

오프닝

안녕하세요. 오늘은 리서치 프로젝트로 진행한 **ETF Atlas** 를 소개하면서, 그 과정에서 사용한 기술들에 대해 이야기하려 합니다.

문제 제기

본격적인 기술 소개에 앞서, 왜 이런 기술이 필요한지부터 짚어보겠습니다.

한국 ETF가 약 900개인데, 그중 순자산 기준으로 걸러낸 우리 유니버스만 해도 수백 개이고, 보유종목이 수천 개입니다. 이 데이터에서 의미 있는 정보를 뽑으려면 **관계를 탐색** 해야 합니다.

예를 들어 "삼성전자를 가장 많이 보유한 ETF는?"이라는 질문은 관계형 DB에서 **JOIN** 3개에 **GROUP BY**, **ORDER BY** 를 써야 합니다. "KODEX 200과 비슷한 ETF는?"은 자기 조인에 서브쿼리, **HAVING** 까지 필요합니다. **쿼리가 복잡해질수록 성능과 가독성 모두 떨어집니다.**

거기에 한 가지 더 욕심을 내봤습니다. 이런 복잡한 질문을 개발자가 아닌 일반 사용자도 할 수 있으면 어떨까? "반도체 ETF 중 보수율 낮은 것 알려줘"라고 **자연어로 물어보면 알아서 데이터를 찾아 답해주는 챗봇** 을 만들고 싶었습니다.

이 두 가지 — **복잡한 관계 탐색과 자연어 질의** — 를 해결하기 위해 선택한 기술이 **Apache AGE**와 **smolagents**입니다. 여기에 데이터 수집 자동화를 위한 **Apache Airflow**까지, 이 세 가지 기술의 실제 적용 경험을 공유하겠습니다.

기술 선택 배경

이 프로젝트는 **회사에서 기술 리서치 차원** 으로 진행했습니다. 주제는 평소 관심이 있던 ETF 투자로 잡았는데, 실제 서비스에 가까운 도메인으로 기술을 검증해야 리서치 결과가 의미 있다고 생각했기 때문입니다.

리서치 대상 기술은 세 가지입니다.

- **그래프 데이터베이스** — ETF 데이터는 본질적으로 관계 중심. Neo4j 대신 PostgreSQL 확장인 **Apache AGE** 선택
- **LLM 에이전트 프레임워크** — LangChain은 추상화가 무거워서 경량 프레임워크인 **smolagents** 선택
- **데이터 파이프라인** — 배치 중심 워크로드에 Python 네이티브인 **Apache Airflow** 선택

이 세 기술이 실제 서비스 수준의 프로젝트에서 얼마나 쓸 만한지 직접 검증해 보는 것이 이번 리서치의 목표였습니다.

Apache AGE 기술 소개

Apache AGE란?

Apache AGE 는 Apache Software Foundation의 인큐베이팅 프로젝트로, PostgreSQL에 그래프 데이터베이스 기능을 추가하는 확장(Extension)입니다. 이름 자체가 "A Graph Extension" 의 약자입니다.

핵심 특징은 세 가지입니다.

- **PostgreSQL 위에서 동작** — 별도 DB를 띄울 필요 없이 확장만 설치하면 됩니다
- **Cypher 쿼리 지원** — Neo4j에서 쓰던 그 Cypher 문법 그대로 쓸 수 있습니다
- **SQL + Cypher 혼용** — 기존 관계형 테이블과 그래프 데이터를 하나의 DB에서 조회 가능

라이선스는 **Apache 2.0** 으로, 상용 환경에서도 자유롭게 사용 가능합니다.

Neo4j vs Apache AGE

Neo4j

- 전용 그래프 DB, 성능과 기능 면에서 가장 성숙
- Community Edition은 AGPL, Enterprise는 상용 라이선스
- 별도 서버 운영 필요
- RDBMS와 데이터 분리 관리

Apache AGE

- PostgreSQL 확장 — 기존 인프라 그대로 활용
- **Apache 2.0** 라이선스
- 하나의 DB에서 관계형 + 그래프
- Cypher 지원 불완전, 에코시스템 작음

이미 PostgreSQL을 쓰고 있고, 그래프 쿼리가 핵심 워크로드의 일부라면 AGE가 합리적인 선택입니다.

AGE 동작 방식

일반 SQL 안에 `cypher()` 함수를 호출하는 형태입니다.

```
SELECT * FROM cypher('etf_graph', $$  
MATCH (e:ETF)-[h:HOLDS]->(s:Stock)  
WHERE e.code = '069500'  
RETURN {stock: s.name, weight: h.weight}  
$$) as (result agtype);
```

주의할 점: 그래프 이름 지정 필수, 결과 타입은 `agtype`, RETURN 값이 여러 개일 때는 맵으로 감싸야 합니다 (단일 컬럼 반환).

smolagents 기술 소개

smolagents란?

smolagents 는 Hugging Face에서 만든 경량 AI 에이전트 프레임워크입니다. 2024년 말에 공개됐고, 핵심 철학은 "최소한의 추상화"입니다.

LangChain이 모든 것을 추상화하려는 접근이라면, smolagents는 필요한 것만 제공하고 나머지는 개발자가 직접 제어하는 방식입니다.

핵심 구성 요소는 세 가지입니다.

- **Model** — 에이전트의 두뇌 역할을 하는 LLM. OpenAI, Anthropic 등 다양한 모델을 플러그인 방식으로 연결
- **Tool** — 에이전트가 실행할 수 있는 도구. `name`, `description`, `inputs`, `output_type`, `forward()` 다섯 가지만 정의
- **Agent** — Model과 Tool을 조합하는 실행 주체. **CodeAgent**는 Python 코드를 직접 생성해서 도구 호출

Model + Tool + Agent, 이 세 가지가 전부입니다.

LangChain vs smolagents

LangChain

- 풍부한 에코시스템 (수백 개 통합)
- 추상화 레이어가 깊어 디버깅 어려움
- 간단한 기능도 여러 클래스 필요
- breaking change 잦음

smolagents

- 코어가 작고 코드를 직접 읽을 수 있을 만큼 단순
- Tool 하나, Agent 하나면 시작 가능
- 에코시스템 작음, 고급 기능은 직접 구현
- 도구 호출 중심 에이전트에 빠른 프로토타이핑

Apache Airflow 기술 소개

Apache Airflow란?

Apache Airflow는 데이터 파이프라인을 작성하고, 스케줄링하고, 모니터링하는 플랫폼입니다. Airbnb에서 2014년에 만들었고, 지금은 Apache 최상위 프로젝트입니다.

핵심 개념은 네 가지입니다.

- **DAG** — Directed Acyclic Graph. 작업 흐름을 Python 코드로 정의한 것
- **Task** — DAG 안에서 실행되는 개별 작업 단위
- **Operator** — Task가 "무엇을 할지" 정의하는 템플릿 (`PythonOperator`, `ShortCircuitOperator` 등)
- **Scheduler** — DAG 스케줄에 따라 작업을 자동 트리거하는 엔진

왜 Airflow를 선택했나 — NiFi와의 비교

NiFi

- NSA에서 만든 데이터 흐름 관리 시스템
- 실시간 데이터 라우팅에 초점
- GUI 드래그 앤 드롭으로 파이프라인 구성
- IoT, 로그 수집, 실시간 전달에 강함

Airflow

- Airbnb에서 만든 워크플로우 오케스트레이터
- 배치 작업 오케스트레이션에 초점
- Python 코드로 워크플로우 정의 → Git 버전 관리 가능
- 배치 ETL, 정기 수집, ML 파이프라인에 강함

"평일 장 마감 후에 KRX에서 데이터를 가져와서 가공한 뒤 DB에 적재한다." — 전형적인 배치 작업.
Python 네이티브인 Airflow가 자연스러운 선택.

프로젝트 개요 & 데이터 파이프라인

프로젝트 개요

ETF Atlas는 한국 ETF 정보를 수집하고, 분석하고, AI 챗봇으로 질의할 수 있는 서비스입니다.

- **ETF 탐색** — 테마별 분류, 검색, 유사 ETF 찾기
- **포트폴리오 관리** — ETF/주식 매수 기록, 목표 비중 설정, 수익률 추적
- **워치리스트** — 관심 ETF 등록, 보유종목 변동 알림
- **AI 챗봇** — 자연어로 ETF 데이터 질의

관심 있는 도메인에 리서치 대상 기술을 적용해서 **실전 수준으로 검증** 한 프로젝트입니다.

전체 아키텍처

시스템은 **4개의 Docker 컨테이너**로 구성됩니다.

- **DB** — PostgreSQL 17 + Apache AGE + pgvector
- **Backend** — FastAPI (Python) + smolagents
- **Frontend** — React + TypeScript + Vite
- **Airflow** — 데이터 수집 파이프라인

하나의 PostgreSQL에 관계형 테이블과 그래프 데이터가 공존합니다. CRUD는 관계형 테이블, 관계 탐색은 그래프 — 이게 Apache AGE의 가장 큰 장점입니다.

데이터 파이프라인 — Airflow 활용

ETF Atlas에서는 **6개의 Airflow DAG**를 운영하고 있습니다.

- **일별 ETF 동기화 (AGE)** — 거래일 다음날 새벽 4시(KST), KRX 데이터 수집 → ETF/Price 노드 생성 → HOLDS 관계 → 수익률 계산 → Discord 알림
- **실시간 가격 수집 (RDB)** — 평일 장중 10분마다, **ShortCircuitOperator**로 장 오픈 여부 판단 후 현재 가 수집 → 포트폴리오 스냅샷 갱신
- **주간 태깅 (AGE)** — 토요일 새벽 3시, 규칙 기반 + 키워드 매칭 + LLM 분류 3단계 전략
- **RDB 메타데이터 동기화** — 평일 오전 7시, KRX에서 ETF 코드/이름 동기화
- **백필 (AGE)** — 수동 실행, 과거 거래일 데이터 한꺼번에 수집 (보유종목 수집에 6시간 타임아웃)

- **코드 예제 임베딩 (RDB)** — 수동 실행, 챗봇용 코드 예제를 LLM으로 질문 일반화 후 OpenAI 임베딩 생성
→ pgvector 저장

Airflow에서 유용했던 기능들

- **Configuration as Code** — DAG가 Python 코드 → Git 버전 관리, 코드 리뷰 가능
- **XCom** — Task 간 데이터 전달. "거래일 목록 조회" Task가 날짜 리스트를 반환하면 다음 Task가 수신
- **웹 UI** — 실행 이력, 성공/실패 상태, 로그 확인. 실패한 Task만 골라서 재실행 가능
- **자동 재시도** — `retries=3, retry_delay=timedelta(minutes=5)` 만 설정하면 5분 간격 3번 자동 재시도
- **병렬 실행** — 의존 관계 없는 Task는 자동 병렬 실행

그래프 데이터 활용

그래프 데이터 모델링

노드 6종류, 관계 5종류:

- ETF, Stock, Price, Company, Tag, User
- HOLDS, HAS_PRICE, MANAGED_BY, TAGGED, WATCHES

"삼성전자를 보유한 ETF 중에서 반도체 태그가 달린 것" 같은 질의를 MATCH 패턴 하나로 표현할 수 있습니다.

Cypher 쿼리 실전 예시

유사 ETF 찾기 — 공통 보유종목의 비중 최솟값을 합산해 유사도 계산:

```

MATCH (e1:ETF {code: '069500'})-[h1:HOLDS]->(s:Stock)
MATCH (e2:ETF)-[h2:HOLDS]->(s) WHERE e1 <> e2
WITH e2, COUNT(s) as overlap,
    SUM(CASE WHEN h1.weight < h2.weight
              THEN h1.weight ELSE h2.weight END) as similarity
RETURN {code: e2.code, name: e2.name,
        overlap: overlap, similarity: similarity}
ORDER BY similarity DESC LIMIT 5

```

관계형 DB였다면 자기 조인 + GROUP BY + HAVING으로 복잡해질 쿼리가, 그래프에서는 MATCH 패턴으로 자연스럽게 표현됩니다.

보유종목 변동 감지 — 특정 ETF의 이번 주와 지난주 보유종목을 비교해서 신규 편입, 제외, 비중 변화를 추출:

```

-- 특정 날짜의 보유종목 조회
MATCH (e:ETF {code: '069500'})-[h:HOLDS]->(s:Stock)
WHERE h.date <= '2026-02-14'
WITH DISTINCT h.date as d
ORDER BY d DESC LIMIT 1

```

기준일과 비교일 각각의 HOLDS 관계를 조회한 뒤, Python에서 두 시점의 종목별 비중을 비교해서 added, removed, increased, decreased로 분류합니다.

ETF 탐색

그래프 데이터가 어떻게 활용되는지 보겠습니다.

- **태그 페이지** — 반도체, AI, 2차전지 같은 테마별로 ETF를 조회. `(ETF) - [:TAGGED] ->(Tag)` 관계 탐색
- **ETF 상세** — 보유종목 TOP 10 표시. `(ETF) - [:HOLDS] ->(Stock)` 관계에서 최신 날짜의 비중 조회
- **유사 ETF** — 두 ETF 간 보유종목의 비중 유사도를 계산. 단순히 겹치는 종목 수가 아니라, 공통 종목의 **비중 중 작은 값을 합산** 해서 유사도를 구함

비중까지 고려하니까 실제로 비슷한 구성의 ETF를 찾을 수 있습니다. 이런 관계 탐색 쿼리들이 그래프 DB의 진가를 보여주는 부분입니다.

보유종목 변동 감지

위치리스트의 핵심 기능입니다. 사용자가 관심 ETF를 등록하면, **일별/주별/월별**로 보유종목 비중 변화를 추적합니다.

그래프에서 두 시점의 **HOLDS** 관계를 비교해서 신규 편입, 제외, 비중 증가, 비중 감소를 분류합니다.

예를 들어 KODEX 반도체가 SK하이닉스 비중을 15%에서 18%로 올렸다면, **"비중 증가 +3%p"**로 표시됩니다.

3%p 이상 변동이 있으면 Discord로 알림을 보냅니다. 관리자 **User** 노드의 **WATCHES** 관계를 타고 알림 대상을 찾습니다.

AI 챗봇

전체 구조

`ChatService`의 구조는 단순합니다. `smolagents`의 `CodeAgent`가 `ReAct` 방식으로 질문을 처리하고, `pgvector` 기반 Few-shot 코드 예시가 정확도를 보완합니다.

모델은 LiteLLM을 통해 `GPT-4.1-mini` 사용, `10개의 커스텀 Tool` 등록.

- **검색:** ETF 검색, 종목 검색, 태그 목록
- **정보:** ETF 상세, ETF 가격, 주식 가격, 보유종목 변동
- **분석:** 유사 ETF, ETF 비교, 직접 Cypher 쿼리

모든 도구가 내부적으로 Apache AGE 그래프를 조회합니다. 두 기술(Apache AGE, smolagents)이 여기서 만납니다.

CodeAgent 동작 흐름

- **1단계:** 사용자가 "삼성전자를 가장 많이 보유한 ETF는?" 질문
- **2단계:** 에이전트가 Python 코드 생성
- **3단계:** 코드 실행 → 도구 호출
- **4단계:** 실행 결과로 최종 답변 생성

```
result = stock_search("삼성전자")
stock_code = json.loads(result)[0]["code"]
holdings = graph_query(
    "MATCH (e:ETF)-[h:HOLDS]->(s:Stock {code: '" +
    + stock_code + "'}) ..."
)
```

최대 **10스텝** 까지 반복. 중간에 에러가 나면 에이전트가 스스로 코드를 수정해서 재시도합니다. 이게 JSON 기반 도구 호출 대비 CodeAgent의 장점입니다. 변수에 값을 저장하고, 조건 분기도 가능하니까요.

실행 과정은 **스트리밍**으로 프론트엔드에 실시간 표시됩니다. 챗봇이 어떤 도구를 호출하고, 어떤 코드를 실행했는지 접을 수 있는 패널로 보여줍니다.

사용자 입장에서는 AI가 어떤 과정을 거쳐 답을 만들었는지 **투명하게** 볼 수 있어서 신뢰도가 높아집니다.

Tool 정의

smolagents에서 도구를 정의하는 방법은 단순합니다. Tool 클래스를 상속받아 **다섯 가지** 만 정의하면 됩니다.

```
class ETFSearchTool(Tool):
    name = "etf_search"
    description = "ETF를 이름이나 코드로 검색합니다"
    inputs = {
        "query": {"type": "string", "description": "검색어"},
        "limit": {"type": "integer", "description": "결과 수"}
    }
    output_type = "string"

    def forward(self, query: str, limit: int = 10) -> str:
        graph = GraphService(self.db)
        results = graph.search_etfs(query, limit)
        return json.dumps(results, ensure_ascii=False)
```

핵심은 **description** — 에이전트가 이 설명을 보고 어떤 도구를 호출할지 결정합니다.

Few-shot 학습

25개의 Python 코드 예시를 사전에 준비해서 [pgvector](#)에 임베딩으로 저장합니다.

사용자가 질문하면, 질문과 유사한 **3개의 코드 예시**를 벡터 검색으로 찾아서 프롬프트에 주입합니다.

"반도체 ETF 3개 비교해줘"라고 물으면 비슷한 비교 패턴의 Python 코드 예시가 컨텍스트로 들어가서 에이전트가 도구를 어떻게 조합해야 하는지 참고할 수 있습니다.

CodeAgent가 Python 코드를 생성하는 주체이므로, 예시도 Python 코드인 것이 자연스럽습니다. 같은 PostgreSQL 안에서 벡터 검색과 그래프 쿼리를 함께 활용합니다.

설계 변천사

지금의 구조가 처음부터 이랬던 건 아닙니다. 몇 번의 시행착오를 거쳐 도달한 결과입니다.

1단계: ReAct만 사용

처음에는 CodeAgent의 ReAct 패턴만 사용했습니다. 단순한 질문에는 잘 동작했지만, 에이전트가 생성하는 Cypher 쿼리의 정확도가 낮았습니다. AGE의 Cypher 문법이 Neo4j와 미묘하게 다른데, 에이전트가 이런 차 이를 알 리가 없으니까요.

2단계: Cypher Few-shot 추가

Cypher 쿼리 100개를 Few-shot 예시로 준비하고, `pgvector`로 유사한 예시를 찾아 프롬프트에 주입했습니다. "삼성전자를 많이 보유한 ETF"라고 물으면 비슷한 Cypher 예시 3개가 컨텍스트로 들어가서 에이전트가 AGE에 맞는 Cypher를 생성할 수 있게 됐습니다.

하지만 새로운 문제가 생겼습니다. "반도체 ETF 3개의 수익률, 보유종목, 가격을 모두 비교해줘" 같은 **복잡한 질문에서 에이전트가 방향을 잃었습니다**. 도구 호출이 7~8번 필요한데, 중간에 불필요한 호출을 반복하거나 멈추는 경우가 있었습니다.

3단계: 분류기 + Plan-Execute-Summarize

앞단에 분류기를 두었습니다. LLM이 질문을 분석해서 예상 도구 호출 횟수를 판단하고, 3회 이하면 기존 ReAct로, 4회 이상이면 **Plan-Execute-Summarize(PES)** 방식으로 전환합니다.

- **Plan** — LLM이 실행 계획을 JSON으로 생성
- **Execute** — 계획대로 도구를 순차 실행
- **Summarize** — 수집된 결과를 종합해서 최종 답변 생성

복잡한 질의에서 안정적인 결과를 얻을 수 있었습니다.

4단계: 깨달음 — CodeAgent에 PES가 필요한가?

smolagents의 CodeAgent는 Python 코드를 직접 생성합니다. `for` 루프를 돌 수 있고, 변수에 값을 저장할 수 있고, 조건 분기도 가능합니다. 그렇다면 외부 플래너가 왜 필요한가?

Plan-Execute-Summarize는 **JSON으로 도구를 호출하는 에이전트**에 필요한 패턴입니다. 도구 호출 사이에 로직을 넣을 수 없으니까 미리 계획을 세워야 하는 겁니다. 하지만 CodeAgent는 코드로 직접 복잡한 로직을 구현할 수 있습니다. "반도체 ETF 3개 비교"를 위해 계획을 세울 필요 없이, `for` 루프 하나면 됩니다.

최종 구조: ReAct 단일 경로

이 깨달음 이후 Plan-Execute-Summarize를 제거하고 **ReAct 단일 경로**로 돌아갔습니다. 대신 두 가지를 보강했습니다.

- **시스템 프롬프트 가이드** — `for` 루프, 결과 체이닝, 데이터 정렬 등 코드로 처리하는 패턴을 명시
- **Few-shot 전환** — Cypher 쿼리 예시를 **Python 코드 예시**로 전환. 에이전트가 코드를 생성하는 주체이니까 예시도 코드여야 자연스럽습니다

```
results = []
for code in etf_codes:
    info = get_etf_info(etf_code=code)
    results.append(json.loads(info))
sorted_results = sorted(results,
    key=lambda x: x.get("expense_ratio", 999))
```

도구의 본질을 이해하는 것이 중요합니다. CodeAgent는 "코드를 생성하는 에이전트"입니다. 이 특성을 살리면, 외부 보조 장치 없이도 충분히 복잡한 작업을 수행할 수 있습니다.

포트폴리오 관리 & 대시보드

포트폴리오 관리

사용자가 ETF와 주식을 매수 기록하면, 현재 가격 기반으로 포트폴리오 가치를 계산합니다. 목표 비중을 설정하면 실제 비중과의 괴리를 보여줍니다.

대시보드

대시보드에서는 [Recharts](#)로 포트폴리오 가치 변화를 시각화합니다. [portfolio_snapshots](#) 테이블에 일별 스냅샷이 저장되고, Airflow에서 가격 수집 후 자동으로 스냅샷을 찍습니다.

이 기능은 관계형 테이블을 사용합니다. 모든 것을 그래프에 넣지 않고, CRUD 중심의 데이터는 관계형으로, 관계 탐색은 **그래프로** 분리한 설계.

시행착오

스냅샷과 면등성 – "오늘"이 도대체 언제인가

"매일 장 마감 후에 포트폴리오 가치를 계산해서 저장한다." 단순해 보이지만, 기준일과 수집일의 정의가 미흡한 상태에서 출발하니 커밋 10개 넘게 수정이 필요했습니다.

구조 변천

- **메인 DAG에 끼워넣기** – `collect_prices` 뒤에 `snapshot_portfolios` 태스크 추가. XCom의 `trading_date`를 기준일로 사용
- **스냅샷 DAG 분리** – 별도 `portfolio_snapshot.py` (평일 16시). yfinance로 독립 수집했으나, 메인 DAG보다 먼저 실행되면 **어제 가격으로 오늘 스냅샷**이 찍히는 문제
- **실시간 DAG 등장** – 장중 10분마다 `ticker_prices` 캐싱. 가격 소스가 AGE/yfinance/ticker_prices 세 곳으로 분산되면서 기준일 정의가 모호해짐
- **스냅샷 DAG 제거, 실시간 DAG로 통합** – `collect_prices >> update_snapshots` 순서로 일관성 확보. `UNIQUE(portfolio_id, date)` + UPSERT로 같은 날짜는 덮어쓰기

면등성을 깨뜨린 것들

- **공휴일 수익률 0% 덮어쓰기** – 금요일이 공휴일이면 KRX 데이터 없음 → `sync_returns` 가 기존 수익률을 0%로 갱신. "데이터 없음" ≠ "값이 0" → 새 데이터 없으면 기존값 유지로 수정
- **수집일 ≠ 거래일** – KRX API에 2/15(비거래일) 요청 시 2/14 데이터 반환. 요청 날짜로 저장하면 이후 2/14 재요청 시 Price 노드 중복 → KRX가 반환하는 `actual_dates`를 XCom으로 전파
- **수동 트리거 시 중복 알림** – `INSERT ON CONFLICT DO NOTHING` 으로 DB는 면등적이었으나 `pg_notify` /Discord는 항상 발행 → `rowcount` 확인 후 알림

수동 스냅샷 적용

사용자는 항상 최신 데이터를 보고 싶어 합니다. 자동 갱신 5군데를 제거하고 **"스냅샷 적용"** 버튼으로 전환. 기준일은 `ticker_prices` 의 `MAX(date)` 를 사용해서 공휴일/주말에도 올바른 거래일을 가리킵니다.

- **기준일** – 어떤 거래일의 가격으로 계산했는가 (예: 2/14)
- **스냅샷 시작** – 언제 계산을 수행했는가 (예: 2/15 10:30)

결국 "기준일"과 "수집일"을 분리 하고, `date.today()` 대신 데이터가 알려주는 날짜를 쓰고, 데이터가 없으면 아무것도 하지 않는 것. 이 세 가지로 정리가 됐는데, 만들면서 하나씩 구체화된 부분이라 처음부터 다 예측하기는 어려웠을 겁니다.

그래프 vs 관계형 – 어디에 뭘 넣을까

그래프에 적합

- 관계 탐색 (ETF-종목 보유, 유사 ETF, 태그 분류)
- 시계열 + 관계 (날짜별 보유종목 비중 변화)
- 다대다 관계 (사용자-ETF 워치리스트)

관계형에 적합

- 단순 CRUD (사용자 정보, 포트폴리오, 매수 기록)
- 집계/정렬 중심 조회 (스냅샷, 가치 계산)
- 트랜잭션이 중요한 데이터 (인증, 주문)

대표적인 사례가 가격 데이터입니다. 처음에는 `etf_prices`, `stock_prices` 관계형 테이블에 저장했지만, 챗봇의 모든 도구가 AGE 그래프만 조회하는 구조여서 SQL과 Cypher를 오가야 했습니다. 결국 `Price` 노드와 `HAS_PRICE` 관계를 추가하고 RDB 테이블은 제거했습니다.

데이터 저장소는 "어디서 주로 조회하느냐"에 맞춰야 한다는 교훈을 얻었습니다.

"전부 그래프" 또는 "전부 관계형"이 아니라 데이터 특성에 맞게 섞어 쓰는 게 현실적인 접근. Apache AGE의 장점이 바로 이 분리를 한 DB에서 할 수 있다는 것입니다.

회고 & 마무리

기술 선택 회고

프로젝트를 마치며 각 기술에 대한 솔직한 평가입니다.

Apache AGE

Good

PostgreSQL 하나로 관계형 + 그래프 + 벡터 검색 까지 가능. 인프라 복잡도가 획기적으로 줄어듭니다.

Bad

Cypher 지원이 Neo4j 대비 불완전. ORM 통합 가이드가 거의 없어서 삽질이 많았습니다. 커뮤니티도 아직 작습니다.

Recommend

이미 PostgreSQL을 쓰고 있고, 그래프 쿼리가 전체 워크로드의 일부일 때.

smolagents

Good

도구 정의가 직관적이고, CodeAgent의 코드 생성 방식이 유연합니다. 소스 코드가 읽을 만한 수준으로 작아서 내부 동작을 이해하기 쉽습니다.

Bad

문서가 부족하고, 고급 기능은 직접 구현해야 합니다. 프롬프트 의존도가 높아서 튜닝에 시간이 많이 듭니다.

Recommend

도구 호출 중심의 에이전트를 빠르게 만들고 싶을 때. 복잡한 RAG 파이프라인이 필요하면 LangChain이 나을 수 있습니다.

Apache Airflow

Good

작업 흐름을 Python 코드로 관리할 수 있다는 것 자체가 큰 가치. 웹 UI, 자동 재시도, 병렬 실행, XCom이 설정만으로 동작합니다.

Bad

초기 설정이 번거롭습니다. "Hello World"까지 단계가 많지만, Docker로 한번 세팅하면 이후에는 DAG 파일만 관리.

Recommend

Python 기반 배치 ETL, 정기 데이터 수집 파이프라인. 실시간 데이터 라우팅이 핵심이라면 NiFi가 더 적합합니다.

감사합니다.

질문 받겠습니다.