0. Design Entry / Reference

본 시스템의 UX/UI 설계 시작점은 아래 Figma Prototype을 기반으로 한다.

Figma: Agent Management Dashboard – Figma Make...

1. 목적과 배경

EAR 시스템의 Super Agent와 Multi-Agent 연동 기능을 기반으로, 개별 에이전트들의 전체 수명주기(Life Cycle)를 관리하는 전용 시스템을 설계한다. 이 시스템은 **에이전트 등록**, **에이전트 정보 관리**, **실시간 모니터링 및 지표 수집**, **에이전트 업무량 관리** 등 에이전트 운영에 필요한 기능을 제공한다. 또한 사용자가 Figma로 제시한 화면 목업을 바탕으로 사용자 경험(UX)을 체계적으로 설계하고, 개발 표준과 아키텍처를 정의한다.

2. 요구사항 및 핵심 기능

2.1 에이전트 등록 및 정보 관리

1. **에이전트 등록** – 신규 에이전트를 시스템에 등록하는 기능이다. 에이전트 ID, 이름, 설명, 에이전트 유형(예: RAG Agent, SAP Agent 등), 실행 환경(예: Python 3.9, Docker image), 최대 동시 수행량(concurrency), 태그 등을 입력한다.
2. **에이전트 수정/삭제** – 등록된 에이전트의 정보를 수정하거나 더 이상 사용하지 않는 에이전트를 비활성화/삭제할 수 있다. 삭제 시 관련 로그와 지표는 아카이빙 된다.
3. **역할 설정** – 각 에이전트가 수행할 업무(Agent Role)를 지정할 수 있다. Super Agent와 LangGraph로 오케스트레이션할 수 있도록 Agent Role을 정의한다.

2.2 에이전트 모니터링 및 지표 수집

1. **상태 모니터링** – 에이전트의 현재 상태(활성/비활성, 실행 중, 오류, 대기 중)를 실시간으로 확인한다. 마지막 하트비트 시간이 표시되며, 상태 갱신 주기는 5–10초 단위로 구성한다.
2. **성능 지표 측정** – 에이전트별로 CPU/메모리 사용률, 처리한 요청 수, 평균 응답 지연(latency), 오류율, 큐 대기 시간 등의 지표를 수집한다. 지표는 Prometheus Exporter 또는 자체 모니터링 모듈에서 주기적으로 수집하고, 시간 시계열 DB(InfluxDB 등)에 저장한다.
3. **로그 조회** – 에이전트가 수행한 작업의 로그를 조회할 수 있다. 로그에는 요청 ID, 타임스탬프, 요청 내용, 응답 결과, 오류 메시지 등이 포함된다.
4. **알림/경보** – 지표가 임계값을 초과하거나 에이전트 상태가 오류 상태가 될 경우 관리자에게 이메일·슬랙·웹훅 등으로 알림을 전송한다.

2.3 에이전트 업무량 관리

1. **작업 큐 관리** – 각 에이전트의 현재 대기 중인 작업 큐를 조회하고, 필요한 경우 작업 우선순위를 조정하거나 특정 에이전트로 작업을 이동시킬 수 있다.
2. **스케줄러 설정** – 에이전트별로 작업 스케줄(예: 매일 00:00에 수행, 10건 이상의 요청이 쌓이면 실행)을 설정할 수 있다. Cron 표현식을 지원하며, 스케줄 변경 시 즉시 반영된다.

3. **부하 분산** – 여러 에이전트에 동일한 역할이 할당된 경우, 시스템은 현재 부하와 성능 지표를 고려해 요청을 균등하게 분배한다. 필요 시 특정 에이전트의 최대 동시 처리량을 제한하여 시스템 전체 성능을 보장한다.

2.4 권한과 보안

1. **역할 기반 접근 제어(RBAC)** – 사용자는 관리자(Admin), 운영자(Operator), 개발자(Developer), 모니터링 전용(Viewer) 등 역할을 갖는다. 관리자만 에이전트 등록/삭제가 가능하며, 운영자는 상태 모니터링과 로그 조회만 가능하다.
2. **감사 로그** – 에이전트 등록, 수정, 삭제와 같은 중요한 이벤트는 감사 로그로 기록하고, 필요 시 이력을 조회할 수 있다.
3. **API 인증** – 에이전트 관리 API는 JWT 또는 OAuth2 토큰 기반 인증을 사용한다. Agent Instance와 통신하는 내부 API는 별도의 서비스 계정을 사용한다.

3. 화면 설계

Figma 목업을 참고하여 주요 화면을 설계하였다. 실제 구현 시에는 컴포넌트 라이브러리(예:
Ant Design, Material UI)를 활용하여 일관된 UI를 제공한다.

3.1 대시보드 화면

- **상단 요약 패널** – 전체 에이전트 수, 활성/비활성 에이전트 수, 오늘 처리된 작업 수, 평균 응답 속도 등의 핵심 지표를 카드 형태로 표시한다.
- **에이전트 상태 차트** – 원형 또는 막대 차트로 각 상태(실행 중, 대기 중, 오류, 비활성) 에이전트의 비율을 보여준다.
- **실시간 성능 그래프** – 시간 축에 따른 CPU/메모리 사용률과 처리량(TPS)을 실시간으로 시각화한다. 필터를 통해 특정 에이전트를 선택하거나 전체를 볼 수 있다.
- **최근 알림** – 임계값 초과, 실패한 작업 등 최근 10개의 알림을 리스트로 보여준다.

3.2 에이전트 목록 화면

- **검색 및 필터링** – 에이전트 이름, 유형, 상태, 역할 등으로 검색/필터링하는 기능을 제공한다.
- **테이블 구성** – 컬럼에는 에이전트 ID, 이름, 유형, 상태, 현재 큐 길이, 평균 응답시간, 마지막 하트비트, 등록일, 업데이트일, 관리 메뉴(상세, 수정, 삭제)가 포함된다.
- **다중 선택 및 일괄 작업** – 여러 에이전트를 선택하여 상태 변경(활성→비활성)이나 삭제 같은 일괄 작업을 수행할 수 있다.
- **페이징** – 대량의 에이전트를 효과적으로 관리하기 위해 페이지네이션을 지원한다.

3.3 에이전트 상세 화면

- **기본 정보 패널** – 에이전트의 고유 ID, 이름, 설명, 유형, 생성일, 수정일, 현재 상태, 최대 동시 처리량 등을 표시한다.
- **실시간 지표** – CPU/메모리 사용률, 평균 응답시간, 오류율, 처리된 요청 수 등을 그래프와 표로 보여준다. 기간(1 시간, 1 일, 1 주)을 선택할 수 있다.
- **작업 로그** – 해당 에이전트가 최근 처리한 작업 내역을 리스트로 표시하며, 각 작업의 시작/종료 시각, 요청 유형, 결과 상태를 확인할 수 있다.

- **설정 변경** – 에이전트의 설명, 최대 동시 처리량, 역할을 수정할 수 있는 버튼을 제공한다. 또한 에이전트를 재시작하거나 비활성화할 수 있는 제어 버튼을 배치한다.

3.4 에이전트 등록/수정 화면

- **입력 폼** – 에이전트 이름, 설명, 에이전트 유형을 선택하는 드롭다운, 실행 환경을 선택하는 영역(파이썬 버전, Docker 이미지 태그 등), 최대 동시 처리량을 입력하는 필드, 태그 입력란을 제공한다.
- **롤 지정** – 체크박스/드롭다운으로 에이전트가 수행할 역할을 선택하고, 필요 시 다중 선택을 지원한다.
- **유효성 검증** – 필수 입력값 누락, 잘못된 값 입력 시 폼 하단에 에러 메시지를 표시하며 저장을 막는다.
- **저장/취소 버튼** – 저장을 누르면 서버에 API 요청을 보내고, 성공 시 목록 화면으로 이동한다. 취소는 변경 사항을 무시하고 이전 화면으로 돌아간다.

3.5 모니터링 및 업무량 관리 화면

- **에이전트별 작업 큐** – 각 에이전트의 현재 대기 중인 작업 수와 평균 대기 시간을 테이블로 표시한다.
- **부하 분산 시뮬레이터** – 특정 에이전트 또는 역할을 선택하면 예상 처리 속도와 응답 지연을 시뮬레이션하여 보여주는 기능을 제공한다. 이를 통해 운영자는 처리량 조정을 검토할 수 있다.
- **스케줄 관리 탭** – 에이전트별 스케줄을 조회하고, Cron 표현식을 수정하거나 즉시 실행을 요청하는 UI를 제공한다.

4. 시스템 아키텍처

4.1 구성 요소

1. **Agent Management UI (Frontend)** – React (TypeScript) 기반 SPA로 구성된다. 사용자 인증/인가를 처리하고, 백엔드 API를 호출하여 에이전트 데이터와 지표를 표시한다. 차트 라이브러리(Echarts/Chart.js)와 UI 프레임워크(Ant Design 등)를 사용한다.
2. **Agent Controller Service (Backend)** – FastAPI 또는 Express.js로 작성된 REST API 서버로, 에이전트 CRUD, 모니터링 지표 조회, 작업 큐 관리, 알림 전송 등을 처리한다. 에이전트 실행 환경과 분리되어 있으며, 각 Agent Instance와 통신할 별도의 gRPC/REST 채널을 제공한다.
3. **Agent Instance / Runner** – 실제 업무를 수행하는 개별 Agent 프로세스들이다. 각 인스턴스는 작업 처리 로직을 포함하며, Agent Controller와 하트비트/상태 업데이트를 주고받는다. LangGraph를 사용하는 Super Agent의 경우 Multi-Agent 오케스트레이션 스크립트를 이 인스턴스가 구동한다.
4. **Metrics Aggregator** – Prometheus Exporter 또는 StatsD를 통해 Agent Instance가 전달한 메트릭을 수집하고, InfluxDB 또는 Prometheus 서버에 저장한다. Grafana와 연동해 대시보드 시각화를 지원한다.
5. **Task Queue & Scheduler** – Redis나 RabbitMQ를 기반으로 작업 큐를 관리하고, 스케줄러(예: Celery Beat, BullMQ)가 Cron 규칙에 따라 작업을 생성하여 에이전트에게 분배한다.
6. **Database** – PostgreSQL을 메인 데이터 저장소로 사용하여 에이전트 메타데이터, 역할, 작업 내역, 감사 로그를 저장한다. 메트릭 데이터는 InfluxDB/Prometheus에 별도로 저장한다.
7. **Notification Service** – 메트릭 임계값 초과나 에이전트 오류 발생 시 이메일, 슬랙, SMS 등으로 알림을 발송하는 모듈이다.

4.2 데이터 모델 (예시)

테이블/컬렉션	주요 필드
agents	id (PK), name, description, type, status (enum), env_config (JSON), max_concurrency, tags (array), created_at, updated_at
agent_roles	agent_id (FK), role_name
agent_metrics	id (PK), agent_id (FK), timestamp, cpu_usage, memory_usage, requests_processed, av
agent_tasks	id (PK), agent_id (FK), job_id, status (enum), received_at, started_at, finished_at, result
job_queue	job_id (PK), payload (JSON), priority, status (enum), assigned_agent_id, created_at, sc
audit_logs	id (PK), user_id, event_type, target_id, timestamp, details (JSON)

5. 기술 스택과 개발 표준

5.1 프론트엔드 표준

- **언어/프레임워크** – React 18 + TypeScript. 상태 관리는 React Query 또는 Redux Toolkit을 활용한다.
- **컴포넌트 설계** – 페이지(Page) → 섹션(Section) → 원자(Atomic) 컴포넌트 계층을 준수한다. 재사용 가능한 UI 컴포넌트는 `components/common` 디렉터리에 두고, 페이지 전용 컴포넌트는 `pages/agent` 하위에 위치한다.
- **스타일 가이드** – 프로젝트 전체에서 Ant Design 테마를 사용하고, 커스텀 스타일은 CSS-in-JS(예: styled-components)나 CSS Modules를 이용한다.

- **코딩 컨벤션** – AirBnB JavaScript/TypeScript 스타일 가이드를 기반으로 ESLint와 Prettier를 설정하여 일관된 코드 품질을 유지한다.
- **API 통신** – axios를 이용해 Backend API와 통신하며, 각 API 호출 로직은 `hooks/useAgentApi.ts` 같은 커스텀 훅으로 분리한다. 공통 오류 처리 로직을 구현한다.
- **테스트** – Jest와 React Testing Library를 이용해 컴포넌트 단위 테스트를 작성하고, Cypress로 E2E 테스트를 수행한다.

5.2 백엔드 표준

- **언어/프레임워크** – Python 3.9 + FastAPI (또는 Node.js + Express). 비동기 처리를 위해 `asyncio`를 적극 활용한다.
- **아키텍처** – MVC(MTV) 패턴을 따라 라우터, 서비스, 데이터 액세스 레이어를 분리한다. 각 API 엔드포인트는 명확한 책임을 가진 서비스 함수에 위임한다.
- **데이터베이스 ORM** – SQLModel/SQLAlchemy (Python) 또는 Prisma (Node.js)를 사용한다. 스키마 마이그레이션 도구로 Alembic 또는 Prisma Migrate를 적용한다.
- **API 문서화** – FastAPI의 자동 스웨거 문서를 활용하여 API 사양을 문서화한다. 엔드포인트, 파라미터, 응답 모델을 명확히 정의한다.
- **로깅** – 구조화된 로그(json)을 남기며, 로깅 레벨(INFO, ERROR)을 구분한다. 센트리(Sentry)와 연동하여 에러를 추적한다.
- **테스트** – PyTest/pytest-asyncio를 이용해 유닛 테스트와 통합 테스트를 작성한다. 데이터베이스는 test container를 사용해 격리된 환경을 제공한다.

5.3 공통 규칙

- **버전 관리** – Git Flow 또는 trunk based 개발 방식을 사용한다. 기능별 브랜치를 생성하고 Pull Request를 통해 코드 리뷰를 진행한다.
- **커밋 메시지 규칙** – Conventional Commits(예: `feat:`, `fix:`, `chore:`) 규칙을 따르며, 업무 ID(예: `AGENT-001`)를 커밋 메시지에 포함한다.
- **보안** – 모든 외부 입력값은 서버 측에서 검증한다. HTTPS를 기본으로 사용하며, 민감 정보는 환경변수와 Secret Manager에 저장한다. 토큰 기반 인증을 사용하여 CSRF/Replay Attack을 방지한다.
- **CI/CD** – GitHub Actions 또는 GitLab CI를 사용해 코드 정적 분석, 테스트 실행, 빌드, 컨테이너 이미지 배포를 자동화한다. Kubernetes 환경에 배포하는 경우 Helm Chart로 버전 관리한다.

6. API 설계 예시

다음은 에이전트 관리 API의 일부 예시이다. 실제 프로젝트에서는 엔드포인트 이름과 파라미터를 상세히 정의한다.

Method	Endpoint	설명
`GET`	`/agents`	에이전트 목록을 페이징 형식으로 조회한다. 필터 파라미터로 상태, 유형, 이름 등을 지원한다.
`POST`	`/agents`	새 에이전트를 등록한다. 요청 본문에는 name, description, type, env_config, max_concurrency, roles, tags가 포함된다.
`GET`	`/agents/{agent_id}`	지정한 에이전트의 상세 정보를 조회한다.

Method	Endpoint	설명
`PUT`	`/agents/{agent_id}`	에이전트 정보를 수정한다. 일부 필드(예: type)는 수정할 수 없다.
`DELETE`	`/agents/{agent_id}`	에이전트를 비활성화하거나 삭제한다.
`GET`	`/agents/{agent_id}/metrics`	특정 에이전트의 성능 지표를 조회한다. 기간(start_time, end_time)과 집계 주기(interval) 파라미터를 받는다.
`GET`	`/agents/{agent_id}/tasks`	특정 에이전트가 처리한 작업 목록을 조회한다. 상태 필터(성공/실패/진행 중)와 페이징을 지원한다.
`POST`	`/jobs`	새 작업을 생성하고, 스케줄러를 통해 적절한 에이전트에 할당한다.
`GET`	`/jobs/{job_id}`	작업의 상태와 결과를 조회한다.

7. 추후 고려 사항

1. **자동 스케일링** – 에이전트 수를 자동으로 확장/축소하는 기능을 도입하여 트래픽 급증 시 안정적으로 대응한다.
2. **권한 템플릿** – 반복적인 역할 설정을 간소화하기 위해 권한 템플릿을 미리 정의할 수 있다.
3. **비주얼 워크플로 편집기** – LangGraph 기반의 워크플로를 시각적으로 구성하고 배포할 수 있는 편집기를 통합하여 Super Agent 관리를 한층 개선한다.
4. **AI 기반 예측** – 에이전트의 성능 지표와 업무량을 기반으로 향후 부하를 예측하고, 최적의 스케줄링을 제안하는 AI 모델을 도입한다.

8. 결론

에이전트 라이프사이클 관리 시스템은 EAR 플랫폼의 핵심 구성요소로서, 에이전트 등록부터 모니터링, 업무량 관리까지 통합적으로 제공해야 한다. 본 설계 문서는 사용자 요구사항과 시스템 아키텍처, UI 화면 설계, 개발 표준 및 API 사양을 제시하였다. 앞으로의 구현 단계에서는 Figma 목업을 상세히 반영하여 UI를 완성하고, 반복적인 피드백을 통해 시스템의 안정성과 사용성을 높여 나가야 한다.

https://github.com/yihana/agenthub
<변경사항>-파일 17개

feat(AGENT-001): Add agent lifecycle management (DB, APIs, UI) #1

Motivation

- Provide a centralized Agent Life Cycle management capability for registering, monitoring, and managing workload for individual agents.
- Persist agent metadata, roles, metrics, tasks and job queue data for both Postgres and HANA backends.
- Expose REST APIs to support Agent CRUD, metrics, task lists and job queue operations so orchestration and automation can integrate with the system.
- Deliver a basic admin UI (dashboard, list, detail, form, monitoring) so operators can manage and observe agents from the frontend.

Description

- Added new database tables and indexes for agent lifecycle management (`agents`, `agent_roles`, `agent_metrics`, `agent_tasks`, `job_queue`, `audit_logs`) and updated schema files for Postgres and HANA and the programmatic DB initializers (`server/db-postgres.ts`, `server/db-hana.ts`, `server/schemas*.sql`).
- Implemented backend endpoints for agent operations and summaries in `server/routes/agents.ts` and a job queue API in `server/routes/jobs.ts`, and registered them in `server/index.ts` as `'/api/agents'` and `'/api/jobs'` respectively.
- Added frontend integration: a new API hook `web/src/hooks/useAgentManagementApi.ts`, UI pages `AgentDashboardPage`, `AgentListPage`, `AgentDetailPage`, `AgentFormPage`, `AgentMonitoringPage`, styles `AgentManagement.css`, icon mappings, and routes wired in `web/src/App.tsx`.
- Seeded menu entries for the new Agent Management area during DB init and added audit logging on key operations via `audit_logs` inserts.

Testing

- Started the frontend dev server with `npm run dev:web` and confirmed the Vite dev server served the app (local host URL reported), which succeeded.
- Captured an automated screenshot of the `/agent-dashboard` route using Playwright to verify UI rendering, and the artifact was produced successfully.
- No automated unit or integration test suites (e.g. Jest/PyTest) were executed as part of this change.
- Database schema changes were added to the migration/initializer scripts but database initialization was not executed in CI here (no automated DB migration run in this rollout).