Hyeonjun Kil

Backend Engineer / Platform Part Leader

kboxstar@gmail.com GitHub ↗ LinkedIn ↗ Web Portfolio ↗

Development Operations Harness Framework

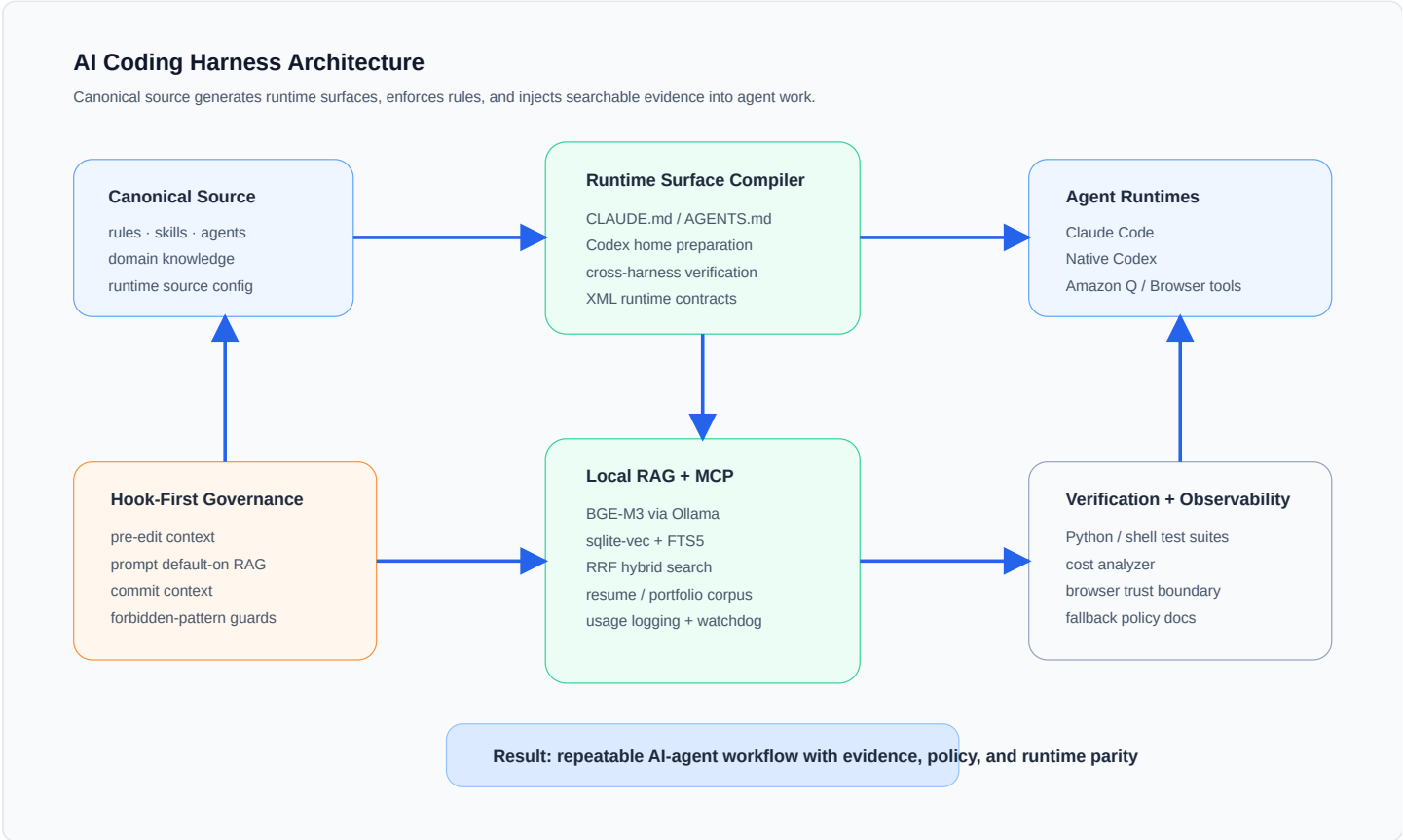
2026.04 — Present · Personal

GitHub ↗ Detail ↗

3 harnesses · RAG MVP · JD feedback loop

Git-based personal harness for reproducible development operations and resume/JD curation — integrating local BGE-M3 RAG, MCP search, Git-hook quality gates, JD-custom PDF generation, Zighang external feedback loops, and safe T1/T2/T3 application gates

TypeScript Python Bash MCP sqlite-vec FTS5 Ollama BGE-M3 Vector RAG Git Hooks Runtime Surface Compiler Quality Gates



Performance Metrics	Before	After
RAG MVP Index	manual context search	**548 files / 3,347 chunks / 20.7 MB** (0.7s warm query)
RAG Default-On Verification	opt-in trigger	**15/15 tests across 3 harnesses** (prompt >=12 chars auto-search)
Resume RAG Corpus	0 indexed chunks	**109 chunks** (32 tests passed)
Work-data Evidence Archive	manual evidence curation / 44% metric coverage	**279 incremental work items / schema v1.2** (50% metric coverage)
JD Feedback Loop	manual JD feedback capture	**JD PDF + Zighang recruitment-roast + T1/T2/T3 triage + form-start guard** (Juvis pass range / Nexon low-fit / Hanwha 82% improve-to-pass)

Problem Solving

Development-runtime context and rules drifted across docs, config, and local runtime homes

Made harness source the canonical source and generated runtime surfaces automatically, adding XML runtime contracts and cross-harness diff verification

→ **Aligned generated docs and runtime contracts across kh/gp/gd harnesses**

Growing markdown knowledge base forced agents to search context manually each time

Implemented sqlite-vec + FTS5 hybrid RAG, local BGE-M3 embeddings, RRF fusion, MCP search server, and automatic reindexing after markdown changes

→ **Indexed 548 files / 3,347 chunks / 20.7 MB, reached 0.7s warm queries, and hit top-2/3 on all 5 gold queries**

RAG was an opt-in reference tool and was missed during real investigation/review prompts

Moved to a default UserPromptSubmit trigger, calling search_harness automatically for prompts of 12+ characters and porting it to gp/gd

→ **Passed 15/15 default-on tests across 3 harnesses and hit the target in a real-prompt smoke test**

Resume/portfolio JSON and JD-specific external feedback were disconnected, leaving job-specific improvement dependent on raw feedback alone

Connected resume/portfolio JSON chunkers, JD-custom PDF generation, Zighang recruitment-roast capture, and T1/T2/T3 triage templates

→ Classified Juvis NestJS as pass-range, Nexon settlement as 35% low-fit, and Hanwha Life AI Backend as 82% improve-to-pass through raw feedback → T1/T2/T3 triage → verification loops

Resume and JD customization evidence was scattered across session logs and work records, forcing manual re-selection and quantification of recent impact each time

Standardized the private work-data archive on schema v1.2 with outcome_metrics, classified Jira/session evidence by resumeProject, and connected it into the resume/portfolio/RAG curation path

→ Incrementally collected 279 work items, improved metric coverage 44%→50%, and established the work-data → resume-candidate selection → verification loop

Tech Decisions

- Chose local BGE-M3 + sqlite-vec over cloud vector DBs to prioritize immediate reindexing after markdown changes and cost control
- Chose hook-first enforcement over doc-only rules to compensate for agents not reading policy documents at the moment of need
- Chose a runtime surface compiler over manual runtime settings to reduce generated-surface drift and automate 3-harness verification
- Chose default-on search over opt-in retrieval to reduce missed evidence during real investigation, review, and edit prompts

Highlights

- Aligned 3 harnesses through automatic runtime-surface generation plus XML runtime contracts
- Indexed 548 files / 3,347 chunks with local BGE-M3 RAG and reached 0.7s warm queries
- Moved RAG to default-on search_harness calls for prompts of 12+ characters, passing 15/15 tests across 3 harnesses
- Integrated resume/portfolio JSON into the RAG corpus, indexing 109 chunks and passing 32 tests
- Documented browser/REPL runtime trust boundaries and fallback order, then reflected settings across 3 harnesses
- Added schema v1.2 + outcome_metrics evidence curation, incrementally collecting 279 work items and improving metric coverage 44%→50%
- Built JD-custom PDF + Zighang feedback loop; kept Juvis in pass range (92%→88%) and classified Nexon as 35% low-fit to avoid overfitting
- Hardened the Zighang recruitment-roast helper with main-scope result capture, Puppeteer element clicks, and upload-form guards to avoid sidebar-history false positives

Lessons Learned

- Learned that AI-assisted development productivity is reproducible only when runtime context, rule enforcement, evidence search, and cost observability share one operating path
- Established that document-only rules remain reference material unless read at the moment of need, so critical rules should be enforced through hooks, tests, and compiler output

---

Concert Reservation Service

2024.10 — 2024.11 · Personal

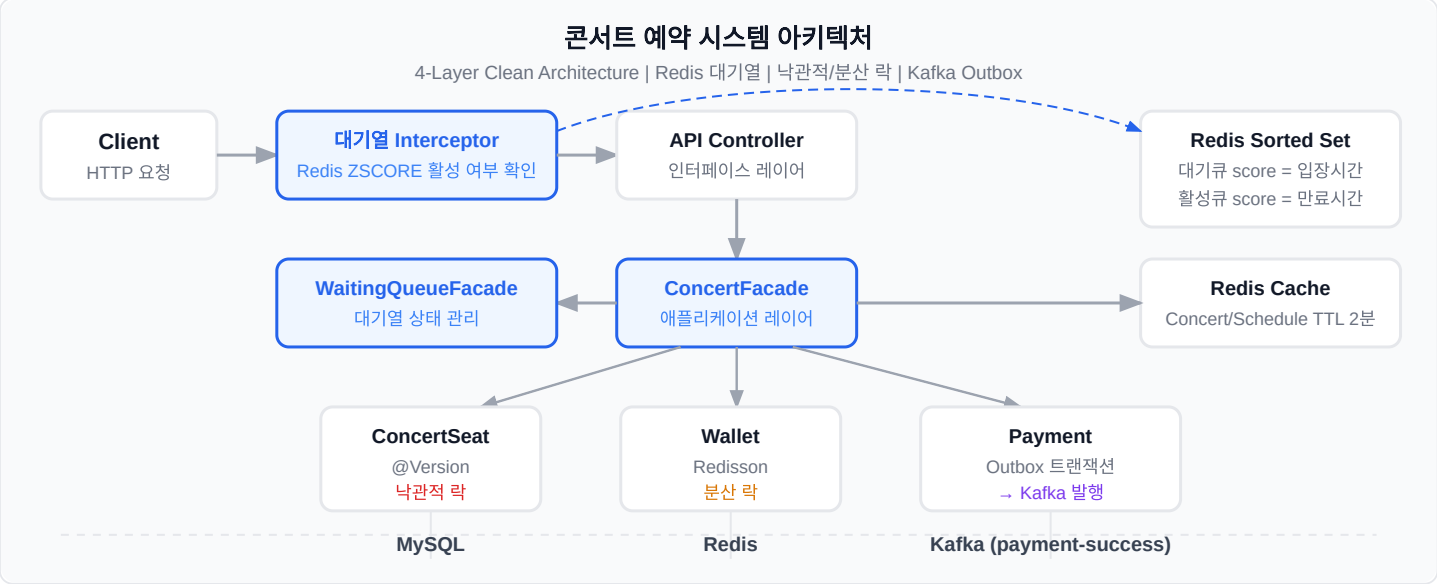
GitHub ↗ Detail ↗ JPA Pessimistic Lock, Optimistic Lock and Retry ↗ Spring Boot Concert Reservation Concurrency Analysis ↗

Distributed Lock Implementation with Spring Boot Redis ↗ Cache and Caching Strategy ↗

10K concurrent · K6 3,000 TPS · Kafka 3-broker · Prometheus+Grafana

Concert seat reservation service resolving concurrency across seat locking, payments, and point charging under 10K concurrent requests — hybrid locking strategy (optimistic + Redisson distributed lock), 15x TPS improvement via Redis caching, Kafka event-driven architecture, K6 load test-driven optimization

Java Spring Boot JPA Redis MySQL Kafka Docker K6



Performance Metrics	Before	After
Reservation latency	1,678ms	**835ms** (-50%)
Seat query TPS	100	**1,500** (15x)
DB query ratio	100%	**6%** (-94%)

Problem Solving

Risk of double-booking under 10K concurrent seat reservation requests

Compared pessimistic, optimistic, and distributed locks via K6; selected @Version optimistic lock optimal for single-row contention

→ **50% reservation latency improvement (1,678ms → 835ms), zero double-bookings**

Balance integrity corruption under concurrent point charging and payments

Analyzed infinite retry risk of optimistic lock; applied Redisson distributed lock (userWalletLock:{userId}) for per-user serialization

→ **100% balance consistency under concurrent charge/payment scenarios**

Concert/seat query TPS only 100, causing DB bottleneck at reservation open

Implemented layered caching strategy combining Redis cache with DB index optimization

→ **TPS 100 → 1,500 (15x improvement), 94% DB query reduction**

Payment/notification tightly coupled via synchronous calls after reservation

Kafka Transactional Outbox: outbox_event inserted within TX, scheduler publishes asynchronously

→ **Eliminated coupling, prevented message loss, enabled independent scaling**

Tech Decisions

- @Version optimistic lock for seat reservation: higher throughput than pessimistic lock on single-row contention, validated with K6 10K VU
- Redisson distributed lock (userWalletLock:{userId}) for payments: avoids infinite retry loops of optimistic lock in balance-critical transactions
- Redis Sorted Set queue: score=Unix timestamp serves dual purpose (entry order + expiry), O(log N) vs DB-based queue
- Transactional Outbox pattern: outbox_event row inserted within payment TX → scheduler publishes to Kafka, preventing message loss

Highlights

- 50% seat reservation response time improvement with optimistic locking (10K concurrent)
- Hybrid locking strategy: optimistic lock for seats, Redisson for payments
- 15x TPS improvement (100 → 1,500), 94% DB query reduction via Redis caching
- Kafka event-driven architecture eliminating inter-domain coupling
- Performance bottleneck identification and resolution via K6 load testing

Lessons Learned

- Understood fundamental differences in concurrency control by comparing pessimistic, optimistic, and distributed lock tradeoffs
- Learned context-driven selection over single solutions through hybrid locking strategy design per scenario

---

Ledgerly

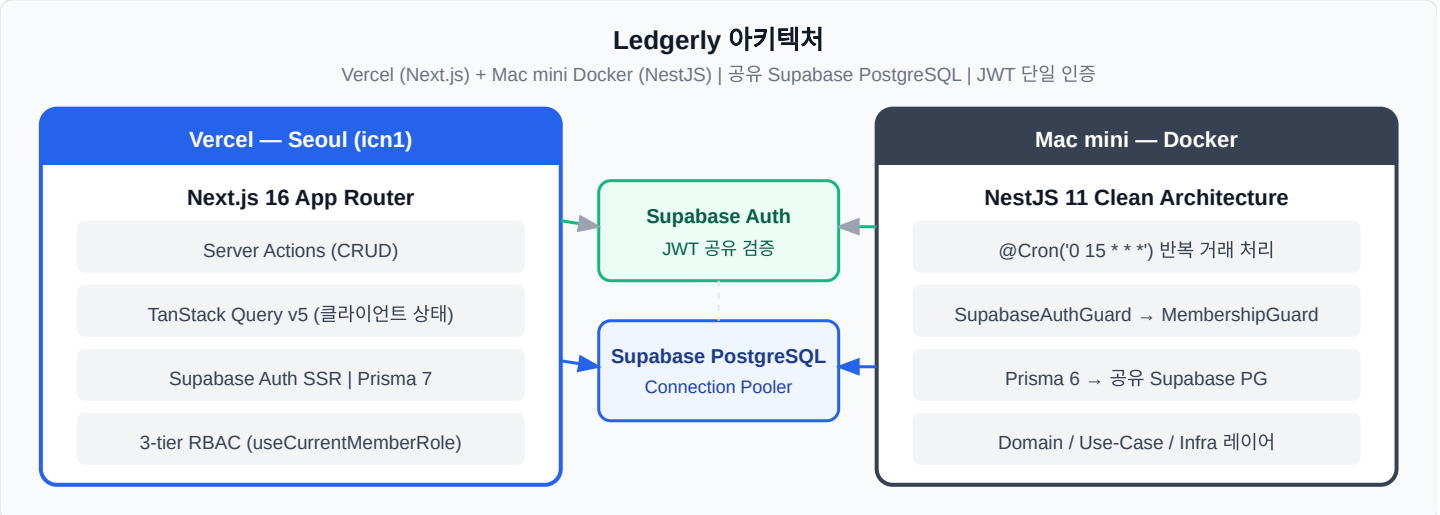
2025.08 — Present · Personal

Detail ↗

Family/org multi-tenancy · 3-tier RBAC · FE+BE separate deployment

Family/organization shared finance management app — Next.js 16 fullstack, Supabase PostgreSQL, 3-tier RBAC (OWNER/ADMIN/MEMBER), NestJS Cron recurring transaction automation, 99.45% line coverage unit tests + 300+ E2E integration tests

Next.js · React · TypeScript · Supabase · PostgreSQL · Prisma · NestJS · Tailwind CSS · TanStack Query · Zod · Vitest · Playwright · Docker · Vercel



Performance Metrics	Before	After
Vitest unit tests	-	**397건 (BE 25 + FE 31 suites)** (99.45% 커버리지)
Playwright E2E	-	**300건+ (24 specs · 5 projects)** (역할별 시나리오)

Problem Solving

Data security risk in shared finance app without role-based permission separation

Designed 3-tier RBAC, 3-layer Guard chain (SupabaseAuthGuard → MembershipGuard → SuperAdminGuard) for dual auth+authz

→ **Independent per-org data management + granular role-based access control**

Vercel Hobby Cron limit (1x/day) preventing recurring transaction automation

Separated NestJS backend via Docker on Mac mini, @nestjs/schedule daily 15:00 KST execution

→ **Daily recurring transaction automation, platform limitation bypassed**

Auth state mismatch between server and client components in SSR

Cookie-based session restoration + auth context propagation pattern with Next.js 16 + Supabase Auth SSR

→ **Seamless auth state sync across server and client components**

Manual testing unable to cover defect risk from complex RBAC + async logic

Vitest unit + Playwright E2E organized by role scenarios (Admin/Member), full-stack integration env with real BE + local Supabase

→ **99.45% line coverage, automated role-based scenario verification system**

Tech Decisions

- NestJS Docker separation: bypasses Vercel Hobby Cron limit (1/day), ensures daily 15:00 KST recurring transaction execution on Mac mini
- Supabase Auth + app-level RBAC: 3-tier guard chain — SupabaseAuthGuard (JWT) → MembershipGuard (role) → SuperAdminGuard (admin-only)
- Prisma + Supabase PG: leverages Prisma's type safety and migration tooling, routes through Supabase connection pooler for serverless connection management

Highlights

- 397 Vitest unit tests with 99.45% line coverage
- 300+ Playwright E2E tests, role-based scenarios
- 3-tier RBAC (OWNER/ADMIN/MEMBER) + Supabase RLS
- NestJS Cron recurring transaction auto (Docker)
- Full-stack integration test env (FE+BE+Supabase)

Lessons Learned

- Learned SSR authentication state management and RLS-based data access control by combining Next.js 16 App Router with Supabase Auth, understanding auth context propagation patterns between server and client components
- Experienced permission separation and data isolation strategies in organizational multi-tenancy by designing a 3-tier RBAC system (OWNER/ADMIN/MEMBER)

Daesin Logistics Dispatch Bot

2026.01 — Present · Personal

GitHub ↗ Detail ↗

Mon-Sat 06:00-20:00 hourly crawl · Blue-Green zero-downtime · SQLite single server

Logistics dispatch data service crawling with Cheerio, queryable via KakaoTalk chatbot skill server and Next.js mobile web — Clean Architecture + TSyringe DI, Express 5, Prisma SQLite, Traefik Blue-Green zero-downtime deployment

TypeScript · Express · Next.js · React · Prisma · SQLite · Cheerio · TSyringe · TanStack Query · Recharts · Docker · Traefik · Tailwind CSS · Vitest

Performance Metrics	Before	After
Crawl frequency	수동 조회	**일 15회 자동** (자동화)
Deploy downtime	수동 재시작	**0초** (Blue-Green)

Problem Solving

Operational inefficiency from manually checking dispatch status on website

Crawled EUC-KR encoded dispatch data with Cheerio + Axios, node-cron auto-sync Mon-Sat 06:00-20:00 hourly

→ **Manual lookup → 15 daily auto-crawls, real-time data availability**

Field workers had no mobile access to dispatch info without PC

Implemented Kakao i Open Builder skill server + Next.js responsive mobile web + Recharts stats dashboard

→ **Dual-channel (KakaoTalk + mobile web) enabling instant field queries**

Business logic changes required when external dependencies changed

Clean Architecture layer separation, TSyringe DI token interface binding, Value Object pattern for type-level domain rule enforcement

→ **Only infra layer changes on dependency swap, zero business logic changes**

Service downtime from manual restart deployment

Traefik file-provider Blue-Green deployment, deploy.sh automating build → health check → YAML rewrite → traffic switch

→ **Zero deploy downtime, security without Docker socket exposure**

Tech Decisions

- Clean Architecture + TSyringe DI: layer separation for swappable external dependencies (crawler, Kakao API, DB), interface binding via DI tokens
- SQLite: eliminates PostgreSQL operational overhead for single-server read-heavy workload, file-based volume mount ensures Docker portability
- Traefik file provider: traffic switching via YAML rewrite without Docker socket exposure, achieving both security and simple rollback

Highlights

- Clean Architecture + TSyringe DI layer separation design
- Cheerio crawling + node-cron hourly auto-sync (Mon-Sat)
- KakaoTalk chatbot skill server (route/vehicle/destination search)
- Traefik Blue-Green zero-downtime deploy + automated deploy scripts
- Next.js mobile web + Recharts statistics dashboard

Lessons Learned

- Learned decoupling business logic from external dependencies (crawler, DB, Kakao API) by applying Clean Architecture layer separation with TSyringe DI container
- Understood chatbot platform request/response protocols and scenario block integration by directly implementing Kakao i Open Builder skill server

Other Projects

NyamNyam WeDu — Cafeteria Menu Alert Bot (Personal) — TypeScript, Playwright, Slack Bot API, Express — Detail ↗

Hospital Job Alert Service (Personal) — NestJS, TypeScript, PostgreSQL, TypeORM — Detail ↗

StartupPool (Team) — NestJS, TypeScript, PostgreSQL, JavaScript — Detail ↗