에이전틱 소프트웨어 설계

AI 에이전트의 행위자성, 툴 계약, 메모리 계층, 신호와 루프를 통한 프로덕션 에이전틱 시스템 구조적 설계 입문

ThakiCloud (Hyojung Han)

에이전틱 소프트웨어 설계

AI 에이전트의 행위자성, 툴 계약, 메모리 계층, 신호와 루프를 통한
프로덕션 에이전틱 시스템 구조적 설계 입문

ThakiCloud (Hyojung Han)

Contents

1 에이전틱 시스템이란 무엇인가

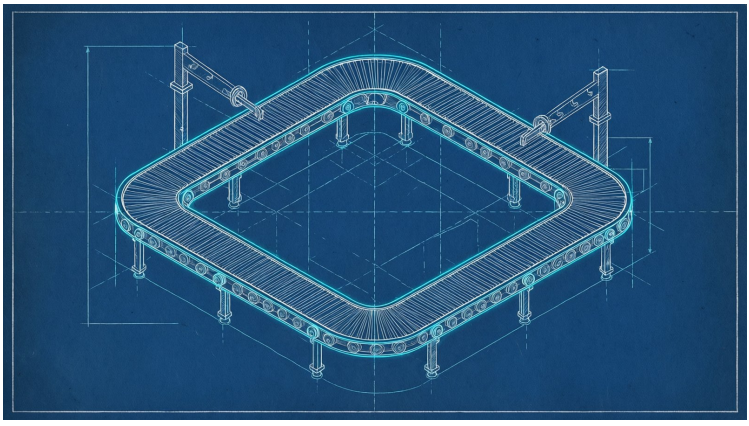

Figure 1.1: 에이전틱 시스템이란 무엇인가

1.1 첫 번째 질문: 응답과 행동의 구분

소프트웨어 시스템이 사람의 물음에 답하는 방식은 대부분 단일 응답으로 끝난다. 질문의 문법을 파악하고 학습된 패턴에서 가장 그럴듯한 답변을 꺼내놓을 뿐이다. 질문과 응답으로 이루어진 1회 세션이 전부이고, 이전 대화의 의미를 다음 결정에 반영하지 않는다.

반면 에이전틱 시스템은 근본적으로 다르다. 에이전트는 입력을 받으면 그것을 해석하고, 행동을 선택하고, 그 결과에 따라 다음 행동을 수정하는 **반복 구조** 위에서 동작한다. 단순화하면 세 단계로 귀결된다. 관찰(Observation)에서는 환경이나 대화의 현재 상태를 읽는다. 판단(Reasoning)에서는 그 관찰을 토대로 다음 행동을 결정한다. 실행(Action)에서는 선택한 행동을 실제로 수행한다. 이 루프가 다시 관찰로 돌아가 결과를 바탕으로 다음 판단을 내린다.

이 구조가 의미하는 바는 명확하다. 에이전트는 시간의 축 위에서 동작하며, 각 순간이 다음 순간의 맥락이 된다. 이것이 행위자성, 즉 agency의 본질이다.

1.2 행위자성과 자율성의 차이

흔히 행위자성과 자율성을 혼동하지만 둘은 뚜렷이 다르다. 자율 시스템은 사전에 정의된 규칙이나 학습된 정책에 따라 움직이므로, 규칙이 변하지 않는 한 행동은 예측 가능하다. 자율주행차가 신호등을 인식해 정지하는 것은 자율적이지만 행위자적이지는 않다. 신호등이라는 입력이 빨간불이라는 출력을 유발했을 뿐, 신호등의 의미나 맥락을 해석해서 다른 행동을 선택한 것은 아니다.

행위자 시스템은 다르다. 행위자는 목표를 인식하고, 그 목표를 달성하기 위해 다양한 수단을 선택하고, 상황에 따라 수단을 바꿀 수 있다. 신호등이 고장났을 때 적절한 판단을 내리고 다른 경로로 우회하는 것은 행위자의 특성이다. 이것은 사전에 프로그래밍된 행위가 아니라 그때그때의 판단에 따른 새로운 행동이다.

실무에서 이 차이를 구별하는 방법은 간단하다. 시스템이 실패한 뒤 자동으로 복구하느냐 마느냐를 보면 된다. 사전에 정의된 예외 처리만 가능한 시스템은 자율적일 뿐 행위적이 아니다. 예상하지 못했던 상황에서 스스로 판단을 내리고 새로운 전략을 구성하는 시스템에서만 행위자성이 나타난다.

1.3 반복 루프와 종료 조건

에이전틱 시스템의 핵심이 반복 구조라고 했다. 하지만 아무리 강력한 에이전트도 무한히 실행될 수는 없다. 따라서 종료 조건의 설계가 아키텍처 수준에서 결정되어야 한다.

종료 조건은 크게 네 가지로 나뉜다. 가장 명확한 것은 목표 달성이다. 에이전트가 설정한 목표를 완전히 이루면 루프가 멈춘다. 다만 복잡한 목표일수록 달성 여부를 판단하는 일 자체가 쉽지 않다. 다음으로 자원 소진을 들 수 있다. 시간, 비용, 토큰 등 사전에 정한 자원이 일정 수준 이하로 떨어지면 루프가 종료된다. 세 번째는 무한 반복 감지다. 같은 관찰-판단 패턴이 일정 횟수 이상 되풀이되면 루프를 강제로 중단하는데, 이는 에이전트가 동일한 판단에 갇혔을 때 작동하는 안전장치다. 마지막은 인간 개입이다. 특정 단계에서 인간의 승인을 요청하거나 의사를 물어 루프를 잠시 멈추게 할 수 있다.

이 네 가지 조건을 어떻게 조합하느냐가 에이전틱 시스템의 성격을 결정한다. 목표 달성만으로는 부족하다. 에이전트가 목표를 잘못 이해하면 무한 루프에 빠질 수 있기 때문이다. 자원 소진 조건을 함께 배치하면 어느 정도 안전장치가 되지만, 에이전트가 유망한 방향을 탐색하다가 갑자기 중단될 위험도 생긴다. 그래서 실무에서는 최소 두 가지 이상의 종료 조건을 함께 운용한다.

1.4 단일 응답 시스템과의 결정적 차이

기존의 많은 AI 시스템은 질문-응답 패러다임에 기반하여 설계되었다. 챗봇이 질문을 받으면 적절한 답변을 생성해 돌려준다. 문장을 입력하면 번역문을 출력한다. 이미지를 입력하면 설명문을 반환한다. 모두 입력-출력의 1회 매핑이 전부이며, 이전 출력 결과가 이후 입력에 영향을 주지 않는다.

이 패러다임이 실패하는 상황은 명확하다. 복잡한 작업을 단일한 응답으로 처리할 수 없을 때이다. 예를 들어 사용자가 "내 일정을 관리해줘"라고 했다고 하자. 단일 응답 시스템은 오늘의 일정을 알려줄 수 있을 뿐이다. 일정을 확인하고, 우선순위를 판단하고, 시간표를 조정하고, 알림을 설정하고, 충돌을 감지해 수정하는 일련의 행동을 연속적으로 수행하려면

에이전틱 구조가 필수다.

에이전틱 구조가 필요한 또 다른 영역은 변화하는 환경에 맞춘 행동의 수정이다. 금융 거래 에이전트를 생각해보자. 시장 상황이 바뀌면 같은 목표(수익 극대화)라도 취해야 할 행동이 달라진다. 단일 응답 시스템은 시장 데이터를 입력받아 추천을 출력할 뿐이지만, 에이전틱 시스템은 시장 변화를 관찰하고, 그에 따라 전략을 수정하고, 수정을 실행에 옮기는 과정을 반복한다.

1.5 이 책의 방향

이 책은 에이전틱 시스템을 설계하고 프로덕션 환경에서 운영하는 데 필요한 원칙과 패턴을 다룬다. 단순히 에이전트를 만드는 방법이 아니라, 에이전트가 도구로 환경과 상호작용하고, 기억으로 맥락을 유지하며, 신호와 루프로 목표를 추구하는 구조를 설계하는 방법을 중점적으로 설명한다.

1장에서는 에이전틱 시스템의 기본 개념인 행위자성과 자율성의 차이를 명확히 했다. 2장에서는 에이전트가 환경과 상호작용하는 창구인 툴의 설계 계약을 다룬다. 3장에서는 에이전트의 기억이 어떻게 구조화되어야 하는지를 설명한다. 4장에서는 에이전틱 시스템의 핵심인 루프의 엔지니어링과 멀티 에이전트 협업의 아키텍처를 짚는다.

에이전틱 시스템은 단순한 기술이 아니라 소프트웨어가 세상을 인식하고 개입하는 방식을 근본적으로 바꾸는 패러다임이다. 이 책이 그 변화를 설계자로서 맞이하는 데 필요한 사고의 틀을 제공하길 바란다.

2 툴 설계 계약

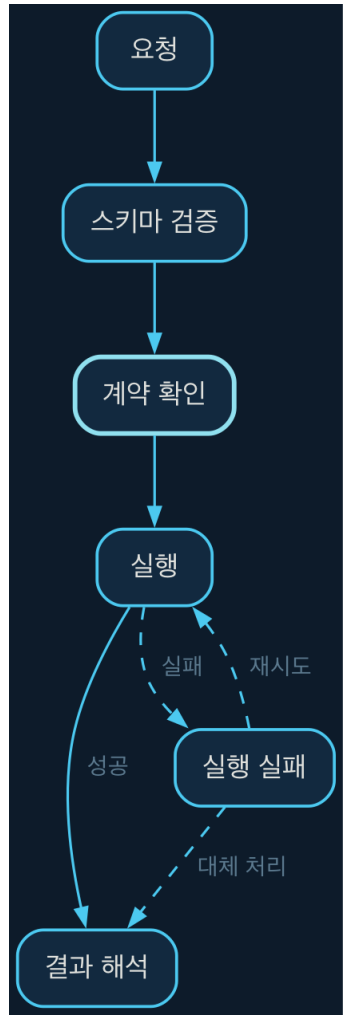

Figure 2.1: 툴 설계 계약

2.1 툴이란 무엇인가

에이전트가 환경과 상호작용하는 방법은 다양하다. 기억을 읽고 쓰거나, 환경에서 데이터를 가져오거나, 외부 시스템에 명령을 내리거나, 다른 에이전트와 통신한다. 이 모든 상호작용의 창구를 툴이라고 부른다.

툴은 에이전트의 능력 범위를 결정한다. 에이전트 자체가 아무리 강력해도 툴이 빈약하면 환경에 대한 영향력이 제한된다. 반대로 툴이 풍부하더라도 그것을 어떻게 사용할지 판단하는 에이전트의 추론 능력이 받쳐주지 못하면 도구만 잔뜩 든 바보가 된다. 그래서 툴 설계와 에이전트 추론 능력의 균형이 아키텍처 전체의 품질을 결정한다.

2.2 툴 스키마의 구조

툴은 크게 세 가지 요소로 구성된다. 스키마는 툴의 입력을 정의한다. 이름과 설명으로 툴의 목적을 표현하고, 파라미터 목록으로 입력 형식을 지정한다. 계약은 에이전트가 툴을 호출할 때 지켜야 할 조건이다. 권한, Rate Limit, 전제 조건 등이 포함된다. 결과 포맷은 툴이 반환하는 출력의 구조를 정의하는데, 에이전트는 이 출력을 해석해 다음 행동을 결정하므로 일관된 결과 포맷이 매우 중요하다.

스키마 설계에서 가장 흔한 실수는 추상화 수준의 불일치다. 너무 추상적인 툴은 범용처럼 보이지만 실제로는 에이전트가 호출할 방법을 찾지 못한다. 반대로 너무 구체적인 툴은 좁은 상황에만 동작하고 일반성이 부족하다. 좋은 툴 설계는 에이전트의 추론 수준에 맞춘 추상화 계층을 갖춘다.

예를 들어보자. 일정 관리 에이전트에서 “일정 생성”이라는 툴 하나만 제공하는 것과, “시간 확인”, “장소 확인”, “참석자 확인”, “충돌 감지”, “일정 등록”이라는 다섯 개의 툴을 제공하는 것은 근본적으로 다른 설계 선택이다. 전자는 에이전트가 일정의 모든 구성 요소를 스스로 판단하라고 요구한다.

후자는 각 판단을 분리된 단위로 제공하고, 에이전트가 필요한 단계를 조합해 목적을 달성하도록 돕는다.

2.3 툴 계약의 설계 원칙

툴을 단순히 기능을 노출하는 수단으로 보지 말고, 에이전트와의 협업 계약을 디자인하는 관점에서 접근해야 한다. 계약 설계의 핵심 원칙은 네 가지다.

첫째는 최소 놀람 원칙이다. 툴의 동작은 이름에서 기대할 수 있는 범위를 벗어나지 말아야 한다. “파일 삭제”라는 툴이 파일을 숨김 처리하거나 다른 위치로 이동시키면 에이전트는 잘못된 mental model을 형성한다. 이름과 설명만으로 동작이 정확히 예측되어야 한다.

둘째는 멱등성이다. 동일한 입력에 대해 동일한 결과를 반환해야 한다. 이 성질 덕분에 에이전트는 계획을 세울 때 툴의 결과를 예측할 수 있고, 실패 시 복구 전략도 쉽게 구축한다. 다만 모든 툴이 완전한 멱등성을 갖는 것은 아니다. “상태 변경” 툴은 호출할 때마다 상태가 바뀌므로 단순한 멱등성이 보장되지 않는다. 이런 경우에는 결과를 예측 가능한 상태 변화로 설계하거나, 에이전트가 결과를 해석해 다음 단계를 판단하도록 계약을 설계한다.

셋째는 명확한 에러 표현이다. 툴이 실패했을 때 반환하는 에러 메시지는 에이전트가 다음 행동을 판단하는 데 직접적인 영향을 미친다. 그래서 에러 메시지는 단순히 “실패했다”는 사실만 전달해서는 안 되고, 왜 실패했는지와 에이전트가 취할 수 있는 대안까지 제시해야 한다. “권한 부족”이라는 에러는 에이전트에게 권한을 확보하라는 메시지다. “대상이 존재하지 않는다”는 에이전트에게 대상을 먼저 생성하라는 메시지다.

넷째는 스키마 문서화의 정확성이다. 툴의 파라미터 설명이 실제 동작과 다르면 에이전트는 잘못된 추론을 한다. 예를 들어 “priority” 파라미터가

1에서 5까지의 정수이고 5가 최고 우선순위라고 설명했는데 실제 구현에서는 5가 최저라면, 에이전트는 역방향으로 우선순위를 부여하게 된다. 스키마 문서는 구현과 완전히 일치해야 하고, 구현이 변경되면 반드시 문서도 갱신해야 한다.

2.4 에러 복구와 Graceful Degradation

모든 툴 호출이 성공하리라는 기대는 비현실적이다. 네트워크 단절, 서비스 장애, Rate Limit 초과, 입력 검증 실패 등 다양한 원인으로 툴 호출은 실패할 수 있다. 에이전틱 시스템에서는 이러한 실패를 어떻게 처리하느냐가 신뢰성의 핵심이다.

기본적인 실패 처리는 세 단계로 나뉜다. 재시도는 동일한 입력으로 다시 호출하는 방식으로, Rate Limit이나 일시적 네트워크 문제에 유효하다. 다만 무제한 재시도는 무한 루프나 자원 낭비로 이어지므로 재시도 횟수에 상한을 설정해야 한다. 대체 툴은 주요 툴이 실패했을 때 같은 목적을 달성할 수 있는 예비 툴을 호출하는 방식이다. 데이터베이스 툴이 실패했을 때 캐시에서 유사한 데이터를 반환하는 것이 한 예다. 축소 범위는 기능을 줄여서라도 동작을 이어가는 전략이다. 전체 일정을 관리하는 툴이 실패하면 일시적으로 “오늘 일정만 조회”라는 축소된 기능만 동작시키거나, 사용자에게 서비스 일시 정지를 공지한다.

이 세 가지 전략을 어떤 상황에 어떤 순서로 적용할지 결정하는 것을 failure recovery planning이라고 한다. 에이전틱 시스템을 설계할 때는 각 툴에 대해 이 플랜을 미리 구성해두어야 한다. 그래야 런타임에 에이전트가 실패를 마주했을 때 즉석에서 판단하지 않고, 사전에 계획된 복구 전략을 그대로 실행할 수 있다.

2.5 툴 조합과 의존성 관리

복잡한 작업을 수행하려면 에이전트가 여러 툴을 순서대로 또는 병렬로 호출해야 하는 경우가 많다. 이때 툴 간의 의존성을 어떻게 관리하느냐가 핵심 문제가 된다.

순서 의존성은 이전 툴의 출력이 다음 툴의 입력이 되는 경우다. 파일 목록을 가져온 후 특정 파일의 내용을 읽는 과정이 대표적이다. 이 경우 에이전트가 아니라 시스템 차원에서 의존성을 관리하는 편이 효과적이다. 에이전트는 의존성을 의식하지 않고 "파일 내용을 확인해서 분석해줘"라고만 지시하면, 시스템이 파일 목록 -> 파일 내용 읽기 -> 분석 순서를 알아서 구성한다.

병렬 의존성은 여러 툴이 동시에 실행 가능하지만 그 결과를 통합해야 하는 경우다. 세 가지 다른 API에서 각각 통화 정보를 가져와 합산하는 것이 그 예다. 이 경우 각 툴 호출의 결과를 취합하는 aggregator의 설계가 필요하다. 각 툴이 동일한 포맷으로 결과를 반환하도록 계약하면 취합이 간단해진다.

공유 상태 의존성은 여러 툴이 동일한 상태를 읽거나 수정하는 경우다. 전형적인 예는 데이터베이스 툴들의 concurrent 접근이다. 이 경우 트랜잭션의 isolation level을 설계에 반영해야 하며, 낙관적 락킹과 비관적 락킹 중 어떤 전략을 적용할지 결정해야 한다.

2.6 툴 설계자의 멘탈 모델

툴 설계자는 무엇보다 에이전트의 관점에서 생각해야 한다. 에이전트는 툴의 내부가 어떻게 구현되어 있는지 모른다. 이름, 설명, 스키마, 실행 결과만으로 툴을 이해한다. 따라서 툴 설계의 질은 에이전트가 툴을 오용하지 않고 정확히 의도한 대로 사용하는 비율로 측정할 수 있다.

그러므로 툴을 설계한 후에는 에이전트가 스키마만으로 툴의 동작을 예측해보는 과정을 권장한다. 에이전트가 설명을 잘못 해석했다면 설명을

수정해야 한다. 설명이 정확한데도 에이전트가 잘못 사용했다면, 툴의 추상화 수준이나 인터페이스가 에이전트의 mental model과 맞지 않는 것이다. 이 피드백 루프로 툴과 에이전트의 협업 계약을 지속적으로 개선해 나간다.

3 메모리 계층의 설계

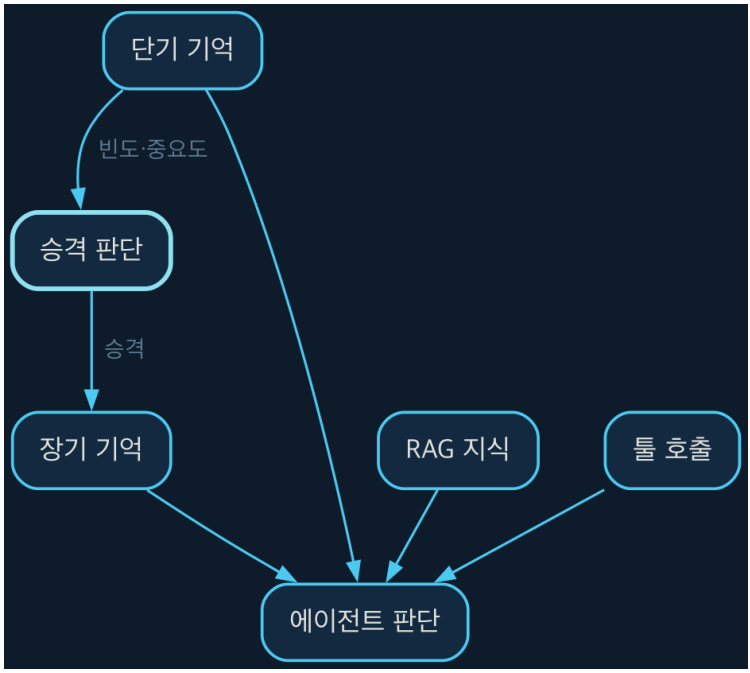

Figure 3.1: 메모리 계층의 설계

3.1 에이전트 메모리의 본질

에이전트가 지속적으로 학습하고 환경에 적응하려면 기존 대화의 맥락을 기억해야 한다. 하지만 컨텍스트 윈도우는 유한해서 모든 정보를 항상 메모리에 담아둘 수는 없다. 그래서 메모리 계층 설계는 에이전틱 시스템에서 핵심적인 위치를 차지한다.

메모리 계층 설계는 정보의 생애주기를 관리하는 일이다. 정보는 생성되고 저장되고 검색되고 노화하다가 결국 삭제된다. 각 단계에서 어떤 정보를

어떤 형식으로 유지할지, 언제 삭제할지를 정하는 것이 메모리 아키텍처의 역할이다.

가장 단순한 접근은 모든 대화를 그대로 유지하는 것이다. 구현은 쉽지만 에이전트의 컨텍스트가 금방 포화한다. 더 정교한 접근은 중요도 점수를 매겨 필요한 정보만 남기는 방식이다. 메모리는 효율적으로 쓸 수 있지만, 중요도를 판단하는 기준이 불명확하면 에이전트가 엉뚱한 판단을 내리게 된다.

좋은 메모리 계층 설계는 명시적인 선택이 쌓여 만들어진다. 어떤 정보를 어떤 형식으로 유지할지 신중하게 정하고, 그 결정이 에이전트의 행동에 어떤 영향을 미치는지 직접 실험해 확인해야 한다.

3.2 단기 기억과 장기 기억의 분리

메모리 계층을 설계할 때 가장 기본이 되는 구분은 단기 기억과 장기 기억을 나누는 것이다. 단기 기억은 지금 진행 중인 작업에 바로 필요한 정보만 담는다. 현재 대화의 맥락, 진행 중인 작업의 상태, 즉각적인 목표가 여기에 해당한다. 장기 기억은 지금 당장은 필요 없지만 나중에 쓸모 있을 정보를 저장한다. 이전 대화에서 파악한 사용자 취향이나 반복적으로 쓰이는 절차적 지식, 그동안 쌓인 경험이 여기에 속한다.

이 둘을 나누는 이유는 명확하다. 컨텍스트 윈도우가 유한한 상황에서 항상 필요한 정보와 가끔 필요한 정보를 같은 수준으로 다루면 전자의 밀도가 희석된다. 단기 기억의 밀도를 지키려면 장기 기억을 따로 저장하고 필요할 때 꺼내 쓰는 구조가 필수다.

구현에서 이 분리의 핵심은 검색 함수의 설계에 있다. 단기 기억에서 장기 기억으로 정보를 옮기는 기준은 빈도와 중요도, 시간이다. 자주 언급되는 개념은 장기 기억으로 승격하고 한 번만 나온 세부 사항은 버리는 것이 일반적이다. 중요도는 사용자의 명시적 피드백이나 작업 결과의 성공

여부로 판단할 수 있고, 시간은 오래된 정보일수록 중요성이 낮아진다는 사실을 반영한다.

3.3 검색 증강 생성과의 관계

검색 증강 생성, 즉 RAG는 에이전틱 시스템에서 큰 비중을 차지한다. RAG는 외부 지식을 검색해 컨텍스트에 추가해 에이전트가 다룰 수 있는 지식의 범위를 넓힌다. 다만 RAG와 에이전트 메모리는 같은 문제를 다른 각도에서 풀고 있을 뿐이다.

RAG는 명시적 질문에 관련 정보를 검색하는 pull 방식이다. 에이전트가 질문하면 시스템이 그에 맞는 지식을 찾아 제공한다. 반면 에이전트 메모리는 시스템이 능동적으로 정보를 기억하는 push 방식이다. 대화가 진행되는 동안 시스템이 알아서 중요한 정보를 골라 저장한다.

둘을 결합하는 가장 효과적인 구조는 다음과 같이 나눌 수 있다. 에이전트의 동작 범위 안에 있는 정보, 예를 들어 이전 대화의 맥락이나 작업 진행 상황은 에이전트 메모리에서 관리한다. 동작 범위 밖에 있지만 검색으로 접근할 수 있는 정보, 예를 들어 사내 문서나 상품 카탈로그는 RAG로 가져온다. 둘 중 어디에도 속하지 않는 정보, 예를 들어 에이전트가 실시간으로 마주하는 환경 정보는 툴 호출로 직접 얻는다.

이 구조에서 RAG는 에이전트 메모리를 대체하지 않고 보완한다. 에이전트 메모리가 에이전트 개인의 경험이라면, RAG는 에이전트가 접근할 수 있는 공동의 지식 기반인 셈이다.

3.4 메모리 구조화의 패턴

메모리 설계에서 또 다른 중요한 선택은 구조화 수준이다. 여기에는 두 가지 극단이 있다. 하나는 전문 저장이다. 모든 대화를 그대로 저장하고 검색할 때 전부를 비교 대상으로 삼는 방식이다. 구현이 간단하고 정보가 손실되지

않는다는 장점이 있지만 검색 효율이 낮고 컨텍스트 크기가 늘어나면서 성능이 떨어진다.

다른 하나는 구조화된 저장이다. 대화의 핵심을 미리 정의한 스키마로 추출해 저장하는 방식이다. 예를 들어 사용자 취향은 "{preferred_language, topic_interests, communication_style}"라는 구조로 저장한다. 검색 효율이 높고 에이전트가 정보를 이해하기 쉽다는 장점이 있지만, 구조화하는 과정에서 정보가 일부 손실될 수 있다.

실무에서는 이 두 극단 사이 어딘가를 선택하는 경우가 많다. 핵심 정보는 구조화된 형식으로 저장하고 나머지는 전문으로 저장하되 인덱싱을 적용하는 방식이 효과적인 균형점이다. 예를 들어 "{task: '코드 리뷰', language: 'python', files: ['/src/main.py', '/tests/test.py']}"라는 구조화된 메타데이터를 두고, 각 파일의 변경 이력은 전문 그대로 유지하는 식이다.

메모리 구조화에서 또 다른 중요한 측면은 시계열화다. 메모리는 시간이 흐르면서 쌓인다. 그래서 메모리 구조는 시간 순서로 정리하면서도 시간 범위별 검색이 가능해야 한다. 예를 들어 "최근 한 달간 이 사용자와의 대화에서 반복적으로 언급된 주제"를 검색하려면 시간 범위 필터와 빈도 기반 랭킹 기능이 함께 필요하다.

3.5 에이전트의 학습과 컨텍스트 관리

에이전트가 경험으로 행동의 품질을 높이는 것은 메모리 계층의 중요한 목표다. 단순한 정보 저장이 아니라 패턴 인식과 행동 개선을 포함하는 능동적인 과정이다.

패턴 인식은 반복적으로 나타나는 상황에서 어떤 응답이 효과적이었는지 저장하는 것이다. 예를 들어 특정 유형의 버그 리포트에서 "문제를 재현해본다"는 행동이 효과적이었다면, 이 경험을 메모리에 저장해 비슷한

상황이 다시 생겼을 때 같은 행동을 취하도록 한다. 그러면 에이전트의 판단 속도가 빨라지고 성공적인 행동을 다시 재현할 가능성도 커진다.

행동 개선은 실패에서 배우는 것이다. 어떤 행동이 실패로 이어졌다면, 그 경험을 메모리에 부정적 피드백으로 기록해 비슷한 상황에서 같은 행동을 피하도록 한다. 이것은 reinforcement learning의 원리와 비슷하지만, 에이전트 메모리에서는 명시적인 지식으로 표현된다는 점이 다르다.

컨텍스트 관리의 핵심은 현재 작업에 필요한 정보를 골라내고 불필요한 정보를 걷어내는 결정에 있다. 이것은 단순한 메모리 용량 관리를 넘어 에이전트의 주의력과 판단 품질을 좌우하는 요소다. 잘못된 정보를 컨텍스트에 포함시키면 에이전트는 그릇된 판단을 내리고, 필요한 정보를 빠뜨리면 불완전한 판단을 내린다. 결국 메모리 계층 설계는 이 선별과 제거의 기준을 어떻게 정의하느냐의 문제로 귀결된다.

4 신호, 루프, 아키텍처

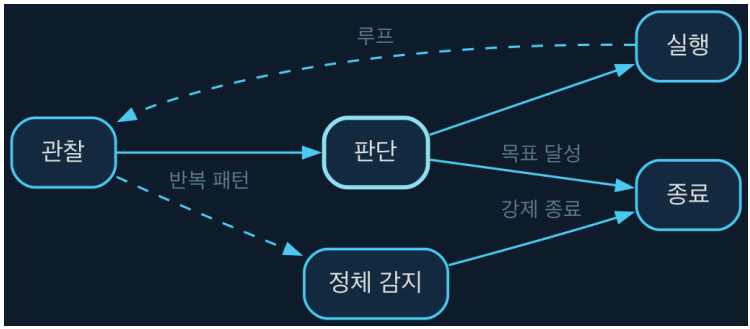

Figure 4.1: 신호, 루프, 아키텍처

4.1 관찰-판단-실행 루프의 구조

에이전틱 시스템의 핵심은 관찰-판단-실행이라는 세 단계가 순환하는 루프다. 이 루프가 어떻게 구성되느냐에 따라 에이전트의 행동 방식이 결정된다.

관찰 단계에서는 에이전트가 환경의 현재 상태를 파악한다. 이것은 사용자의 입력일 수도 있고, 툴 호출의 결과일 수도 있고, 주기적인 상태 점검의 결과일 수도 있다. 중요한 것은 관찰이 에이전트의 내적 상태와 환경의 외적 상태를 모두 포함한다는 점이다. 에이전트가 현재 진행 중인 작업의 상태를 인지하는 것도 관찰의 일부다.

판단 단계에서는 관찰 결과를 토대로 다음 행동을 결정한다. 규칙 기반의 결정일 수도 있고, 학습된 정책에 따른 결정일 수도 있다. 판단의 입력은 관찰의 출력이며, 판단의 출력은 실행할 행동의 목록이다.

실행 단계에서는 선택된 행동을 실제로 수행한다. 툴 호출, 다른 에이전트와의 통신, 상태 업데이트 등이 여기에 해당한다. 실행의 결과는

다시 관찰 단계로 피드백되어 다음 판단의 입력으로 쓰인다.

이 루프에는 세 가지 특성이 있다. 판단은 로컬하게 이루어진다. 각 판단은 현재 관찰만을 입력으로 삼으며, 과거의 판단과 현재 판단 사이에 명시적 연결 고리는 없다. 반복은 유한하다. 루프는 무한히 돌지 않고 종료 조건이 만족되면 멈춘다. 그리고 각 실행의 결과가 다음 관찰에 영향을 주면서, 루프 전체가 하나의 적응적 시스템으로 작동한다.

4.2 종료 조건의 설계와 검증

루프 엔지니어링에서 가장 중요한 결정은 종료 조건의 설계다. 잘못된 종료 조건은 두 가지 위험을 낳는다. 하나는 조기 종료로, 에이전트가 목표를 완전히 달성하기 전에 멈춘다. 다른 하나는 무한 루프로, 에이전트가 목표를 달성했음에도 계속 실행한다.

조기 종료는 주로 평가 기준의 불명확함에서 비롯된다. "충분히 좋은" 결과를 정의할 수 없으면, 에이전트는 언제 멈춰야 할지 알지 못한다. 따라서 종료 조건의 설계는 목표의 성격에 따라 달라진다. 명시적 정량 조건이 있는 목표는 종료 조건도 명확하다. 예를 들어 "100건의 데이터 처리"는 처리 건수가 100에 도달하면 종료다. 반면 "가장 좋은 방법을 찾아라"와 같은 목표는 "가장 좋은"을 어떻게 정의할 것인지가 불명확하므로 종료 조건도 모호해진다.

무한 루프는 판단의 품질이 충분하지 않을 때 주로 발생한다. 에이전트가 동일한 판단을 반복하면서 진전을 이루지 못하면 루프는 끝나지 않는다. 이를 막으려면 두 가지 장치가 필요하다. 하나는 반복 횟수 제한으로, 같은 판단 패턴이 일정 횟수 이상 반복되면 무조건 종료시킨다. 다른 하나는 진전 척도로, 반복마다 목표 달성 방향으로 유의미한 진전이 있었는지를 측정해 진전이 없으면 멈춘다.

종료 조건의 검증은 에이전틱 시스템의 품질을 판단하는 핵심 기준이다.

종료 조건을 사전에 정의하고, 시스템이 종료되었을 때 목표가 실제로 달성되었는지를 사후 검증하는 과정이 반드시 필요하다. 이것은 테스트의 영역이고, 가능하면 자동화된 검증 체계를 갖추는 편이 낫다.

4.3 멀티 에이전트 협업의 구조

단일 에이전트의 능력에는 한계가 있다. 더 복잡한 작업을 수행하려면 여러 에이전트가 협업하는 구조가 필요하다. 멀티 에이전트 시스템에서는 에이전트 간의 통신 프로토콜과 책임의 분배가 핵심 설계 결정이 된다.

통신 프로토콜은 에이전트 간에 정보를 교환하는 방식의 계약이다. 동기 통신은 보내는 에이전트가 응답을 기다린 후 다음 행동으로 넘어간다. 비동기 통신은 메시지를 보낸 후 기다리지 않고 다른 작업을 계속한다. 각 방식은 상황에 따라 적합성이 다르다. 긴급한 조치가 필요한 상황에서는 동기 통신이 적합하고, 여러 에이전트가 독립적으로 작업을 수행하는 상황에서는 비동기 통신이 적합하다.

책임의 분배는 작업을 어떻게 여러 에이전트에 할당할 것인지를 결정한다. 기능별 분배는 각 에이전트가 특화된 기능 영역을 담당한다. 예를 들어 분석 에이전트, 계획 에이전트, 실행 에이전트가 각각 전문 분야를 맡는 방식이다. 단계별 분배는 작업의 단계를 따라 각 에이전트가 담당한다. 예를 들어 첫 번째 에이전트가 문제를 파악하고, 두 번째 에이전트가 해결책을 제안하고, 세 번째 에이전트가 실행하는 방식이다. 혼합형 분배는 기능별과 단계별을 조합한다.

멀티 에이전트 협업에서 특히 중요한 것은 충돌 해결이다. 여러 에이전트가 동시에 같은 자원에 접근하거나 상충하는 목표를 추구할 때 충돌이 발생한다. 이를 해결하는 방법으로 세 가지가 있다. 우선순위 기반 해결은 사전에 정의된 우선순위에 따라 충돌 시 높은 우선순위의 에이전트가 승리한다. 협상 기반 해결은 충돌에 관여한 에이전트들이 서로 정보를 교환하고 타협점을 찾는 방식이다. 중앙 조정자 방식은 충돌 상황을

판단하고 해결책을 지시하는 중앙 조정 에이전트를 두는 방식이다.

4.4 프로덕션 에이전틱 시스템의 검증

에이전틱 시스템이 프로덕션 환경에서 동작하려면, 설계 단계에서의 검증과 운영 단계에서의 모니터링이 모두 필요하다.

설계 단계의 검증은 시스템이 의도한 대로 동작하는지를 확인하는 과정이다. 여기에는 세 가지 핵심 검증이 있다. 동작 검증은 시스템이 설계된 대로 작동하는지를 확인한다. 입력에 따른 출력, 종료 조건의 동작, 에러 상황에서의 복구 행동 등을 테스트한다. 성능 검증은 시스템이 정해진 자원 내에서 목표를 달성하는지를 확인한다. 시간, 비용, 토큰 사용량 등이 설계 기준을 벗어나지 않는지를 측정한다. 안전성 검증은 시스템이 위험한 행동을 하지 않는지를 확인한다. 예측 불가능한 상황에서의 행동, 자원 남용, 그리고 의도하지 않은 부작용이 없는지를 테스트한다.

운영 단계의 모니터링은 시스템이 프로덕션에서 의도한 대로 동작하는지를 지속적으로 확인하는 과정이다. 핵심 모니터링 항목은 네 가지로 볼 수 있다. 가장 먼저 루프 빈도를 본다. 에이전트가 단위 시간당 몇 번의 판단-실행 사이클을 도는지를 추적하는데, 빈도가 너무 높으면 불필요한 작업을 반복하고 있을 가능성이 있고 너무 낮으면 에이전트가 막혀 있을 가능성이 있다. 다음으로는 결정 품질을 본다. 에이전트의 결정이 목표 달성에 얼마나 기여하는지를 추적하며, 목표 달성에 걸린 평균 시간과 달성률, 작업 품질 지표로 측정한다. 자원 사용률도 빼놓을 수 없다. CPU, 메모리, 토큰 등 시스템 자원이 얼마나 쓰이는지를 모니터링하는데, 사용량이 설계 한계를 넘으면 시스템 전체 성능이 떨어진다. 마지막으로 에러 비율이 있다. 툴 호출 실패, 예외 상황, 에이전트의 판단 실수가 어느 정도 빈도로 일어나는지를 추적한다.

이 모니터링 결과들이 쌓이면서 에이전틱 시스템은 계속 개선된다. 프로덕션에서 드러난 문제점을 메모리 계층에 반영하고, 루프의 파라미터를

조정하고, 새 툴을 추가하는 작업들이 시스템의 진화로 이어진다.

4.5 아키텍처 결정의 균형점

에이전틱 시스템의 아키텍처를 설계할 때 마주하는 근본적 갈등은 복잡성과 기능성의 균형이다. 더 많은 에이전트를 배치하고, 더 복잡한 통신 프로토콜을 설계하고, 더 정교한 메모리 구조를 구성하면 시스템의 기능성은 높아진다. 하지만 동시에 시스템의 예측 가능성은 낮아지고, 디버깅과 유지보수의 난이도는 높아진다.

좋은 아키텍처는 필요한 기능성을 최소한의 복잡성으로 구현한다. 에이전트 수를 최소화하고, 통신 프로토콜을 최대한 단순하게 유지하고, 메모리 구조가 불필요하게 복잡해지지 않으면서도 필요한 정보를 충분히 담아낼 수 있는 수준으로 설계해야 한다는 뜻이다.

이러한 균형점은 시스템의 목적에 따라 달라진다. 실험적 프로토타입에서는 기능성이 복잡성보다 우선될 수 있다. 프로덕션 시스템에서는 예측 가능성과 유지보수성이 기능성보다 중요할 수 있다. 그러니 아키텍처를 결정할 때는 항상 시스템의 목적과 제약 조건을 함께 고려해야 한다.