 빈 배열 반환      ← 아무도 강제하지 않는 관례
      if (status === 404) {
        return [];
      }

      const message =
        (axiosError.response?.data as { detail?: string })?.detail ||
        axiosError.message ||
        'Soul-E service failed';

      throw new HttpException({ statusCode: status, message, service: 'soul-e' }, status);
    }
    // … 이 파일에는 상대 서비스의 Pydantic 스키마를 TS 인터페이스로
    //    손 복사한 코드가 ~130줄 더 있다 ("스키마와 동일" 주석 약속만 존재)
  }
}
```

After · 라우터 14줄 + import 한 줄

yeirin-platform/packages/app_guardian/.../routers/counsel_requests.py — 같은 기능 전체

```python
@router.get(
    "/counsel-requests/assessment-results/child/{child_id}",
    summary="아동별 심리검사 결과 조회",
    description="해당 아동의 심리검사 결과 목록을 조회합니다.",
)
async def get_assessment_results_by_child(
    child_id: UUID,
    user: CurrentUser,
    handler=Depends(get_results_by_child_handler),
) -> list[AssessmentResultDto]:
    from yeirin_assessment.application.queries import GetResultsByChildQuery

    return await handler.handle(GetResultsByChildQuery(child_id=child_id))

# 638줄(클라이언트 593 + 리매핑 45) + 네트워크 홉 + 공유 시크릿 + 타임아웃
#   → 위의 14줄. AssessmentResultDto는 양쪽에서 같은 파이썬 객체다.
```

네트워크 홉·공유 시크릿·타임아웃·'404는 빈 결과' 관례가 전부 사라졌습니다 (−98%)

우리 회사 안에서
우리 회사한테 HTTP를 쳤습니다.

서비스간 배관 코드 왕복 1,465줄 → import 한 줄
클라이언트·웹훅·미러 모델 합산 약 3,185줄이 0줄이 되었습니다

전체 타임라인

2025-11 ~ 2026-03

레거시 MSA 시대 — 4개월

4개 레포 · 432커밋

NestJS + FastAPI×3 병렬 운영

03-18 ~ 03-31

마이그레이션

단 14일

도메인 13일

+ 앱 조립 1일

2026-04 ~ 2026-08

모놀리스 확장 — 4.7개월

1개 레포 · 283커밋

62 → 666 엔드포인트 · plobi-school 285라우트 신설

레거시 마지막 실기능 커밋은 모놀리스 시작 8일 후 — 긴 이중 운영 없이 빠르게 전환했습니다 · 전 기간 개발자 1명

레거시 4개월 → 전환 14일 → 확장 4.7개월 · 전 기간 개발자 1명이었습니다

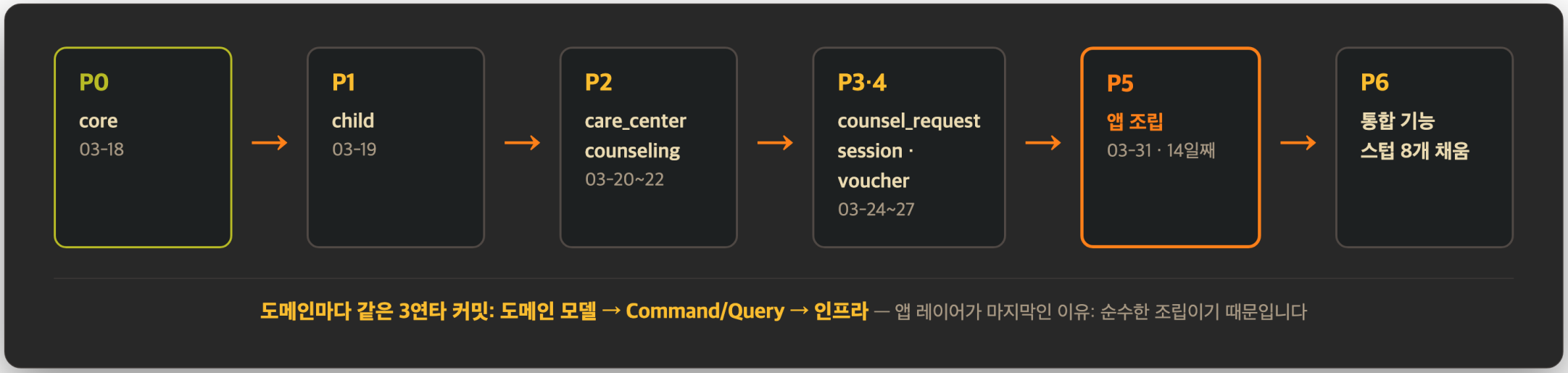
git log가 증거다

yeirin-platform — git log가 마이그레이션 전략의 증거다

```
$ git log --reverse --format='%ad  %s' --date=short | head -14
2026-03-18  🎉 init: uv workspace 모노레포 초기 설정
2026-03-18  ✨ feat(core): Base Entity, AggregateRoot, ValueObject 정의
2026-03-18  ✨ feat(core): Settings, Database, Auth 인프라 구현
2026-03-18  ✨ feat(core): 예외 핸들러, S3/SMS 클라이언트, 페이지네이션 DTO
2026-03-19  ✨ feat(child): 아동 도메인 모델 및 Value Objects 정의
2026-03-19  ✨ feat(child): 아동 CRUD Command/Query 핸들러 구현
2026-03-19  ✨ feat(child): SQLAlchemy 모델, Mapper, Repository 구현
2026-03-20  ✨ feat(care-center): 돌봄기관 Institution/Staff 도메인 모델
2026-03-20  ✨ feat(care-center): 인증/조회 Command/Query, DTO, 대시보드
2026-03-20  ✨ feat(care-center): 인프라 계층 — 모델, 매퍼, 리포지토리
2026-03-22  ✨ feat(counseling): 상담기관/종사자 도메인 모델 및 VO
2026-03-22  ✨ feat(counseling): 기관/스태프 인증 Command, Query, DTO
2026-03-24  ✨ feat(counsel-request): 상담의뢰 도메인 모델 및 워크플로우
2026-03-24  ✨ feat(counsel-request): CRUD + 추천/매칭 Command/Query

# 도메인마다 같은 3연타: 도메인 모델 → Command/Query → 인프라
# 13일간 도메인만 쌓고, 14일째(03-31) 앱을 조립했습니다 — 앱 레이어는 순수 조립
```

도메인마다 같은 3연타였습니다: 도메인 모델 → Command/Query → 인프라

스트랭글러 순서



core → child → … → 앱 조립은 맨 마지막 — 앱 레이어는 순수한 조립입니다

삽질 로그 #3 — 모노레포의 대가

삽질 로그 #3

몇 달간 '우연히' 돌아가던 패키지

상황

app-plobi-school이 yeirin-voucher를 **선언 없이 import** — 몇 달간 아무 문제 없이 돌아갔습니다

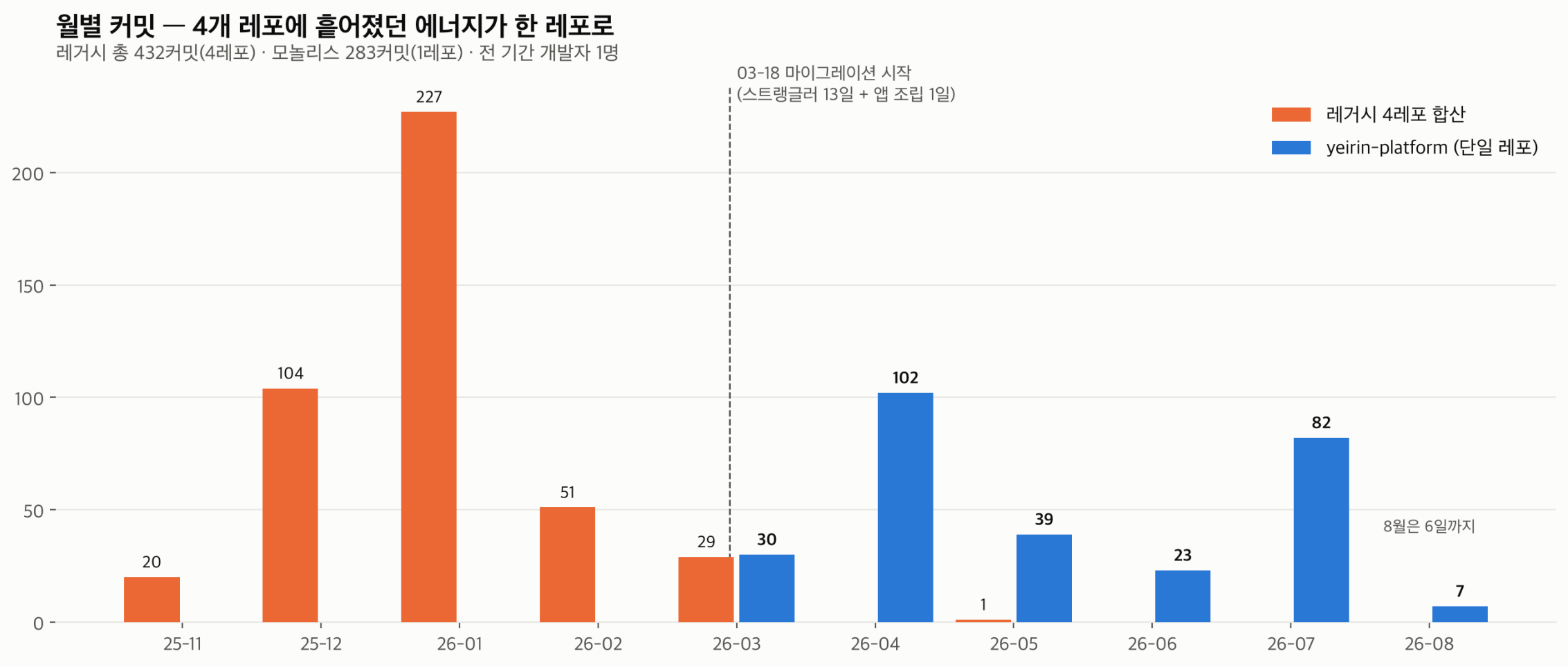
원인

`uv sync --all-packages`는 모든 패키지를 한 venv에 설치합니다.
선언하지 않아도 **옆 패키지가 그냥 존재**합니다 —
모노레포의 편리함이 경계 검증을 잠재우고 있었습니다

남긴 것

커밋 메시지: "yeirin-voucher 의존성 명시 선언".
경계는 선언만으로 완성되지 않는다는 배움과 함께
import-linter가 백로그 1순위가 된 날입니다

에너지의 이동



4개 레포에 흩어졌던 커밋이 한 레포로 모였습니다 — 총 432 vs 283

05 운영 이야기

배포·CI, 그리고 삽질 로그

Before — 배포 스크립트의 고고학

yeirin-backend/.github/workflows/deploy.yml — .env를 만드는 방법 (25줄 중 발췌)

```
# Load environment from SSM and create .env
echo "Loading environment from SSM Parameter Store..."

# Generate .env file using separate commands
echo "NODE_ENV=$(aws ssm get-parameter --name '/yeirin/dev/yeirin/node-env' --query 'Parameter.Value' --output text --region ap-northeast-2)" > .env
echo "PORT=$(aws ssm get-parameter --name '/yeirin/dev/yeirin/port' --query 'Parameter.Value' --output text --region ap-northeast-2)" >> .env
echo "DB_HOST=$(aws ssm get-parameter --name '/yeirin/dev/yeirin/db-host' --query 'Parameter.Value' --output text --region ap-northeast-2)" >> .env
echo "DB_PORT=$(aws ssm get-parameter --name '/yeirin/dev/yeirin/db-port' --query 'Parameter.Value' --output text --region ap-northeast-2)" >> .env
echo "DB_USERNAME=$(aws ssm get-parameter --name '/yeirin/dev/yeirin/db-username' --with-decryption --query 'Parameter.Value' --output text --region ap-northeast-2)" >> .env
echo "DB_PASSWORD=$(aws ssm get-parameter --name '/yeirin/dev/yeirin/db-password' --with-decryption --query 'Parameter.Value' --output text --region ap-northeast-2)" >> .env
echo "JWT_SECRET=$(aws ssm get-parameter --name '/yeirin/dev/common/jwt-secret' --with-decryption --query 'Parameter.Value' --output text --region ap-northeast-2)" >> .env
# … 같은 줄이 16개 더: SSM API를 순차로 23회 호출해서 .env를 한 줄씩 조립 …
echo "OPENAI_API_KEY=$(aws ssm get-parameter --name '/yeirin/dev/common/openai-api-key' --with-decryption --query 'Parameter.Value' --output text --region ap-northeast-2)" >> .env

# 그 전에는: yarn install --frozen-lockfile && yarn build
#            → 2vCPU/2GiB t3.small 위에서 프로덕션 빌드
# 그 후에는: pm2 delete yeirin && pm2 start  → 단일 인스턴스 = 배포마다 다운타임
# 이 워크플로우에 테스트 스텝은 없다.
```

SSM 순차 호출 23회로 .env를 조립하고, t3.small에서 프로덕션을 빌드했습니다 · 테스트 스텝은 없었습니다

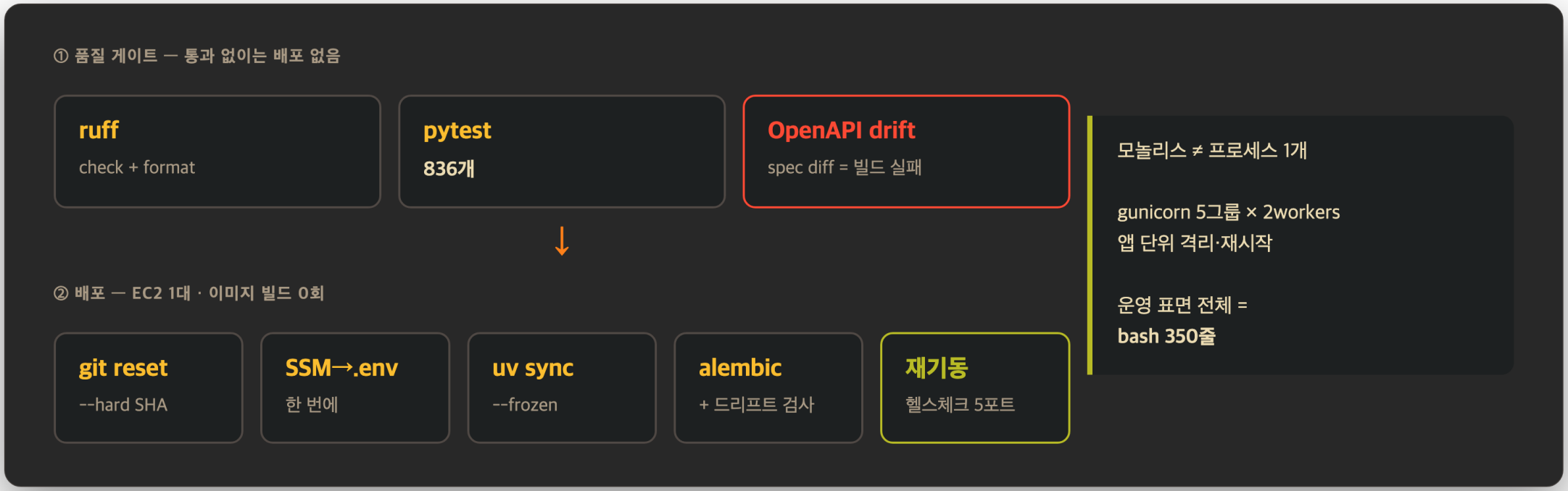
After — 파이프라인 한 파일

yeirin-platform/.github/workflows/pipeline.yml — CI 5벌을 대체한 1개 파일 (발췌)

```yaml
jobs:
  quality:
    name: 품질 게이트 (ruff · pytest 757 · OpenAPI drift)
    steps:
      - uses: astral-sh/setup-uv@v5
        with: { enable-cache: true }
      - name: Python 3.12 + 의존성 (uv.lock 고정)
        run: |
          uv python install 3.12
          uv sync --all-packages --frozen
      - name: Lint (ruff check)
        run: uv run ruff check .
      - name: Format check (ruff format)
        run: uv run ruff format --check .
      - name: Test (pytest)
        run: uv run pytest -q
      - name: OpenAPI spec drift              # ← 스펙 재추출 후 git diff가 있으면 실패
        run: ./scripts/ci-openapi.sh          #    커밋된 OpenAPI = 기계가 지키는 계약

  deploy-dev:
    needs: quality                            # 품질 게이트 통과 없이는 배포 없음
    steps:
      - uses: appleboy/ssh-action@v1
        with:
          script: |
            git reset --hard ${{ github.sha }}          # 이미지 빌드 0회
            SSM_PREFIX=/yeirin-platform/dev scripts/deploy.sh env   # SSM → .env 한 번에
            uv sync --all-packages --frozen
            set -a && source .env && set +a             # alembic은 .env를 직접 안 읽는다
            PYTHONPATH=. uv run alembic upgrade head
            PYTHONPATH=. uv run python scripts/check_schema_drift.py \
              || echo "::warning::스키마 드리프트 발견 (레거시 message_logs는 무해)"
            scripts/deploy.sh restart                   # 5개 앱 재기동 + 헬스체크
```

품질 게이트(ruff·pytest 836·OpenAPI 드리프트)를 통과해야만 배포됩니다

배포 토폴로지



모놀리스 ≠ 프로세스 1개 — gunicorn 5그룹×2workers, 운영 표면은 bash 350줄입니다

마이그레이션 체인 하나

yeirin-platform — 102개 테이블, 마이그레이션 체인은 하나

```
$ ls alembic/versions/*.py | wc -l
      72

$ PYTHONPATH=. uv run alembic heads
cc99dd00ee11 (head)

# 리비전 72개 · head 정확히 1개 — 패키지 25개가 체인 하나를 공유합니다
# 레거시: TypeORM synchronize:true (마이그레이션 0개) + 분산 alembic 체인 3개
# 배포마다: alembic upgrade head 1회 + 스키마 드리프트 검사가 게이트
```

리비전 72개 · head 1개 — 패키지 25개가 체인 하나를 함께 씁니다

린터 대신 훅 — 2026년식 거버넌스

yeirin-platform/.claude/hooks/validate-layer.sh — 린터가 아니라 AI 에이전트 훅으로 강제하는 아키텍처

```bash
#!/bin/bash
# PostToolUse hook: Write/Edit 후 도메인 레이어 순수성을 검증한다.
# domain/ 레이어에 프레임워크 의존이 들어가면 경고를 출력한다.

INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | python3 -c "import sys,json; print(json.load(sys.stdin)['tool_input']['file_path'])")

if echo "$FILE_PATH" | grep -q '/domain/'; then
  VIOLATIONS=$(grep -nE '^\s*(from|import)\s+(sqlalchemy|fastapi|uvicorn|httpx|redis|asyncpg|langchain|chromadb)' "$FILE_PATH")
  if [ -n "$VIOLATIONS" ]; then
    echo "⚠️  [yeirin-hook] domain/ 레이어에 프레임워크 import가 감지되었습니다!"
    echo "   domain 레이어는 Pydantic 외 외부 라이브러리에 의존하면 안 됩니다."
    echo "$VIOLATIONS" | sed 's/^/     /'
    echo "   → infrastructure/ 레이어로 이동하거나 Protocol을 사용하세요."
  fi
fi

# 함께 동작하는 훅 2개:
#   block-non-uv.sh    → pip/poetry/pipenv/conda 명령 자체를 차단
#   check-naming.sh    → packages/ 경로의 하이픈 디렉토리명 차단
# 2026년의 아키텍처 거버넌스: 위키 문서가 아니라 실행되는 스크립트 + 27개 CLAUDE.md
```

아키텍처 규칙을 위키가 아니라, 실행되는 스크립트와 27개의 CLAUDE.md에 담았습니다

삽질 로그 #2

삽질 로그 #2

린트 규칙 하나 켰다가 noqa 1,200개

상황

ruff의 TCH(flake8-type-checking)를 켰더니
순식간에 **noqa 주석 1,200개**가 생겼습니다

원인

Pydantic/FastAPI는 타입 어노테이션을 **런타임에 읽습니다.**
`TYPE_CHECKING`으로 옮기면 OpenAPI 생성이 깨집니다 —
일반 파이썬 프로젝트의 상식이 여기서는 역효과였습니다

남긴 것

규칙을 끄고, **끈 이유를 pyproject 주석으로 남겼습니다.**
"FastAPI 코드베이스의 린트 설정은 다릅니다" —
오늘 준비한 것 중 가장 실용적인 한 줄입니다

06 됩니다, 다만

정직한 청구서

경계는 새고 있다

yeirin-platform/packages/counseling/.../counsel_request/repositories.py — 경계가 새는 지점 (실코드)

```python
async def find_detail_with_voucher_context(self, *, request_id: UUID):
    """단일 join 으로 의뢰서 + 아동 + voucher_linkage + 매칭된 상담기관까지."""
    from yeirin_child.infrastructure.models import ChildProfileModel          # ← 다른 도메인의
    from yeirin_counseling.infrastructure.models import CounselingInstitutionModel
    from yeirin_voucher.infrastructure.linkage.models import VoucherLinkageModel  # ← ORM 모델을
                                                                                   #     함수 안에서 import

    stmt = (
        select(
            CounselRequestModel,
            ChildProfileModel.legal_name,
            VoucherLinkageModel.status,
            CounselingInstitutionModel.name,
        )
        .outerjoin(ChildProfileModel, CounselRequestModel.child_id == ChildProfileModel.id)
        .outerjoin(
            VoucherLinkageModel,
            (VoucherLinkageModel.child_id == CounselRequestModel.child_id)
            & (VoucherLinkageModel.linkage_project == CounselRequestModel.linkage_project)
            & (VoucherLinkageModel.is_deleted.is_(False)),
        )
        .where(CounselRequestModel.id == request_id)
    )

# 모놀리스의 payoff: MSA에서 네트워크 3홉이던 것이 SQL join 1방.
# 모놀리스의 cost:  README는 "도메인끼리 import 금지"라 말하지만,
#                   함수 지역 import라 모듈 레벨 순환이 아니어서 파이썬은 침묵한다.
# 완화책: child 패키지는 공개 join helper(current_state.py)를 제공 — "직접 컬럼 참조 금지"
```

payoff(네트워크 3홉 → SQL 1방)와 cost(경계 용해)는 같은 자리에 있습니다

누수의 지도

README의 주장 — "도메인 패키지는 core에만 의존한다.
도메인 패키지끼리 import하지 않는다."

↓ grep의 현실

child ⇄ counseling ⇄ voucher ⇄ care_center

누수 메커니즘
repository 메서드 안의 **함수 지역 import**로 도메인을 넘는 join을 만듭니다 — 모듈 레벨 순환이 아니어서 파이썬은 조용히 지나갑니다

완화책
공개 join helper 표면 (`current_state.py`)
"직접 컬럼 참조 금지" — 다음 과제는 import-linter입니다

README의 주장과 grep의 현실 — 4개 도메인이 순환하고 있었습니다

삽질 로그 #4 — 과거의 나와의 대화

삽질 로그 #4

649행 전부 NULL인 컬럼의 고고학

상황

`participation_status` 컬럼 + enum이 마이그레이션에
추가되어 있는데, **한 번도 값이 쓰인 적이 없었습니다**
(649행 전부 NULL)

원인

문서에 남긴 진단 그대로입니다 — "이전 누군가 같은 고민을
하다 멈춘 흔적이며, 지금 정식으로 결론 내야 함".
그 '누군가'는 4개월 전의 저였습니다

남긴 것

1인 개발의 코드 리뷰어는 미래의 자신이라는 것.
결론은 문서로 남겼습니다 — "지금 고치는 이유:
production 2 rows, 백필 부담이 가장 작은 순간"

JWT 하나를 다섯 앱이

BEFORE — 4벌 · 830줄 (같은 시크릿으로 각자 서명)

NestJS passport 50줄 + admin 제2 스택
soul-e jwt_service 742줄 · myfriend 38줄

→

AFTER — core/auth.py 160줄

bcrypt + python-jose · TokenPayload
토큰 패밀리: staff · child · plobi-school 역할 3종

↓

guardian | admin | myfriend | plobi-school | cida

정직한 잔여 리스크 — 시크릿 1개를 5개 앱이 공유하고, 방어는 claim 체크뿐입니다 (취약점을 docstring에 미리 적어 두었습니다)

인증 4벌 830줄 → 160줄 — 다만 시크릿 공유는 남은 숙제입니다

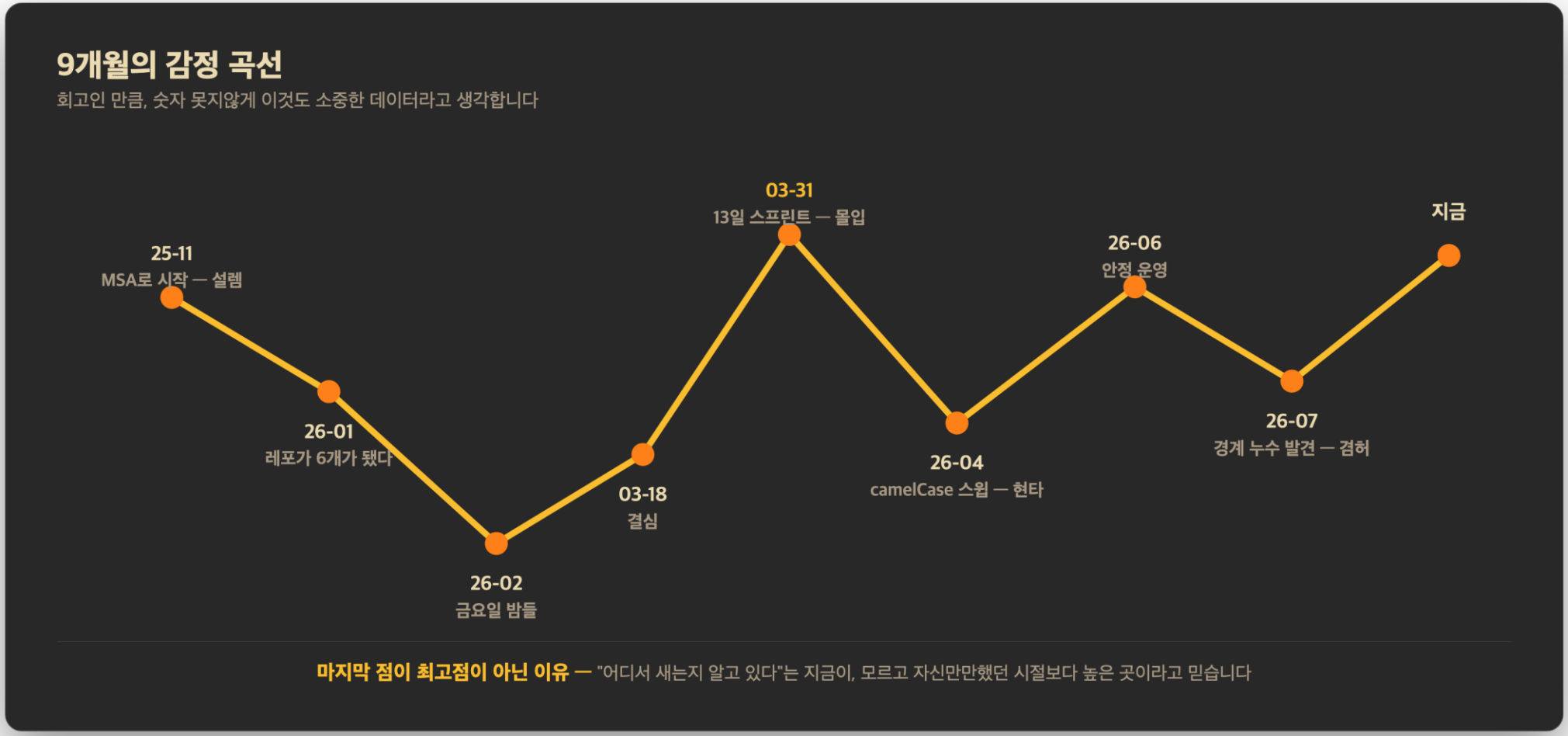
모듈 경계는 새고 있습니다.
어디서 새는지 알고 있다는 게
MSA 시절과의 차이입니다.

도메인 6개가 repository 안 함수 지역 import로 도메인을 넘는 join을 만듭니다
네트워크 3홉 대신 SQL 1방 — payoff와 cost는 같은 자리에 있습니다

돌아보며

감정 곡선 · KPT · 1년이 남긴 다섯 문장

9개월의 감정 곡선


9개월의 감정 곡선
회고인 만큼, 숫자 못지않게 이것도 소중한 데이터라고 생각합니다
25-11
MSA로 시작 — 설렘
26-01
레포가 6개가 됐다
26-02
금요일 밤들
03-18
결심
03-31
13일 스프린트 — 몰입
26-04
camelCase 스윕 — 현타
26-06
안정 운영
26-07
경계 누수 발견 — 겸허
지금
마지막 점이 최고점이 아닌 이유 — "어디서 새는지 알고 있다"는 지금이, 모르고 자신만만했던 시절보다 높은 곳이라고 믿습니다

마이그레이션이 지운 것

마이그레이션이 지운 것 — 머릿속 상주 메모리

서버 비용보다 먼저 가벼워진 것들입니다

BEFORE — 항상 로드되어 있던 것들

레포 6개의 컨텍스트 · 포트 지도 8개 · 언어 2개/버전 4벌 · 패키지 매니저 3종 · 내부 시크릿 헤더 3종 · DB 4개 접속 정보 · 서비스 기동 순서 · PM2와 ECS 두 벌의 운영법 · "그 코드가 어느 레포에 있더라?"

→

AFTER — 남은 것들

레포 1개 · 명령 사전 하나 (`make` · `deploy.sh`) · 규칙은 **훅과 CLAUDE.md가 기억** · 계약은 **OpenAPI 스펙이 기억**

기억을 사람에서 시스템으로 옮기는 일 — 그것이 이 리팩토링의 실체였습니다

저는 아키텍처가 아니라
제 삶을 리팩토링했습니다.

배포 유닛 9 → 5 · 머릿속 상주 컨텍스트 6레포 → 1레포
— 금요일 밤이 조용해졌습니다

숫자로 보는 전환

숫자로 보는 전환 — 그대로인 것은 팀 규모(1명)뿐

폴리글랏 MSA(NestJS 1 + FastAPI 3, 레포 6개) → uv workspace FastAPI 모듈러 모놀리스 · 2026-03 ~ 2026-08

레포	배포 유닛	서버	CI 파이프라인	데이터베이스
4	9	EC2 3대+ECS	5벌	4
↓	↓	↓	↓	↓
1	**5**	**1대**	**1벌**	**1**
	프로세스 그룹			스키마 1개

env 키	JWT 구현	서비스간 배관	CI 테스트	마이그레이션
121	4벌	3,185줄	0	체인 3+auto sync
↓	↓	↓	↓	↓
28	**1벌**	**0줄**	**836**	**72리비전 1체인**
SSM 자동생성	core 160줄	import로 대체	품질 게이트	alembic

그대로인 것은 팀 규모(1명)뿐입니다

회고 — Keep · Problem · Try

회고 — Keep · Problem · Try

Keep

- 도메인 먼저, 앱은 마지막 — 앱 레이어를 순수 조립으로 유지한 것
- 커밋된 OpenAPI + CI 드리프트 게이트 — 기계가 지키는 계약
- 흉터를 규칙으로 문서화 (enum · camelCase · TCH)

Problem

- 도메인 경계 누수 — 6개 패키지가 함수 지역 import로 순환
- mypy strict 설정만 있고 CI에 없음
- JWT 시크릿 1개를 5개 앱이 공유

Try

- import-linter로 경계를 기계 검증으로 승격
- mypy를 품질 게이트에 편입
- 앱별 시크릿 분리 · 팀 확장 대비 패키지 소유권 명문화

1년이 남긴 다섯 문장

1. 아키텍처는 기술이 아니라 **조직의 함수**입니다 — 1인 팀의 MSA는 세금만 내는 계약이었습니다
2. 성능 지표가 좋은데 삶이 나쁘다면, **그것이 기술부채**입니다 (p95 1.3초 · 가동률 99.87% — 그리고 금요일 밤)
3. 마이그레이션의 단위는 코드가 아니라 **도메인**입니다 — 하루 하나씩, 같은 리듬으로
4. 규칙은 **흉터가 있어야** 지켜집니다 — enum 규칙 뒤에는 수리 스크립트의 기억이 있습니다
5. 문서는 언젠가 낡습니다 — 꼭 지키고 싶은 것은 **CI와 혹과 타입**에 맡겨 주세요

삽질은 기록되는 순간
자산이 됩니다.

noqa 1,200개 → pyproject 주석 · PG enum → 절대규칙 · NULL 649행 → 결론 문서
오늘 발표의 절반은 그 기록에서 나왔습니다

다시, 처음의 질문으로

「FastAPI로 그게 됩니까?」

번역이 가능해진 이유 — 세 개의 기둥

번역이 가능해진 이유 — 세 개의 기둥

3년 전이었다면 훨씬 어려웠을 일입니다. 언어와 도구가 먼저 성숙해 주었습니다

uv

workspace가 모듈 경계의 뼈대가 됩니다
— 패키지 25개 · 락파일 1개(174 deps)

`uv sync --frozen` 초 단위 재현 ·
파이썬 버전 관리까지 한 도구로

모노레포 모듈러 모놀리스의
물리적 토대

Python 3.12+

타입 시스템의 성숙 —
`Protocol` 구조적 인터페이스 ·
`StrEnum` · `Annotated`

DDD의 어휘(인터페이스·VO·경계)를
표준 라이브러리만으로 표현할 수
있게 되었습니다

FastAPI

FastAPI + Pydantic v2

`Depends` 그래프 + 요청 단위 캐시
— DI 컨테이너 없는 의존성 주입

Rust 코어(pydantic-core)의 검증 성능 ·
타입 힌트가 곧 OpenAPI 계약

async 네이티브 — LLM 스트리밍까지
같은 프레임워크로

'빠른 API 도구'가 아니라, 엔터프라이즈 설계 어휘를 전부 받아낼 수 있는 생태계가 되었습니다

정설이 지키던 패턴들 — 번역 완료 확인서

정설이 지키던 패턴들 — 번역 완료 확인서

Spring · NestJS의 어휘 → 파이썬의 어휘 · 오른쪽 숫자는 전부 저희 프로덕션 실측입니다

✓ 계층형 아키텍처	Layered Architecture	domain / application / infrastructure — 도메인 패키지 19개
✓ DDD 전술 패턴	Aggregate · VO · Domain Event	AggregateRoot 53 · ValueObject · 도메인 이벤트 92클래스
✓ CQRS	Command / Query 분리	Command·Query Handler 497개
✓ 의존성 주입	@Injectable · DI 컨테이너	Depends 체인 — provider 452개, 컨테이너 없음
✓ Unit of Work	@Transactional	요청 = 작업 단위 — 세션 제너레이터 40줄
✓ 계약 기반 협업	인터페이스 · API Contract	OpenAPI 518 paths 커밋 + CI 드리프트 게이트
✓ 모듈 경계	Module System · Modulith	uv workspace 패키지 25개 — 선언이 곧 경계

이론이 아니라, 오늘 아침에도 돌아가고 있던 코드입니다

FastAPI의 Fast는
'서빙 속도'로 시작했지만,
저희에게는 '조직이 움직이는 속도'가
되었습니다.

엔드포인트 32 → 65를 하루 만에 · 전환 14일 · 운영 표면 bash 350줄
— 빠른 것은 API만이 아니었습니다

이제, 겁먹지 않으셔도 됩니다.
여러분의 파이썬은 이미
엔터프라이즈를 감당할 준비가
되어 있습니다.

---

uv · Python 3.12 · FastAPI + Pydantic v2

— 그리고 오늘 보여드린 105,594줄의 실측 전부가 그 증거입니다

감사합니다

질문 환영합니다

예이린 사회적협동조합 · sxngt.dev@gmail.com · 발표 자료의 모든 수치는 실측입니다