ChatGPT

Sigma-X 자동매매 시스템 아키텍처 분석

아키텍처 개요 및 구성 요소

SYSTEM SIGMA-X는 자동매매를 위한 백엔드 시스템으로, **초기화 단계**, **실시간 처리 루프**, **스케줄러 작업**, **시뮬레이션/백테스트 모드**, **API 및 대시보드** 등의 구성 요소를 포함하도록 설계되었습니다 [1] [2]. 각 구성 요소는 모듈화되어 있으며, 환경설정/DB/로그 초기화, 실시간 데이터 수집 및 신호 처리, 주문 실행, 전략 적응, 스케줄링, 사용자 알림, 모니터링 등을 담당하는 개별 모듈로 구성됩니다. 주요 구성 요소는 다음과 같습니다:

- **초기화 단계**: 애플리케이션 시작 시 `sigma.system.initialize()`가 호출되어 환경 변수 로드와 DB 설정, ORM 테이블 생성, 로깅 시스템, 플러그인 및 사용자 설정, 모니터링 지표, 시스템 헬스 체크, 캐시, 알림 서비스, API 서비스, 이벤트 루프, 세션 관리, 추가 로깅 등의 초기화가 순차적으로 이뤄집니다 [3]. 이 단계에서 `.env` 파일 및 DB의 `system_config` 테이블로부터 설정값을 불러오고, SQLAlchemy ORM으로 필요한 테이블을 자동 생성하며, Slack/Redis/DB 연결 확인 등의 건강 체크도 수행합니다 [4] [5]. 현재 코드에서는 이 초기화 시퀀스의 골격이 구현되어 있으며(`initialize()` 함수), 각 세부 모듈(예: `plugin_loader`, `metrics`, `health_check` 등)은 주로 더미 구현으로 로그만 남기는 형태입니다 [6] [7].

- **실시간 처리 루프**: 시스템은 **24시간 무인 자동매매**를 목표로 하며, 실시간 시세 데이터를 WebSocket 등으로 받아와(`DataCollector`) Redis Pub/Sub을 통해 전체 시스템에 배포하고, TradingBot이 전략 신호를 생성하며(OrderExecutor를 통해 주문 실행) 실시간 주문 처리를 하는 흐름으로 설계되었습니다 [8]. 또한 **모의 체결(SimRunner)**, **시장 국면 탐지(RegimeDetector)** 및 **전략 파라미터 자동 조정(ParamAdjuster)**, **주문 처리 큐(RabbitMQ + OrderWorker)**, **리스크 관리**와 **알림(NotificationService)**, **품질 평가 및 피드백**, **이상치/뉴스 이벤트 처리** 등이 실시간 루프에 통합되어야 함이 명시되어 있습니다 [9] [10]. 현 코드의 `TradingBot` 클래스는 DataCollector로부터 시장 데이터를 받아(`fetch_market_data`) 전략 신호를 생성하고(`generate_signals`), OrderExecutor로 주문 실행을 호출하는 기본 루프를 제공합니다 [11]. 그러나 현재 DataCollector는 실제 시세 수집 대신 고정 값(`price: 100.0`)을 반환하고 있고 [12], WebSocket/Redis를 통한 비동기 실시간 데이터 수신 및 전파는 구현되지 않았습니다. OrderExecutor 또한 실제 주문 API 연동 대신 시뮬레이션 모드로 로그만 출력하는 더미 상태입니다 [13]. RabbitMQ 기반의 주문 큐나 OrderWorker(주문 처리 소비자) 등은 코드에 존재하지 않아, 실시간 루프의 **핵심 아키텍처**가 현재는 계획 수준에 머물러 있습니다.

- **스케줄러 작업**: 전략의 일일 최적화, 주기적인 데이터 정리, 시스템 점검, 보고서 생성, 머신러닝 기반 자동 튜닝 등 정해진 주기의 작업을 수행하기 위한 스케줄링 기능이 요구되었습니다 [14] [15]. 설계 상 APScheduler를 이용한 백그라운드 스케줄러 또는 자체 스레드 기반 간단 스케줄러를 통해 TradingBot의 주기적 실행이나 기타 작업을 예약하도록 되어 있습니다 [16] [17]. 현재 코드에서는 APScheduler 사용을 고려한 `SimpleScheduler`와 `BackgroundScheduler`의 양방향 구현이 존재하며, `start_bot_scheduler()`로 TradingBot을 일정 간격으로 실행할 수 있는 구조를 제공합니다 [18]. 그러나 이 스케줄러를 활용하여 실제로 어떤 작업들이 등록되고 언제 실행되는지에 대한 구체적 구현은 미흡합니다. (`run_bot.py`에서는 스케줄러 호출 없이 한 번만 `bot.run()`을 실행하고 종료합니다 [19].) 따라서 일일 전략 선택/튜닝이나 주간 보고 등 **스케줄러 기반 작업들은 현재 미구현** 상태로 보입니다.

- **시뮬레이션 및 백테스트 모드**: 실제 운영과 유사한 환경에서 전략을 검증하기 위해 **모의투자 시뮬레이터**와 **과거 데이터 백테스트** 모드가 요구되었습니다 [2] [20]. 설계 의도는 실시간 모드와 최대한 코드 구조를 공유하면서도, 데이터 소스만 변경하여(예: Redis 구독 대신 히스토리컬 데이터 로드) 전략을 테스트할 수 있게 하는 것입니다. 예를 들어 runner_sim.py로 **실시간 가격 이벤트**를 구독하여 시뮬레이션 모드를 돌리거나, backtest.py

CLI를 통해 기간과 전략을 입력받아 과거 데이터에 대해 TradingBot을 실행하는 흐름이 제안되었습니다 [21] [22]. 현재 구현에서는 TradingBot과 OrderExecutor에 `is_simulation` 플래그를 두어 시뮬레이션일 경우 실제 주문 대신 로그만 남기도록 하고 있어**일부 통합을 시도**하고 있습니다 [13]. 그러나 **전용 시뮬레이션 실행기(SimulatorExecutor)**나 **히스토리컬 데이터 로더** 등의 코드는 존재하지 않으며, 백테스트 결과를 요약/시각화하거나 비교 분석하는 기능도 구현 전입니다. 다시 말해, 현 단계의 코드는 **실시간 루프 로직을 거의 동일하게 재사용하여 시뮬레이션/백테스트를 수행할 수 있는 준비**는 갖추었지만, 이를 위한 별도 실행 모드나 데이터 입출력 경로가 마련되어 있지 않습니다.

- **API 및 대시보드**: 사용자에게 **실시간 데이터 및 과거 성과를 시각화**하고, 봇의 동작을 제어할 수 있는 UI/API 계층이 요구되었습니다 [23]. 설계는 FastAPI 기반의 REST API와 WebSocket 실시간 데이터 스트림, React 기반 프론트엔드 대시보드를 통해 **전략 제어, 시세/성과 조회, 알림 확인, 사용자 설정 관리** 등을 제공하도록 명시하고 있습니다 [23] [24]. 현재 백엔드 코드에는 FastAPI 앱 초기화와 `/metrics` 엔드포인트 노출 부분이 구현되어 있으며, `uvicorn`을 별도 데몬 스레드로 구동시켜 메인 봇과 병행 실행하도록 구성했습니다 [25]. 이를 통해 운영 중에도 Prometheus가 수집하는 메트릭 데이터를 노출할 수 있습니다. 그러나 그 외에 **전략 시작/중지, 데이터 제공 등의 REST API나 WebSocket** 엔드포인트, **JWT 인증** 등은 아직 구현되지 않았습니다. React 대시보드 역시 이 저장소에는 포함되어 있지 않아, 현재는 **백엔드의 최소한의 상태 확인용 API**만 존재하는 상황입니다.

이상의 구조를 종합하면, Sigma-X 시스템은 **모듈화된 계층 구조**(config/db/core/data/system/utils/web 등)로 설계되어 있으며, 각 기능을 개별 컴포넌트로 분리함으로써 **확장성과 유연성**을 확보하려 한 모습입니다. 그러나 실제 코드 구현은 **설계상 준비된 모듈들의 골격만 존재**하고, 실시간 자동매매에 필요한 핵심 로직 상당수가 더미이거나 미구현 상태입니다. 아래에서는 이러한 현재 아키텍처가 제시된 **평가 기준**에 부합하는지, 그리고 구체적으로 어떤 부분이 보완되어야 하는지를 분석합니다.

실시간 자동매매에 대한 적합성

실시간성은 자동매매 시스템의 핵심 요건입니다. Sigma-X의 설계는 웹소켓을 통한 실시간 시세 수집과 Redis Pub/Sub, RabbitMQ 큐를 활용한 **비동기 이벤트 기반 처리**를 추구하고 있어, 개념적으로는 **실시간 데이터 대응에 적합한 구조**로 보입니다 [8]. 이러한 아키텍처에서는 데이터 수집, 신호 생성, 주문 실행이 **헐겁게 결합(loose coupling)** 되어 각 구성요소가 병렬적으로 동작하므로, 시세 변동에 대한 지연(latency)을 최소화하고 병목을 줄일 수 있다는 장점이 있습니다. 예를 들어, 시세 수집기가 새로운 가격을 수신하면 즉시 Redis 채널을 통해 배포하고, 여러 전략 봇(TradingBot)이 구독을 통해 거의 동시다발적으로 신호를 생성할 수 있습니다. 주문 실행도 RabbitMQ 큐에 신호를 넣어 비동기로 처리하면, **메인 스레드가 다음 데이터 처리로 곧바로 넘어갈 수 있어** 전체 파이프라인의 처리율이 높아집니다.

그러나 **현재 구현 수준**에서는 이러한 실시간성 장점이 발휘되지 못하고 있습니다. TradingBot은 내부에서 **동기적 루프**를 돌며 매번 하나의 데이터 포인트를 가져와 신호를 생성하고, 순차적으로 주문 실행을 호출하는 구조입니다 [11]. **WebSocket 기반**의 지속적인 데이터 스트림 수집이나 Redis 퍼블리시/구독 메커니즘은 아직 구현되지 않아, **실시간 데이터 피드**가 없는 상태입니다. DataCollector가 매 호출마다 고정값을 반환하는 현재 구조로는 실시간 시세 변동을 반영할 수 없으며 [26], TradingBot의 `run()`도 기본 1회 iteration만 수행하도록 되어있어(별도 반복 호출 없을 경우) **지속적인 24/7 실행 루프가 보장되지 않습니다**. 이는 설계 목표인 "24시간 무인 자동매매 운영"에 미치지 못하는 부분입니다 [27].

또한 이벤트 기반이 아닌 **폴링 방식**(loop 내에서 fetch 호출)이므로 시장 상황 급변 시 신속 대응이 어렵고, 단일 스레드에서 일련의 작업을 처리하므로 데이터 수집과 전략 실행이 **동시에 수행되지 못하는** 한계가 있습니다. 실시간성을 높이려면 **비동기 I/O 또는 멀티스레드/멀티프로세스** 활용이 필수인데, 현재 event_loop나 Async 처리기가 구현되지 않아(더미로 존재) [7] 개선이 필요합니다.

요약하면, Sigma-X의 **설계 방향은 실시간 자동매매에 적합하도록 잘 잡혀 있지만 구현은 미흡**합니다. 아키텍처 측면에서 WebSocket+Redis+RabbitMQ 조합은 고빈도 데이터 처리에 어울리는 비동기 이벤트 주도 모델이나, 구현 부재로 성능을 논하기 어려운 상태입니다. 실시간 대응 측면에서 우려되는 추가 지점은 다음과 같습니다:

- **데이터 수집 지연**: 실제 거래소 API (예: 바이낸스 WebSocket) 연결 시 네트워크 지연, 재연결 로직 등이 필요합니다. 현재 이러한 부분이 전혀 고려되지 않았으므로, 향후 구현 시 **재연결 및 지연 최소화** 전략이 필요합니다.
- **주문 실행 지연**: OrderExecutor가 실제 API 연동 시 REST API 호출 지연, 요청 제한(rate limit) 대응 등이 필연적인데, 이를 비동기로 다루지 않으면 전략 루프가 지연될 수 있습니다. RabbitMQ 기반 분산 처리가 이를 완화할 예정이지만 아직 미구현입니다.
- **실시간 모니터링**: Prometheus 기반 메트릭 수집이 초기화 시 활성화되지만 [28], 실시간으로 시스템 부하나 지연을 추적하는 지표는 정의되어 있지 않습니다. 추후 **tick 처리 속도, 큐 대기 시간, API 응답 시간** 등의 메트릭을 수집하면 실시간성 보장에 도움이 될 것입니다.

결론적으로, **구조적으로는 실시간 처리를 염두에 두고 모듈을 나누고 있으나** 실제 동작은 아직 동기적이고 단순한 형태입니다. **현재 구조를 비동기로 개선**하고, 설계된 WebSocket->Redis->RabbitMQ 파이프라인을 구현함으로써 요구되는 실시간 성능을 충족시킬 수 있을 것입니다.

병렬성, 확장성 및 신뢰성

Sigma-X는 설계 단계에서 **다중 스레드/프로세스와 메시지 큐**를 활용하여 병렬성과 확장성을 확보하려고 하고 있습니다. 예를 들어, FastAPI 웹 서버를 메인 봇과 별도 스레드로 실행하도록 한 점 [25], APScheduler(또는 자체 스레드)로 주기 작업을 백그라운드에서 수행하도록 한 점 [18] 은 **메인 전략 실행과 병렬로 다른 작업을 처리**하려는 의도가 엿보입니다. RabbitMQ를 통한 주문 처리 비동분산도 설계에 포함되어 있는데, 이는 주문 처리 부분을 **독립 프로세스(또는 워커)**로 분리하여 TradingBot과 병렬로 동작시킴으로써 전체 처리량을 높이고 한 컴포넌트의 지연이 전체 시스템에 영향을 덜 주도록 하는 훌륭한 패턴입니다 [9]. Redis Pub/Sub 역시 **다중 구독자 패턴**을 가능하게 하여, 향후 전략 인스턴스를 여러 개 늘리거나, 모니터링 프로세스 등 여러 소비자가 동일 시세 스트림을 병렬로 받아 처리할 수 있는 토대를 제공합니다.

확장성 측면에서, 이론적으로 Sigma-X 아키텍처는 **수평 확장**을 지원할 수 있습니다. 예를 들어 주문 처리 워커를 여러 개 두어 RabbitMQ 큐를 병렬 소비하게 하거나, 서로 다른 전략 봇 프로세스를 여러 개 띄워 다양한 코인/전략을 동시에 운용하는 구성이 가능합니다. 모든 구성요소가 느슨하게 연결되어 있기 때문에, 특정 모듈만 별도로 확장(스케일 아웃)하거나 교체하는 것도 비교적 수월할 것으로 판단됩니다. 데이터베이스(PostgreSQL + TimescaleDB 권장)는 중앙 저장소로써 여러 프로세스가 공동 이용하고, Redis/RabbitMQ는 분산 메시징으로 데이터 일관성을 유지하면서 확장을 돕는 구조입니다 [29]. 이러한 분산 시스템 구성은 **신뢰성**에도 도움을 줍니다. 한 모듈에 문제가 발생해도 메시지 큐에 남은 작업을 후속 프로세스가 이어 처리하거나, 다른 인스턴스가 역할을 대체하는 형태로 **폴트 톨러런스(fault-tolerance)**를 구축할 수 있기 때문입니다.

하지만 **현재 구현 상태에서는 병렬성/확장성의 이점이 아직 현실화되지 않았습니다.** RabbitMQ 연동이나 멀티스레딩이 핵심 기능에서는 사용되고 있지 않고, TradingBot도 단일 스레드 루프로 동작합니다. FastAPI 서버 스레드를 띄우는 것 외에는 병렬 처리 요소가 거의 없으며, 실제 RabbitMQ를 사용할 OrderWorker 코드가 없어 **주문 처리는 여전히 TradingBot 내부(OrderExecutor)에 묶여있는 형태**입니다 [13]. 이는 한정된 스레드 내에서 일련의 작업이 순차로 처리됨을 의미하므로, 다수의 거래나 다전략 동시 운용 시 병목이 생길 수 있습니다. **멀티코어 활용**도 거의 되지 않으며, 현재 구조로는 하나의 프로세스가 모든 일을 처리하기 때문에 CPU 1개 이상의 효과를 내기 어렵습니다.

신뢰성에 관련해서도, 개선 여지가 있습니다. 현 구현의 `health_check`는 애플리케이션 시작 시 **DB, Redis 연결 및 Slack 설정을 한번 확인**하는 데 그칩니다 [4] [5]. 운영 중 지속적인 헬스 모니터링(예: 30초마다 DB/큐 상태 점검)이나 오류 발생 시 자동 조치(프로세스 재시작, 대기 등의 메커니즘)는 보이지 않습니다. 예를 들어, WebSocket 연결이 끊어지거나 주문 API 호출에 실패하는 경우를 고려한 **재시도/예외 처리 로직**이 필요하지만, 아직 그런 부분이 코드에 추

가되지 않았습니다. “헬스체크 주기화: 30초 단위, Slack 알림 연동” 등의 계획 [30] 이 문서화되어 있으므로, 이러한 주기적 헬스체크 및 알림을 구현한다면 문제가 발생해도 관리자가 신속히 인지하고 대처할 수 있을 것입니다.

정리하면, **설계상 병렬성과 확장성을 얻을 수 있는 아키텍처를 채택하고 있으나 구현이 따라주지 않아 현재는 단일 프로세스 성능/신뢰성에 제한**됩니다. 아키텍처의 강점을 살리기 위해 다음이 필요합니다: 핵심 컴포넌트의 멀티스레드/프로세스화, 메시지 큐 활용 구현, 모듈별 예외 처리 및 복구 로직 추가, 그리고 다중 인스턴스 운영을 고려한 세션 관리나 중복 방지 장치. 예컨대, RabbitMQ 큐의 지속성과 재시도 메커니즘을 활용하고, SlackNotifier 등을 통해 오류를 즉시 통보하며, `RotatingFileHandler`로 **로컬 로그를 관리**하여 장애 전후 로그를 보존하는 등의 신뢰성 강화가 요구됩니다 [31].

플러그인 전략 관리 및 업데이트 자동화

전략의 플러그인화와 **자동 업데이트**는 Sigma-X가 **유연성**을 갖추기 위해 내세운 중요한 특징입니다. 사용자가 새로운 전략을 손쉽게 추가하거나 교체하고, 시장 상황에 맞게 전략을 자동 조정하는 기능이 이에 해당합니다. 설계 문서에는 “외부 플러그인 및 사용자 선호 설정 불러오기” 단계가 초기화에 포함되어 있고 [32], TradingBot이 동작 중 **전략 파라미터 자동 조정(ParamAdjuster)**을 수행하며, 일별 **전략 선택/튜닝(StrategySelector, MLModule)** 작업이 스케줄러에 포함되어 있습니다 [33] [34]. 이는 전략 로직을 하드코딩하지 않고 **모듈화/파라미터화**하여, 플러그인처럼 주입하고 운용 중 변경할 수 있도록 고려한 것입니다.

현재 구현을 보면, **전략 플러그인 로더**(`sigma.system.plugin_loader`)가 존재하지만 내부 구현은 단순히 "플러그인 로드 완료"라는 로그 출력에 그치고 있습니다 [6]. 실제로는 기본 전략(`BaseStrategy` 추상클래스)과 예시 더미 전략(`DummyStrategy`)만 정의되어 있을 뿐 [35] [36], **여러 개의 외부 전략을 동적으로 탐색/로딩**하는 로직은 아직 마련되지 않았습니다. 따라서 현 시점에서 새로운 전략을 추가하려면 코드에 직접 클래스를 추가하고 TradingBot 생성시 명시해야 하며, 이는 플러그인 관리라고 부르기 어렵습니다. 추가로, 전략을 실행 중에 **핫 스왑(hot swap)**하거나 조정하는 기능도 구현돼 있지 않습니다 (예: 시장 국면에 따라 전략 A에서 B로 변경하는 로직 등).

그러나 일부 **전략 업데이트 자동화 기반**은 갖춰져 있습니다. `ParamAdjuster`는 주어진 파라미터 값을 DB의 `strategy_param` 테이블에 저장/갱신하는 기능을 가지고 있어 [37], **전략 매개변수를 중앙 DB에 기록**하고 참조할 수 있습니다. 이와 함께 `RegimeDetector`가 단순하게나마 시장 추세(bull/bear)를 판단하는 메서드를 제공하고 있고 [38], `QualityAssessment`와 `FeedbackMechanism` 클래스로 전략 성과를 기록/평가할 수 있게 되어 있습니다 [39] [40]. 이는 향후 **전략 성능을 모니터링하여 일정 기준 미달 시 알림을 주거나(품질 평가), 장기간의 성과를 축적하여 학습(피드백 메커니즘)**하는 토대를 마련한 것으로 평가됩니다. 실제로 Improvement 문서에서도 “전략 성과 평가, 기준 미달 시 경고”, “장기 학습용 로그 저장” 등이 제시되어 있는데 [41], 이러한 요구에 맞춰 클래스 레벨에서는 준비를 해 둔 상황입니다.

다만, **이러한 전략 적응 메커니즘이 TradingBot 실행 흐름에 통합되지 않았다**는 것이 문제입니다. 예를 들어, TradingBot이 매 루프마다 RegimeDetector를 사용하여 현재 시장 국면을 확인하고, ParamAdjuster를 통해 해당 국면에 맞는 전략 파라미터를 업데이트하는 로직이 있어야 하는데, 코드에는 그런 호출이 없습니다. StrategyParam DB에 값이 저장되더라도 **그 값을 읽어 전략 로직에 반영하는 부분**도 존재하지 않습니다. (예컨대 DummyStrategy는 고정된 신호를 내보낼 뿐 외부 파라미터를 참고하지 않습니다.) 또한, StrategySelector나 MLModule 등이 아예 미구현이므로, **일별로 최적의 전략을 선택/튜닝**하거나 **머신러닝으로 전략을 보정**하는 자동화는 현재 단계에서는 구현되지 않은 아이디어 수준입니다.

플러그인 전략 관리를 제대로 구현하려면, `plugin_loader.load_plugins()`가 **설정된 디렉터리나 엔트리 포인트에서 전략 모듈들을 동적 로드**하도록 개선되어야 합니다. 예컨대 `strategies/` 폴더 내의 모든 `.py` 파일을 읽어 BaseStrategy를 상속한 클래스를 찾아 등록하거나, `SystemConfig`에 사용자가 활성화하려는 전략 이름을 키로 저장해 두고 해당 클래스만 로딩하는 방법을 생각해볼 수 있습니다. 또한, 로딩된 여러 전략을 식별하고 관리하기 위한 **전략 레지스트리** 혹은 **팩토리 패턴**도 고려할 수 있습니다. 이렇게 하면 플러그인 구조로 새로운 전략 추가/제거가 유

연해지고, 일부 전략의 오류가 다른 부분에 영향을 주는 것을 방지하거나 (예: 개별 전략 격리), 특정 전략만 재시작하는 등의 운영도 가능해집니다.

업데이트 자동화 측면에서는, 우선 **전략 파라미터의 실시간 반영** 기능을 들 수 있습니다. 현재 ParamAdjuster가 DB에 값을 쓰는 것은 구현되었으므로, **전략 실행 시마다 DB나 캐시로부터 최신 파라미터를 불러와 적용**하도록 전략 코드를 수정하면 비교적 쉽게 달성할 수 있습니다. 더 나아가, 스케줄러를 통해 **정기적인 전략 최적화 작업** (예: 과거 1달 데이터로 백테스트해 가장 좋은 파라미터 산출 -> DB 갱신)을 자동화하면 사람이 개입하지 않아도 전략이 조정되는 **자동 튜닝 시스템**이 완성됩니다 [16] . 이러한 기능은 아직 코드에 없지만, 시스템 아키텍처상 APScheduler와 ParamAdjuster 등을 활용하면 충분히 구현 가능할 것입니다. 단, 머신러닝을 통한 자동 튜닝(예: 강화학습이나 유전 알고리즘으로 전략 진화)은 복잡한 작업이므로 별도의 모듈/리소스 분리가 필요할 수 있습니다. (예를 들어, ML 튜닝은 오프라인에서 대량 연산 후 결과만 시스템에 반영하거나, 또는 낮은 빈도로 수행하도록 스케줄링.)

결론적으로 Sigma-X는 **전략 플러그인화와 자동 업데이트를 지향하는 유연한 구조**를 구상하였으나, 핵심 구현이 빠져 있어 현재는 단일 고정 전략에 머물러 있습니다. 설계대로 **플러그인 로딩, 파라미터 동적 적용, 전략 평가/교체 자동화**가 구현된다면 다양한 전략 실험과 신속한 전략 대응이 가능해져 시스템의 **적응성과 유연성**이 크게 향상될 것입니다.

시뮬레이션 및 백테스트 환경의 통일성

자동매매 시스템에서 **시뮬레이션(모의투자) 및 백테스트 환경**은 실제 운용과 최대한 유사해야, 검증된 전략이 실계좌에서도 동일하게 동작할 수 있습니다. Sigma-X의 설계는 이를 고려하여 **실시간 운용 루프와 시뮬레이터/백테스트가 구조적으로 통일**되도록 하고 있습니다. 구체적으로, **StrategyManager/TradingBot, 시그널 생성 로직, 주문 실행 모듈** 등을 실시간 모드와 시뮬레이션/백테스트 모드에서 공유하고, 차이는 **데이터 공급원과 체결 처리 방식만 다르게** 설정하는 접근입니다 [21] [22] . 이렇게 하면 전략 코드 한 번 작성으로 **실거래와 테스트에 일관성**을 유지할 수 있고, 백테스트를 통해 발견한 문제가 실제 운영에도 동일하게 적용되어 신뢰도를 높일 수 있습니다.

현재 Sigma-X 구현을 보면, **통일성 측면에서 긍정적인 부분**과 **개선이 필요한 부분**이 혼재합니다. 긍정적인 면은, 앞서 언급한 대로 TradingBot/Strategy/Executor 구조를 **재사용**하여 시뮬레이션을 할 수 있게 설계했다는 점입니다. OrderExecutor에 `is_simulation` 옵션을 둬서 실거래 시에는 실제 API 호출로, 시뮬레이션 시에는 로그 출력으로 분기하고 있으며 [13] , MarketData 수집기도 실제 구현시 실시간/백테스트 모드에 따라 다른 소스에서 데이터를 가져오도록 추상화할 여지가 있습니다. 예를 들어, `fetch_market_data()`를 라이브 모드에선 거래소 API를 통해, 백테스트 모드에선 CSV 파일이나 DB로부터 히스토리 데이터를 읽어오는 식으로 확장할 수 있습니다 (현재는 더미지만 함수 구조는 존재함). 또한, Strategy 자체도 데이터 포맷만 같다면 실시간/백테스트를 가리지 않고 동일하게 동작할 것이므로, **전략 로직의 재사용성**은 확보된 셈입니다.

그러나 **현재 통일성이 부족한 부분**은, 시뮬레이션/백테스트 시 실제 거래와 동일한 결과를 얻기 위한 **주문 체결/체인 처리 로직**이 전무하다는 것입니다. 실거래 모드에서는 (미래 구현 상) OrderExecutor -> RabbitMQ -> OrderWorker -> 거래소 API -> DB 저장의 흐름으로 체결이 이뤄질 예정인 반면 [42] [22] , 시뮬레이션 모드에서는 **SimulatorExecutor**가 신호를 받아 가상의 체결을 즉시 생성하고, 그 결과를 별도의 `sim_orders`나 `simulation_results` 테이블에 기록하는 흐름을 의도했습니다 [43] [44] . 하지만 이러한 **SimulatorExecutor 및 결과 저장/분석 모듈**이 구현되지 않아, 현재 시뮬레이션을 수행하면 단순히 OrderExecutor에서 "[SIM] execute BUY/SELL" 로그만 남을 뿐 포트폴리오 잔고, 포지션, 수익률 등의 **결과 상태를 추적하지 못합니다**. 즉, 모의 체결과 실제 체결 결과의 비교는커녕, 모의 투자 자체의 성과도 계산되지 않습니다. 백테스트 모드 역시 히스토리컬 데이터를 입력받는 진입점이 없고, 가령 1년치 데이터를 빠르게 돌려본다거나 하는 기능이 없습니다. **백테스트 결과를 요약/시각화, 성능 비교** 등의 요구사항 [45] 을 충족하려면, 별도의 백테스트 모듈이 TradingBot이나 Strategy를 호출하면서 과거 데이터를 루프 돌리고, 그 결과를 저장/분석해야 하는데 이 부분이 빠져 있습니다.

구조 통일성을 높이기 위해서는, 단일 코드베이스로 라이브/시뮬레이션/백테스트를 모두 커버하되 **모드에 따라 입출력 채널만 전환**하는 형태로 구현하는 것이 이상적입니다. 현재 초기화 과정에서 RUN_MODE 등을 설정하여 해당 모드에서만 필요한 서비스들을 올리는 방식도 생각해볼 수 있습니다. 예를 들어, 실시간 운용 모드에서는 WebSocket 수집+RabbitMQ 실행, 시뮬레이션 모드에서는 히스토리 데이터 로드+SimulatorExecutor 실행, 백테스트 모드에서는 CLI

입력으로 기간 설정 후 히스토리 데이터 일괄 처리… 등의 분기가 필요합니다. 이때도 Strategy나 TradingBot, ParamAdjuster 등의 코어 로직은 최대한 **공유**해야 합니다.

또 하나 고려할 점은 **데이터 및 로직의 일치**입니다. 백테스트용 데이터가 실시간 모드에서 사용하는 데이터와 다른 소스이면(예: 실시간은 거래소 API, 백테스트는 자체 DB) 체결 방식이나 슬리피지 등 미묘한 차이가 생길 수 있습니다. Sigma-X는 이를 의식해서 TimescaleDB 등의 활용을 제안하고 있습니다 [46]. 즉, 실시간으로는 Redis/TimescaleDB에 시세를 모두 저장해 놓고, 백테스트 시 그 DB를 그대로 읽어 사용하면 데이터 불일치가 없을 것입니다. 현재는 MarketData 테이블 정도만 정의되어 있으나 [47], 추후 **Tick 데이터의 저장 및 재활용**이 구현된다면 라이브/시뮬레이션 간 데이터 동등성이 향상됩니다. 또한, 실제 주문과 시뮬레이션 주문의 기록 형식을 맞추는 것도 통일성에 중요합니다. 예컨대, 실제는 orders 테이블, 시뮬레이션은 sim_orders 테이블 두 개로 관리하기보단, 하나의 orders 테이블에 `is_simulation` 플래그를 두고 관리하면 동일한 대시보드에서 실계좌와 모의계좌의 실적을 비교하기 수월해집니다 (필요 시 뷰나 파티셔닝으로 분리). 현재 orders 테이블이 아예 없으므로 설계만의 문제이긴 하나, 구현 시 이런 점을 염두에 둘 수 있습니다.

요약하면, Sigma-X는 **코드 구조의 재사용성 측면에서는 운용/시뮬레이션 통일성에 유리한 설계를 갖고 있으나**, 아직 체결/분석 부분 구현 부재로 **실질적 통일성은 확보 못한** 상태입니다. 실제 구현을 진행할 때 **동일한 전략 로직 + 유사한 데이터 흐름**을 유지하도록 신경쓴다면, **백테스트 결과와 실매매 결과의 괴리**를 최소화할 수 있을 것입니다. 이를 위해 **시뮬레이터 모듈 구현**, **히스토리컬 데이터 관리**, **성과 분석 자동화** 등을 향후 개발 시 반영해야 합니다.

데이터 흐름, 저장소 및 메시지 큐 활용의 일관성

Sigma-X 아키텍처에서 데이터는 **외부 -> 캐시/메시지큐 -> 처리 -> 영구저장(DB) & 알림**의 흐름을 따르도록 구상되었습니다. 이 흐름의 각 단계는 **역할 분담이 명확**한데, 예를 들면 **Redis**는 **실시간성 요구가 높은 데이터의 단기 저장/공유**(예: 최신 호가, 임시 계산값)를 담당하고, **RabbitMQ**는 **비동기 작업 처리와 구성요소 간 연결(decoupling)** 역할을, **PostgreSQL/TimescaleDB**는 **영구적 데이터 보관과 질의** 역할을 맡도록 한 것입니다 [29] [48]. 이러한 역할 구분이 잘 지켜지면 시스템은 **일관성과 효율성**을 얻습니다. 예를 들어, 초당 수십 회 오는 시세 틱은 Redis에 넣고 필요한 곳에서 subscribe하여 사용하되, 일정 주기로만 DB에 배치 저장함으로써 DB 부하를 낮추고, 대량 데이터는 TimescaleDB로 아카이빙하는 방식이 가능합니다. 한편 주문 체결 이벤트나 알림 등은 발생 빈도가 낮으므로 RabbitMQ를 통해 안전하게 처리한 뒤 DB에 반드시 기록하여 **사후 분석 및 추적 가능성**을 확보합니다 [46]. 이렇듯 **데이터의 속성별로 저장소와 전달 매체를 달리**함으로써 각 요소가 가장 잘 하는 역할만 담당하게 되고, 전체 시스템 성능과 일관성이 향상됩니다.

현재 구현에서는 이러한 **데이터 흐름의 큰 그림은 그려져 있으나, 실제 활용은 매우 제한적**입니다. 우선 **Redis**의 활용: `sigma.system.cache` 모듈은 더미로서 Redis 초기화 로직이 없고 [49], 시스템 어디에서도 Redis Pub/Sub 또는 캐시 get/set을 하고 있지 않습니다. 따라서 현재는 **Redis가 사실상 미사용** 상태입니다. 반면 **DB**는 초기화 시 자동 테이블 생성, 설정 값 로드 등에 쓰이고 있습니다. `SystemConfig`, `StrategyParam`, `Alert`, `MarketData`의 4개 테이블이 정의되어 있는데 [50] [51], **주문/포지션 관련 테이블은 미정의**인 상태입니다. Alert 테이블은 NotificationService.notify 호출 시 알림 메시지를 DB에 저장하는 데 사용되고 있어 데이터 영구 저장으로 **일관성있게 활용**되고 있습니다 [52]. SystemConfig와 StrategyParam도 load_db_config나 ParamAdjuster 등을 통해 DB 중심으로 설정/파라미터를 관리하려는 의도가 엿보입니다. 이러한 **설정/알림/파라미터의 DB화**는 일관성 면에서 좋은 접근이며, 코드에서도 실제 사용되고 있는 부분이라 현 단계에서는 긍정적으로 평가됩니다.

한편, **RabbitMQ**나 **메시지 큐**의 활용은 아직 전혀 없습니다. RabbitMQ 관련 코드나 설정은 존재하지 않고, OrderExecutor가 곧바로 주문을 (로그로) 처리하기 때문에, **주문 관련 데이터 흐름이 단일 프로세스 내부로 머물러 있습니다**. 향후 RabbitMQ를 도입하면 OrderWorker에서 주문 체결 결과(성공/실패, 체결가격 등)를 받아 DB에 기록하거나, 실패 시 재시도하는 로직을 분리 구현할 수 있을 것입니다. 설계 상 OrderWorker -> DB 저장 및 PaymentProcessor(정산)까지 염두에 두고 있으므로 [53], 해당 부분이 구현되면 **주문 데이터의 흐름**도 완결될 것으로 보입니다.

데이터 일관성 측면에서 현재 우려되는 부분은, **MarketData (시세 데이터) 저장 전략**입니다. MarketData 테이블이 정의는 되어있으나 아직 활용되지 않고 있는데 [47], 실시간 시세를 얼마나 자주 이 테이블에 넣을지 정책이 필요합니다. 모든 틱을 DB에 넣으면 DB 부하와 저장용량 문제가 생길 수 있으므로, Redis에 실시간 저장 -> 일정 기간 누적 후 묶어서 DB 저장, 또는 중요한 시점 봉자료만 DB 저장 등의 방식이 고려됩니다. TimescaleDB를 사용하면 다소 완화되겠지만 근본적으로 초당 수십건 이상을 매초 commit하는 것은 피해야 하므로, **Redis와 DB간 데이터 이관 전략**이 수립되어야 합니다. 현재 구현에는 그런 부분이 없어, 실시간 처리와 데이터 영속성 간의 균형을 맞추는 로직이 빠져 있습니다.

또 하나 일관성 이슈는 **전략 파라미터/설정의 일원화**입니다. 현 구조에서는 `.env`로부터 로드한 환경변수와 DB의 SystemConfig 모두 설정으로 사용되고 있습니다 [54]. improvement 문서에서는 `.env` 의존성을 최소화하고 DB를 단일 설정 소스로 삼으려고 제안하고 있으며 [55], 실제 `config_loader.load_db_config()` 구현도 DB에서 모든 설정을 읽어들이도록 하고 있습니다 [56]. 이는 매우 바람직한 방향입니다. 다만 아직 Slack 토큰/채널 등 몇몇 값은 .env에 남아 있는 것으로 보이는데, 이러한 부분도 **DB에 일원화**하고 UI/API를 통해 수정할 수 있도록 하면 실시간으로 시스템 설정을 변경하기 용이할 것입니다. NotificationService도 현재 SlackNotifier에 강하게 묶여 있지만, 향후 Email 등 다른 알림 채널을 추가하려면 설정값에 따라 분기하도록 개선해야 할 것입니다 [57].

마지막으로, **메시지 큐와 캐시의 일관적 활용** 측면에서는, Redis와 RabbitMQ를 **정확한 목적에 맞게 사용**하도록 역할을 구분하는 것이 중요합니다. 설계에는 Redis Pub/Sub과 RabbitMQ Queue가 모두 등장하는데, 때로는 이 둘의 역할이 겹치거나 혼동될 수 있습니다. 예를 들어, **주문 신호 전송**에 RabbitMQ를 쓰기로 했으면 동일 신호를 Redis Pub/Sub으로도 보내는 이중 처리는 피하고, 반대로 **시세 데이터 전파**에는 RabbitMQ 대신 Redis를 쓰는 식으로 구분해야 합니다. 현재 코드에선 구현이 안 되어 있어 충돌은 없지만, 향후 개발 시 이러한 **일관성 있는 설계 원칙**을 준수하는 것이 좋습니다. 이는 시스템 이해도와 디버깅 용이성을 높입니다.

전체적으로, Sigma-X는 데이터가 생성->처리->저장되는 **전 과정의 흐름을 고려하여 아키텍처를 설계**했으나, 실제 코드는 그 **일부분만 반영**하고 있습니다. DB 중심의 설정/로그 저장은 어느 정도 구현됐으나 실시간 캐시/큐 활용과 시세/주문 데이터 저장은 미진합니다. 앞으로 구현이 진행되면서 **데이터 파이프라인**이 설계대로 완성된다면, 시스템은 **실시간성과 영속성, 일관성을 모두 갖춘** 탄탄한 구조를 갖추게 될 것입니다.

구현 상태와 설계 요구사항 반영 여부

이 섹션에서는 앞서 언급한 요구사항들이 실제 코드에 어느 정도 반영되어 있는지 정리합니다. 앞서 살펴본 대로, Sigma-X는 **설계 문서상 많은 기능을 계획**해 두었으나 **현 시점 구현은 골격 위주의 1차 버전**입니다. 주요 기능별 구현 현황을 요구사항 대비 간략 평가하면 다음과 같습니다:

- **환경설정 DB화 및 초기화** – **부분 구현**: `.env`의 환경 변수와 DB의 `system_config` 테이블을 함께 활용하여 설정을 로드하는 기능이 구현되었습니다 [54]. 초기화(`initialize()`) 시 환경/DB설정 로드, 테이블 생성, 로깅, Slack/Redis/DB 헬스체크, Prometheus metrics 서버 기동, FastAPI 스레드 기동 등 핵심 단계들이 순서대로 호출되며 요구된 플로우차트 흐름을 대체로 따릅니다 [3]. 다만 plugin, cache 등은 더미로 처리되고, ConfigSync(설정 동기화)나 LogRotator(로그 관리) 등 고급 기능은 미포함입니다.

- **실시간 데이터 수집 및 처리** – **미구현 (설계만)**: WebSocket을 통한 실시간 시세 수집, Redis Pub/Sub에 의한 브로드캐스트는 코드에 전혀 구현이 없습니다. TradingBot은 현재 더미 데이터로 동작하며 WebSocket 연결/재접속 로직도 없습니다. **Redis 캐시 활용** 및 **RabbitMQ 기반 주문 비동처리**도 미구현입니다.

- **자동매매 루프 (TradingBot + OrderExecutor)** – **부분 구현**: TradingBot 클래스 및 OrderExecutor의 기본 뼈대는 존재하여 신호 생성->주문 실행의 한 사이클 처리는 가능하게 만들어졌습니다 [11]. 그러나 OrderExecutor의 실제 주문 로직은 TODO로 남아있고 [13], OrderWorker 등 분리된 구성요소는 없습니다. RiskManager, NewsHandler, AnomalyDetector 등의 보조 모듈도 코드에 찾아볼 수 없습니다 (플로우차트에는 있으나 미구현).

- **알림 및 모니터링** – **부분 구현**: SlackNotifier 및 NotificationService가 마련되어 있어, `notify()` 호출 시 DB에 Alert 저장 및 Slack 메시지 전송이 구현되었습니다 [52]. 이는 요구사항 중 "리스크 관리 및 알림 서비스"에 부합하는 부분입니다 [58]. Prometheus `REQUEST_COUNT` Counter 정의와 `/metrics` 엔드포인트 공개도 구현되어 모니터링 토대를 마련했습니다 [59] [60]. 반면, Alertmanager 연계나 Grafana 대시보드 등의 구축은 코드 외부의 영역이고, 시스템 상태 이상 시 Slack 알림(헬스체크 주기화)은 아직 수동 작업입니다.

- **스케줄러 기반 작업** – **부분 구현**: APScheduler를 사용할 수 있도록 코드가 구조화되어 있고, 간단한 백그라운드 스레드 실행기도 포함되었습니다 [18]. 그러나 정기 작업으로 등록된 구체적인 job은 없습니다. (예: "일일 전략 선택/튜닝, 주간 리포트 생성, 로그 회전" 등이 계획되었으나 실제 `scheduler.add_job()`으로 그런 함수들이 추가되지 않음) [16]. Scheduler 자체는 있지만 활용이 안 된 상태입니다.

- **전략 적응(국면 탐지/파라미터 조정)** – **클래스 구현, 통합 미흡**: RegimeDetector, ParamAdjuster, QualityAssessment, FeedbackMechanism 클래스가 모두 정의되고 테스트 커버리지가 있습니다 [61] [62]. 이는 요구된 기능을 염두에 둔 것으로 평가됩니다. 그러나 TradingBot 루프나 Scheduler 작업 내에 이들을 호출/활용하는 코드가 없어 **실시간 동작에 녹아들지 못한** 상태입니다. (예: 시장 국면에 따른 파라미터 조정이 실제 일어나지 않음).

- **시뮬레이션/백테스트 모드** – **미구현 (구조만)**: simulation, backtest 관련하여 별도 모드 전환이나 실행 스크립트가 없습니다. OrderExecutor의 `is_simulation` 플래그 정도만 실매매/모의매매 분기를 제공하며, 데이터 입력 소스 전환이나 결과 기록은 구현 전입니다. 시뮬레이터, 백테스터 모듈은 아직 아이디어 단계로 남아있습니다.

- **데이터베이스 스키마** – **부분 구현**: 최소한의 테이블 (market_data, system_config, strategy_param, alert)은 정의되었으나 [50] [51], 실제 활용은 system_config(설정 로드)와 alert(알림 저장) 정도입니다. 주문 및 포지션 관리를 위한 테이블이 빠져 있어, 주문 체결 내역이나 보유 자산 상태를 DB로 관리하지 못합니다. 이 부분은 백테스트/실매매 수익률 계산 등에 필수적인 요소인데 누락되어 있습니다.

- **API/UI 대시보드** – **미구현 (기본틀만)**: FastAPI로 `/metrics` 하나를 제공하지만, 그 외 REST API (전략 시작/정지, 데이터 조회 등)와 WebSocket 실시간 전송은 없습니다. React 대시보드도 별도 구현 사항으로 남아 있습니다. 인증/권한 관리 또한 미도입입니다.

위 내용을 요약하면, **Sigma-X의 현재 코드 구현은 설계 요구사항의 약 30~40% 수준의 골격과 일부 기능을 갖춘 상태**라고 볼 수 있습니다. 핵심 트레이딩 로직과 데이터파이프라인은 대부분 앞으로 구현해야 하는 과제로 남아있고, **일부 보조 기능(알림, 설정 관리 등)은 비교적 잘 반영**되었습니다. **전체 아키텍처의 방향성은 요구사항과 합치**하지만 **세부 구현이 따라가지 못해 아직 실사용이 어려운 미완성 프로토타입 단계**입니다.

개선 사항 및 제안

앞서 분석을 통해 파악된 **현재 구조의 부족한 부분과 개선 필요점**을 정리하면 다음과 같습니다. (각 항목은 해당 영역의 문제점과 개선방향을 함께 제시합니다.)

- **실시간 데이터 수집 및 전파 구현**: WebSocket 기반 시세 수집기를 구현하여 실제 시장 데이터가 TradingBot에 입력되도록 해야 합니다. 예를 들어 바이낸스 등의 실시간 데이터 API를 비동기로 연결하고, 수신한 틱 데이터를 **Redis Pub/Sub 채널**에 퍼블리시하여 여러 소비자가 구독할 수 있게 개선해야 합니다 [8]. 현재 DataCollector가 고정값을 반환하는 더미이므로 [26], 이를 교체해 **실시간 데이터 스트림**을 처리하도록 합니다. 또한 event_loop를 asyncio 등의 방법으로 구현하여, **무한 루프 형태로 데이터 수신 대기 -> 처리**를 지속 수행하게 함으로써 24시간 동작을 보장해야 합니다.

- **주문 처리 비동화 및 RabbitMQ 도입**: 현재 주문 실행이 TradingBot 내부(OrderExecutor)에서 동기적으로 이뤄지지만, 이를 **RabbitMQ** 큐잉으로 전환하여 **OrderWorker 프로세스/스레드**가 따로 주문을 처리하도록 변경이 필요합니다 [63] . RabbitMQ를 도입하면 주문 신호를 큐에 넣은 뒤 TradingBot은 곧바로 다음 작업을 수행할 수 있고, OrderWorker는 큐에서 신호를 꺼내 **실제 거래소 API 호출** 및 **주문 체결 결과를 DB에 기록**하는 흐름을 맡게 됩니다. 이때 주문 성공/실패 여부나 세부 체결 정보(가격, 수량 등)를 `orders` 테이블(신규 설계 필요)에 저장하고, 필요시 Slack 알림을 발생시켜 **신뢰성**과 **추적성**을 높입니다. RabbitMQ는 **영구 모드(persistent)**로 설정하여 만약 OrderWorker 다운시에도 신호가 유실되지 않게 하고, OrderWorker도 **자동 재시도/예외 처리** 로직을 갖춰 일시적 오류(통신 오류 등) 시 재처리를 시도하도록 해야 합니다.

- **병렬 처리 구조 보완**: 실시간 데이터 수집, 신호 생성, 주문 실행, 알림 전송 등의 작업들이 최대한 병렬로 일어날 수 있도록 **아키텍처 조정을 제안**합니다. Python GIL로 인해 멀티스레드의 한계가 있으므로, **프로세스 분리 또는 asyncio 활용**이 고려될 수 있습니다. 예를 들어, 시세 수집기/Redis 퍼블리셔는 별도 프로세스로 분리하고, 각 TradingBot(전략 실행 엔진)을 프로세스로 띄워서 서로 다른 전략/종목을 병렬 처리하며, 주문 실행 워커는 또 별도 프로세스로 운영하는 **마이크로서비스 구조**도 장기적으로 검토할 수 있습니다. 현재 구조를 크게 바꾸지 않으면서 개선하려면, **asyncio 이벤트 루프**를 도입하여 WebSocket 수신 코루틴, Redis 소비 코루틴, TradingBot 실행 코루틴 등을 단일 스레드에서 비동기로 처리하는 방법도 있습니다. 이 경우 각 작업 간 await로 양보하여 병렬성에 준하는 효과를 얻고, Python 단일 프로세스로도 일정 수준 동시성을 달성할 수 있습니다. 병렬성 향상은 곧 **확장성**과 직결되므로, 이 부분에 대한 개선이 이루어져야 실제 거래 환경에서 시스템이 버틸 수 있을 것입니다.

- **전략 플러그인 로딩 기능 구현**: `plugin_loader.load_plugins()`를 실제 동작하도록 개선해야 합니다. 우선 **플러그인 전략의 배포 방식**을 정해야 하는데, 예를 들어 `plugins/` 디렉터리에 사용자 정의 전략 모듈(.py 파일)을 넣으면 시스템이 부트시에 자동 인식하도록 할 수 있습니다. Python의 importlib나 pkgutil을 활용해 해당 디렉터리를 검색, BaseStrategy를 상속한 클래스들을 찾아 인스턴스화하거나 Registry에 등록하는 방식을 구현합니다. 그리고 `SystemConfig` 또는 별도 테이블에 현재 활성화할 전략 목록이나 기본 매개변수를 지정해두면, **코드 수정 없이도 DB 설정을 변경하여 전략 교체/추가**가 가능해집니다. 이러한 동적 로딩을 구현함으로써 **유연한 전략 교체**와 **플러그인 생태계**를 지원하게 될 것입니다. 아울러, 플러그인 로드 시 예외 처리 (문법 에러가 있는 플러그인 무시 등)와 보안 검토도 필요합니다.

- **전략 파라미터의 동적 업데이트 반영**: 현재 ParamAdjuster가 DB에 파라미터를 써넣는 것은 구현되었으나, **전략이 이를 실시간 반영**하도록 만드는 것이 중요합니다. 개선 방안으로는 전략 객체가 매 신호 생성 시 자신의 파라미터를 DB나 캐시에서 불러오거나, NotificationService 등을 통해 파라미터 변경 이벤트를 수신하면 내부 값을 갱신하도록 할 수 있습니다. 또는 보다 단순히, 매 트레이딩 루프마다 `StrategyParam` 테이블을 조회해 해당 전략의 중요한 파라미터를 세팅해주는 것도 가능합니다. 효율을 위해 Redis 캐시에 현재 전략 파라미터를 올려두고 (DB 변경 시 invalidate), TradingBot이 Redis로부터 읽어오는 구조도 생각할 수 있습니다. 궁극적으로 **사용자 개입 최소화**를 원한다면, 스케줄러에 **자동 파라미터 튜닝 작업**을 추가해야 합니다. 예컨대 매일 새벽 과거 데이터를 백테스트해 최적의 파라미터를 찾은 뒤 `strategy_param`을 업데이트하도록 하고, TradingBot은 다음 날 그 파라미터로 운용되는 식입니다. 이 때 변경된 파라미터를 Slack 등으로 알림 보내주면 투명성도 확보됩니다.

- **시뮬레이터/백테스트 모드 구현**: 시뮬레이션과 백테스트는 현재 구조상 가능은 하지만 이를 실행할 인터페이스가 없습니다. **별도의 실행 스크립트/명령**을 제공하여 사용자가 손쉽게 모드 전환을 할 수 있게 해야 합니다. 가령 `run_bot.py --mode live/sim` 또는 `python backtest.py --strategy=X --from=20210101 --to=20211231` 형태로 CLI를 설계해, 해당 모드에 맞는 초기화와 루프가 돌게 합니다. **시뮬레이션 모드**에서는 Redis 실시간 데이터 대신 과거 데이터 리플레이어나, 또는 실시간과 동일한 로직으로 PriceFeed를 Redis에 publish하고 TradingBot은 subscribe하여 진행해도 됩니다. 중요한 것은 **SimulatorExecutor**를 구현해 **가상의 체결 알고리즘**을 넣는 것입니다. 예를 들어, 시장가 매매라 가정하고 신호 발생 시 해당 시점 가격으로 즉시 체결되었다고 기록하거나, 지정가 매매일 경우 다음 틱에서 체결 여부를 판단하는 간단한 모델을 넣을 수 있습니다. 체결 결과는 메모리상의 가상 포지션에 반영하고, 최종적으로 전체 거래내역, PnL 등을 산출하도록 합니다. **백테스트 모드**에서는 속도가 중요하므로, Redis 등을 거치지 말고 과거

데이터 배열을 루프 돌면서 곧바로 TradingBot에 전달하는 식으로 구현할 수 있습니다. 백테스트 결과는 `bt_summary` 등의 테이블에 저장하거나, 일회성으로 Pandas DataFrame 등을 만들어 사용자에게 리포트 생성할 수 있습니다. 이러한 시뮬레이션/백테스트 모드를 제대로 구현하면, **운용 전에 전략을 충분히 시험**해볼 수 있고, 운용 후에도 **실제 성과와 백테스트를 비교**하여 전략 유효성을 검증하는 등 고도화에 큰 도움이 될 것입니다 [45].

- **DB 스키마 확장 및 일관성 보완**: 주문과 포지션을 저장할 테이블을 설계/구현하는 것이 시급합니다 [48]. 최소한 `orders` 테이블을 만들어 (주문ID, 시간, 종목, 매수/매도, 수량, 가격, 상태 등) 실매매와 모의매매의 주문 내역을 모두 기록해야 합니다. 그리고 `positions` 또는 `portfolio` 테이블을 두어 현재 보유 포지션과 잔고를 관리하면 리스크 관리나 대시보드 구현에 도움이 됩니다. 현재 MarketData 테이블이 있으나 사용이 안 되고 있으므로, 이를 **Tick/캔들 데이터 저장**에 활용하는 방안을 마련합니다. TimescaleDB 확장을 사용할 예정이라면, Tick 데이터를 해당 하이브리드 시간열 DB에 밀어넣어 효율적으로 보관/조회할 수 있도록 해야 합니다. 이렇게 DB 스키마가 확충되면, **모든 중요한 이벤트/상태가 DB에 일관되게 저장**되어 사후 분석, 리포팅, 장애 복구 등에 대비할 수 있습니다. 아울러, 테스트 중 발견된 이슈로 DB 스키마 변경이 필요할 경우 마이그레이션 도구(예: Alembic)를 도입하여 운영 DB와 코드 간 스키마 동기화를 관리하는 것도 권장됩니다.

- **API 및 대시보드 기능 추가**: FastAPI 기반으로 현재 `/metrics`만 노출하고 있는데, 여기에 **운용 제어 및 데이터 제공 API**를 추가로 구현해야 합니다 [24]. 예를 들어, **봇 제어**를 위해 `/bot/start`, `/bot/stop` 엔드포인트를 두고 실시간으로 TradingBot의 동작을 제어할 수 있어야 합니다. (현재는 run_bot.py를 실행/종료해야 하는데, 이는 운영상의 불편함이 있습니다.) 또 **전략 변경/업데이트** API (`/strategy/{name}/activate` 등)를 제공하면 UI에서 바로 특정 전략을 활성/비활성화할 수 있을 것입니다. **데이터 조회 API**도 중요한데, `/performance`나 `/positions` 등을 만들어 DB에 저장된 과거 수익률이나 현재 포지션, `alert` 테이블의 최근 알림 등을 조회 가능하게 해야 합니다. 이러한 API들은 추후 **프론트엔드 대시보드(React)**와 연동되어 사용자에게 정보를 제공하게 됩니다. 또한 **WebSocket**을 FastAPI에 통합하여 `/ws` 같은 경로로 실시간 시세나 이벤트를 브라우저에 push하면 사용자 대시보드에서 실시간 업데이트를 볼 수 있습니다. 인증/권한 관리를 위해 OAuth2 또는 JWT를 적용해, 대시보드 접속 시 로그인 확인 및 민감 정보 보호도 해야 합니다. 현재는 단일 사용자 시스템으로 보이지만, 향후 다중 사용자나 클라우드 서비스화도 염두에 둔다면 이 부분도 미리 구조를 잡아둘 필요가 있습니다.

- **로그 관리 및 예외 처리 강화**: 장기간 운영되는 자동매매 시스템에서는 **로그 파일 관리**와 **예외 처리**가 중요합니다. `sigma.utils.logger`가 기본 logging만 설정하고 파일 핸들러는 없으므로, **RotatingFileHandler** 등을 이용해 로그 파일이 용량 초과 시 순환되도록 해야 합니다 [64]. 또한 debug, info, error 레벨을 세분화하고, 주요 이벤트(전략 변경, 주문 체결, 오류 발생 등)에 대해서는 **DB Alert 저장 및 알림**을 남기는 식으로 로그와 알림을 연계하면 관리 효율이 높아집니다. 예외 처리 측면에서는, 외부 연동부 (API 통신, DB commit 등)에 try-except를 철저히 걸고 SlackNotifier 등을 통해 개발자에게 알려주는 패턴을 전역적으로 적용하는 것이 좋습니다 [65]. 현재 코드에서도 DB 설정 로드 실패 시 logger.warning을 남기는 처리 등이 있지만 [66], 이를 더욱 체계화하여 **전략 실행 오류 -> 해당 전략 일시 중지 및 알림**, **주문 API 실패 -> 재시도 & 알림**, **웹소켓 끊김 -> 재연결 시도 & 알림** 등의 시나리오를 구현해야 합니다. 이를 통해 시스템이 어떤 예외 상황에서도 완전히 멈추지 않고 최대한 **자동으로 복구(recovery)**하거나 최소한 **관리자에게 즉시 통보**하여 개입할 수 있게 만드는 것이 바람직합니다.

- **리스크 관리, 이상치/뉴스 처리**: 현재 구현에는 없지만 요구사항에 언급된 [10] **RiskManager, AnomalyDetector, NewsHandler** 등의 모듈도 추후 고려해야 합니다. 예를 들어, **RiskManager**는 포트폴리오 전체의 익스포저를 모니터링하여 과도한 손실 발생시 모든 포지션 청산 또는 신규진입 중단 등을 결정할 수 있습니다. **AnomalyDetector**는 입력 데이터의 이상치를 감지해 해당 틱을 무시하거나 별도 알림을 보낼 수 있고, **NewsHandler**는 큰 뉴스 이벤트(예: 경제지표 발표) 시 일정 시간 거래를 회피하거나 특별 전략으로 대응하게 할 수 있습니다. 이러한 기능들은 필수는 아니지만 시스템의 **신뢰성과 안정성**을 높여주는 요소이므로, 코어 기능 구현 후 여력이 생기면 점진적으로 추가하는 것을 추천합니다. 구조적으로는 TradingBot 실행 루프 내에 훅을 걸어 RiskManager/AnomalyDetector를 체크하고, 그 결과에 따라 전략 신호를 무시하거나, NotificationService로 경고를 보내는 통합이 가능할 것입니다.

요약하면, **Sigma-X 시스템은 현재 토대를 마련한 단계이므로, 상기 제안된 개선사항들을 중심으로 기능을 보완하면 자동매매 시스템으로서의 완성도를 크게 높일 수 있습니다.** 특히 실시간 데이터 처리와 주문 실행의 비동기화, 전략 관리의 유연성, 시뮬레이터와 백테스트의 구현은 핵심 우선순위로 보입니다. 이러한 개선을 통해 Sigma-X의 아키텍처는 요구된 성능, 실시간성, 확장성, 유연성, 관리성, 자동화 측면에서 더욱 **적합한 구조로 발전**하게 될 것입니다.

---

1 2 8 9 10 14 15 20 21 22 23 27 29 32 33 34 42 43 44 45 53 58 AGENTS.md
https://github.com/tjqjaqhd/sigma-x/blob/cb701cd56a2274e1c592fff1a42d6d746536c0fc/AGENTS.md

3 __init__.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/system/__init__.py

4 5 health_check.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/system/health_check.py

6 plugin_loader.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/system/plugin_loader.py

7 event_loop.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/system/event_loop.py

11 bot.py
https://github.com/tjqjaqhd/sigma-x/blob/cb701cd56a2274e1c592fff1a42d6d746536c0fc/sigma/core/bot.py

12 26 collector.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/data/collector.py

13 execution.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/core/execution.py

16 24 30 31 41 46 48 55 57 63 64 65 개선방향제안서.md
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/%EA%B0%9C%EC%84%A0%EB%B0%A9%ED%96%A5%EC%A0%9C%EC%95%88%EC%84%9C.md

17 18 scheduler.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/core/scheduler.py

19 run_bot.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/run_bot.py

25 60 api_service.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/system/api_service.py

28 59 metrics.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/system/metrics.py

35 36 strategies.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/core/strategies.py

37 38 39 40 adaptation.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/core/adaptation.py

47 50 51 models.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/data/models.py

49 cache.py
https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/system/cache.py

52 notification_service.py

https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/system/notification_service.py

54 56 66 config_loader.py

https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/sigma/config_loader.py

61 62 test_adaptation.py

https://github.com/tjqjaqhd/sigma-x/blob/cf9d5957825854089f390425ad2475f4f565d9ca/tests/test_adaptation.py