AI 에이전트 핸스 설계

프롬프트와 도구 정의를 넘어서

ThakiCloud (Hyojung Han)

AI 에이전트 핸스 설계

프롬프트와 도구 정의를 넘어서

ThakiCloud (Hyojung Han)

Contents

1 왜 모델을 올려도 에이전트가 불안정한가

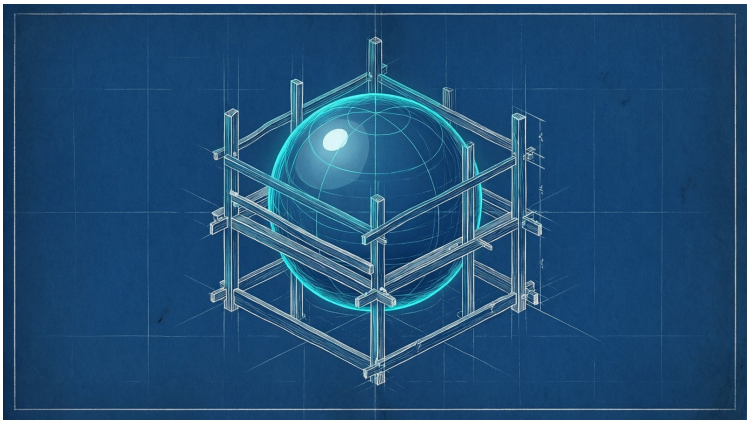

Figure 1.1: 왜 모델을 올려도 에이전트가 불안정한가

1.1 모델 등급은 해답이 아니다

LLM 에이전트를 프로덕션에 배포한 경험이 있는 엔지니어라면 이 패턴을 안다. Sonnet에서 Opus로 모델을 올렸더니 한동안 잘 동작한다. 하지만 며칠이 지나면 같은 입력에 다른 출력이 나온다. 다시 프롬프트를 다듬는다. 또 잘 된다. 그리고 다시 무너진다. 이 악순환의 근원에 모델 업그레이드가 놓여 있다고 생각하기 쉽다. 그러나 실상은 다르다.

모델 등급과 에이전트 신뢰성의 관계는 선형이 아니다. Opus 4.8이 Sonnet 5보다 더 강력한 추론 능력을 가진 것은 사실이다. 그러나 에이전트의 행동 일관성을 결정하는 것은 모델의 추론 능력이 아니라 **핸스**라는 별개의 변수다. 핸스는 에이전트가 어떻게 사고하고 어떤 도구를 쓰며 출력을 어떻게 검증하는지를 정의하는 구조체다. 모델이 추론을 잘한다고 그 추론의 방향까지 올바르게 잡아주지는 않는다.

1.2 핸스 부재의 증상

같은 입력에 다른 출력이 나오는 현상은 모델의 창의성이 아니라 핸스의 부재를 증명한다. 핸스가 부재하면 모델은 매번 스스로 판단해야 한다. 판단은 문맥에 민감하다. 입력의 미세한 차이가 모델의 주의 방향을 바꾸고, 그에 따라 출력이 흔들린다.

예를 들어보자. 파일 목록을 읽고 요약하는 에이전트가 있다. 파일이 5개일 때는 정상 동작한다. 파일이 30개가 되면 출력이 불안정해진다. 모델이 "너무 많다"는 압박을 느끼면서 일부 파일을 건너뛰거나, 요약 깊이를 임의로 조절하기 시작한다. 이때 반응은 모델 등급과 무관하다. Opus도 같은 현상을 보인다. 원인은 모델이 아니라 파일을 한 번에 다룰지 분할할지를 에이전트 구조가 정의하지 않았기 때문이다.

핸스가 부재할 때는 몇 가지 증상이 반복해서 나타난다.

출력 형식부터 흔들린다. 한 번은 JSON으로 반환하고 다음에는 자연어로 반환한다. 모델이 출력 형식을 유추해야 하는 한 이 흔들림은 피할 수 없다.

도구 선택도 일관되지 않는다. 사용자가 "데이터를 확인해줘"라고 하면 어떤 때는 데이터베이스를 쿼리하고 어떤 때는 웹검색을 실행한다. 모델이 스스로 판단해야 하니 판단 기준도 매번 달라진다.

실패했을 때 회복도 안 된다. 한 단계에서 오류가 발생하면 에이전트가 멈추거나 잘못된 방향으로 진행한다. 모델은 실패를 인식할 수는 있어도 실패 후 무엇을 할지는 스스로 결정하지 못한다.

1.3 핸스가 결정하는 세 가지 핵심 축

에이전트 품질을 결정하는 변수는 시스템 프롬프트, 도구 정의, 출력 검증이라는 세 축으로 나뉜다. 이 세 축은 서로 독립적이지 않다. 시스템 프롬프트가 도구 사용의 맥락을 제공하고 도구 정의가 모델의 행동 범위를

제한하며 출력 검증이 그 결과를 최종적으로 확인한다.

이 세 축이 모두 부재하거나 불완전하면 에이전트는 불안정해진다. 하나만 갖춰서는 부족하다. 시스템 프롬프트만 정교하게 써도 도구가 명확하지 않으면 모델은 어떤 행동을 취해야 할지 모른다. 도구 정의만 제대로 해도 출력을 검증하지 않으면 잘못된 결과가 통과한다. 시스템 프롬프트와 도구 정의를 잘 만들어도 세 번째 축이 없으면 에이전트는 실패를 반복한다.

1.4 모델 업그레이드의 진짜 효과

모델을 올렸을 때 에이전트가 좋아지는 것 같아 보일 때는 대개 두 가지 경우다. 하나는 기존 핸스가 불완전했지만 모델의 추론 능력으로 인해 숨겨졌던 것이다. 더 강력한 모델은 그 불완전함을 더 잘 보완한다. 그러나 핸스가 근본적으로 개선된 것이 아니므로 시간 경과와 함께 다시 문제가 드러난다.

다른 하나는 모델 등급 상승과 함께 핸스도 같이 개선된 경우다. 이 경우에만 품질 향상은 지속된다. 그러나 대부분의 팀은 모델만 올리고 핸스는 유지한다. 그래서 "모델을 올렸더니 처음 두 주간은 잘 되다가 다시 무너졌다"는 보고가 흔한 것이다.

모델은 엔진이다. 엔진이 강하면 속도는 빨라지지만 방향을 트는 것은 핸스다. 핸스 없이 엔진만 바꾸는 건 차를 바꿔도 내비게이션이 없는 것과 같다.

2 시스템 프롬프트를 엔진이 아니라 계약으로 설계하기

Figure 2.1: 시스템 프롬프트를 엔진이 아니라 계약으로 설계하기

2.1 프롬프트는 엔진이 아니라 계약이다

시스템 프롬프트를 "모델에게 지시를 내리는 문서"로 이해하는 팀이 많다. 이 관점에서는 프롬프트가 강할수록 모델이 더 잘 따른다고 본다. 그래서 프롬프트는 점점 길고 상세해진다. 하지만 이 접근은 역효과를 낳는다. 프롬프트가 길어질수록 모델은 지시사항 전부를 고르게 반영하지 못한다. 일부 지시는 무시하고 다른 지시에만 과잉 반응하는 식이다.

더 효과적인 방법은 프롬프트를 **계약**으로 설계하는 것이다. 계약에서는 각 당사자가 무엇을 주고받는지 명확히 정의한다. 모델도 마찬가지다. 무엇을 입력으로 받고 무엇을 출력으로 내보낼지, 어떤 상황에서 어떻게 대응해야 하는지를 규정해야 한다. 지시사항을 나열하는 게 아니라 행위자와 계약의 집합을 구조화하는 일이다.

2.2 역할 지시: 단일 초점 원칙

시스템 프롬프트에 역할을 정의할 때 흔한 실수는 여러 역할을 한꺼번에 부여하는 것이다. "너는 데이터를 분석하고 보고서를 작성하며, 필요하면 웹검색도 하고, 최종 결과를 검증까지 하는 AI 어시스턴트다." 이 문장에서 모델은 네 가지 역할을 동시에 수행해야 한다. 역할이 많아질수록 각 역할에 배분되는 주의력은 그만큼 흩어진다.

단일 초점 원칙은 간단하다. 프롬프트 하나에는 역할 하나만 부여한다. 여러 단계가 필요하면 각 단계를 별도의 계약으로 분리하고, 에이전트가 그 계약들을 연결하도록 구조를 설계한다. 단일 프롬프트에 모든 행동을 욱여넣는 것보다 이 방식이 효과적이다. 역할이 하나면 모델은 그 역할에 주의력을 온전히 집중한다.

단일 초점 원칙을 어기는 예시를 보자. "너는 고객 지원 에이전트다. 사용자의 문제를 해결하고, 해결이 불가능하면 티켓을 생성하며, 모든 대화의 감정을 분석해서 만족도를 측정해라." 여기서는 세 가지 책임이

하나로 묶여 있다. 모델은 문제를 해결하는 동시에 감정 분석까지 신경 써야 한다. 두 과제가 충돌하면(사용자가 실망했을 때 더 자세한 해결책을 주는 게 오히려 감정 점수를 낮출 수 있다) 모델은 그 균형을 스스로 잡아야 하는데, 판단 기준은 프롬프트 어디에도 없다.

이걸 분리하면 어떻게 될까. "너는 고객 지원 에이전트다. 사용자의 문제를 해결하는 것이 유일한 목표다." 그리고 별도의 계약으로 "문제 해결 후, 만족도 예측 결과를 생성하라." 이렇게 하면 각 계약은 자기 책임에만 집중하고, 에이전트 구조가 두 계약을 연결한다.

2.3 제약 조건의 표현: 허용이 아니라 금지로

프롬프트에서 제약 조건을 표현하는 방식은 두 가지다. 허용할 행동을 나열하는 방식과 금지할 행동을 나열하는 방식이다. 대부분의 프롬프트는 허용 목록을 쓴다. "답변은 다음 형식을 따라야 한다. 항목 A, 항목 B, 항목 C." 하지만 허용 목록은 예측하지 못한 상황에 대응하지 못한다. 모델이 허용되지 않은 방식으로 응답할 때마다 프롬프트를 갱신해야 하니 프롬프트는 계속 길어진다.

금지 목록을 쓰면 모델의 행동 범위를 더 효과적으로 제한할 수 있다. "절대로 함수를 직접 호출하지 마라. 항상 도구 설명에 명시된 인터페이스를 사용하라." 이 방식은 모델이 하지 말아야 할 일을 명확히 알게 한다. 금지된 행동의 목록은 대개 허용된 행동의 목록보다 짧고, 모델은 짧은 금지 목록을 더 정확히 지킨다.

실제 프로덕션 사례를 보자. 웹검색 에이전트에서 모델이 검색 결과를 그대로 옮겨 답하는 경우가 있다. 이를 막으려면 "검색 결과를 그대로 응답하는 대신, 결과를 기반으로 자신만의 분석을 제공하라"는 허용 지시보다 "검색 결과의 문장을 그대로 복사하여 답변에 포함하지 마라"는 금지 지시가 더 효과적이다. 전자는 모델이 "분석"의 수준을 스스로 판단해야 하고 그 판단은 문맥마다 달라진다. 후자는 행동의 경계를 명확히

그어준다.

2.4 출력 형식의 명시: 예시 없는 명시적 구조

LLM에게 출력 형식을 전달할 때 예시(few-shot)를 함께 주는 건 흔한 관행이다. 예시는 모델에게 원하는 출력의 모양을 시각적으로 보여준다. 하지만 예시에 의존하는 방식에는 문제가 있다. 예시는 특정 입력과 특정 출력의 쌍 하나만 보여준다. 모델은 그 쌍에서 형식을 배우지만, 새로운 입력에 적용할 때 형식이 무너지는 경우가 생긴다. 특히 입력이 예시와 다른 구조를 가지면 모델은 형식을 추측하려 들고, 그 추측은 종종 틀린다.

더 확실한 방식은 예시 없이 출력 형식을 명시적으로 정의하는 것이다. JSON 스키마를 쓰든 마크다운 구조를 쓰든 상관없다. 핵심은 "무엇이 있어야 하는가"를 열거하고, "어떻게 보이는가"는 모델에게 맡기지 않는 것이다.

마크다운으로 출력해야 하는 경우를 보자. 잘못된 방식은 다음과 같다.

```
좋은 예:
# 제목
## 부제목
내용입니다.
```

올바른 방식은 다음과 같다.

```
출력 형식:
- 최상위 헤딩은 하나만 허용
- 헤딩 수준은 H1 -> H2 -> H3 순서로만 증가
- 각 섹션의 첫 문장은 결론을 먼저 제시
- [...]
```

형식 규칙을 열거하면 모델은 문맥과 무관하게 그 규칙을 지킬 수 있다. 입력이 복잡할수록 예시의 효용은 줄고 규칙의 효용은 늘어난다.

프롬프트에서 예시 비중을 줄이고 규칙 비중을 늘리는 것, 그것이 에이전트의 일관성을 높이는 열쇠다.

3 도구 정의의 세밀함: 이름이 행동을 결정한다

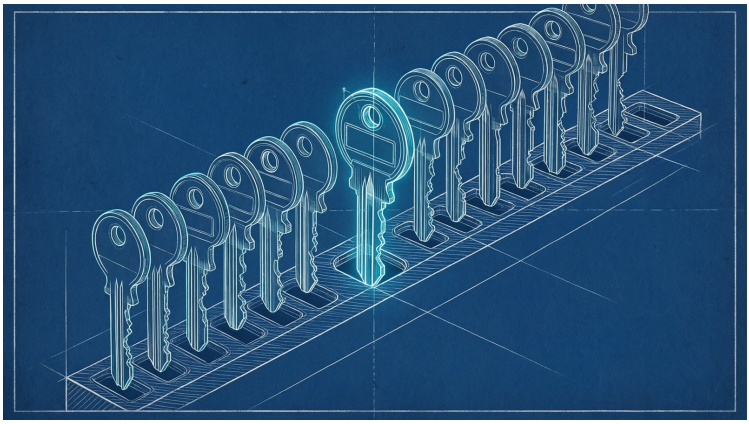

Figure 3.1: 도구 정의의 세밀함: 이름이 행동을 결정한다

3.1 도구 이름은 동사 원형으로

LLM 에이전트에서 도구 정의는 모델이 행동을 결정하는 토대가 된다. 도구 이름이 무엇이냐에 따라 모델이 그 도구를 언제 호출할지가 갈린다. 그런데 많은 팀이 도구 이름을 명사 중심으로 짓는다. "dataLoader", "report-Generator", "searchEngine" 같은 식이다. 이런 이름은 기술적 구현은 반영하지만, 모델에게 언제 그 도구를 써야 하는지는 알려주지 않는다.

도구 이름은 항상 **동사 원형**으로 짓는다. "load_data", "generate_report", "search_web" 식이다. 이런 네이밍은 모델에게 "이 도구를 호출하면 이 행동을 한다"는 정보를 곧바로 전달한다. 도구 이름이 "report-Generator"라면 모델은 리포트를 "생성"해야 한다는 것만 알 뿐, 그 생성이라는 행위에 어떤 입력이 필요한지는 스스로 예측해야 한다.

“generate_report”라면 모델은 필요한 입력이 무엇인지 도구 설명에서 찾아본다.

이 원칙을 지키지 않으면 실무에서 문제가 바로 드러난다. “사용자 데이터를 분석해줘”라는 요청에 모델이 “user_data_analyzer”를 불러야 할지 “process_user_data”를 불러야 할지 헷갈리는 식이다. 두 도구가 비슷해 보여도 이름이 다르면 모델은 그 미묘한 차이를 해석하려 들고, 그 해석은 호출할 때마다 달라질 수 있다. 도구 이름이 명확한 동사면 모델의 판단이 한 방향으로 모인다.

3.2 파라미터는 최소한으로

도구가 여러 파라미터를 요구하면 모델은 그 값을 스스로 정해야 한다. 파라미터가 많아질수록 정확한 값을 채울 확률은 줄어든다. 더 심각한 문제는 따로 있다. 파라미터 값이 불확실하면 모델의 confidence가 떨어지고, 결국 도구 호출 자체를 회피하거나 잘못된 기본값을 쓰게 된다.

파라미터 최소화는 도구 설계의 핵심 원칙이다. 필수 파라미터만 노출하고 선택적 파라미터는 되도록 빼야 한다. 필수 파라미터라도 모델이 판단하기 어려운 값은 기본값으로 미리 채워둔다.

예를 들어보자. 데이터베이스 쿼리 도구를 설계한다고 하자. 나쁜 설계는 이렇다.

```
search_users(table: string, columns: list[string], filters:
↪  dict, sort_by: string, limit: int, offset: int)
```

모델은 table은 정할 수 있어도 columns, filters, sort_by, limit, offset을 한꺼번에 정확히 채우기는 어렵다. 쿼리 결과가 너무 많을 때 limit를 높여야 할지 filters를 좁혀야 할지도 모델이 판단해야 하는데, 그 판단은 애초에 모델의 몫이 아니다.

좋은 설계는 이렇다.

```
search_users(query: string, max_results: int = 10)
```

query만 필수고 나머지는 기본값이 적용된다. 모델은 사용자 질문에서 쿼리를 뽑아내고, 그 쿼리로 충분한 결과를 얻을 수 있다고 판단되면 이 도구를 호출한다. 복잡한 필터링이 필요할 때는 모델이 고급 쿼리를 query에 담아 대신 전달하면 된다. 도구 호출의 입도는 이렇게 자연스럽게 정해진다.

3.3 에러 시나리오를 도구 계약에 포함해야 하는 이유

도구를 정의할 때 성공 시나리오만 설명하는 팀이 많다. "이 도구는 성공하면 이런 결과를 반환한다"는 식이다. 그러나 에이전트의 실패는 대부분 도구 실행 단계에서 일어난다. 네트워크 오류, 권한 부족, 타임아웃, 잘못된 입력 형식 같은 상황들이다. 이런 상황을 도구 정의에 명시하지 않으면 모델은 실패에 어떻게 대응할지 스스로 궁리해야 한다.

실패 대처를 문서화하는 표준 포맷을 도구 계약에 넣어야 한다. 성공 결과의 형태와 함께 가능한 실패 유형, 그때의 반환 값을 정의하는 것이다. 모델은 이 정보를 보고 "이 도구가 실패하면 이 방향으로 다시 시도하거나 실패를 상위로 전파한다"는 걸 익힌다.

실제 구현 예시를 보자. 파일 읽기 도구는 이렇게 정의한다.

```
read_file(path: string) -> string
성공: 파일 내용을 문자열로 반환
실패 유형:
 - FILE_NOT_FOUND: 파일이 존재하지 않음 -> "파일을 찾을 수
 ↪ 없습니다" 반환
 - PERMISSION_DENIED: 읽기 권한 없음 -> "접근 권한이 없습니다"
 ↪ 반환
```

```
- TIMEOUT: 10초 이내 미완료 -> null 반환, 호출자에게 타임아웃
↪ 알림
```

이런 정의가 없으면 모델은 파일을 못 읽었을 때 재시도할지, 다른 방법을 시도할지, 그냥 응답을 만들어낼지를 알아서 판단한다. 그 판단은 모델마다 다르고, 같은 모델이라도 입력에 따라 달라진다. 계약에 에러 시나리오가 정의돼 있으면 모델의 대응이 일관되게 유지된다.

3.4 불필요한 도구를 숨기는 것

도구를 많이 정의할수록 모델의 행동 범위가 넓어질 것 같지만 실상은 반대다. 도구가 많아지면 모델은 "이 상황에서 어떤 도구가 적절한가"를 판단하는 부담을 떠안는다. 그만큼 판단 오류가 날 확률도 높아진다. 불필요한 도구를 없애거나 숨기면 이 판단 오류가 줄어든다.

여기서 숨기기란 도구 정의를 아예 없앤다는 뜻이 아니다. 계약에는 존재하되 모델이 접근할 수 있는 도구 목록에서만 빼는 것이다. 특정 역할의 에이전트에게는 그 역할과 관련된 도구만 노출하면 된다.

예를 들어보자. 코드 리뷰 에이전트가 있다고 하자. 이 에이전트에게 "웹검색" 도구를 노출할 필요가 있을까. 대부분의 코드 리뷰는 코드 자체만으로 충분히 수행된다. 웹검색이 노출돼 있으면 모델은 불확실할 때 "검색해보자"고 판단할 수 있고, 그 판단이 검색 결과를 근거로 답변을 바꾸는 쪽으로 흐를 수 있다. 이는 코드 리뷰의 품질을 떨어뜨린다. 웹검색 도구를 빼면 모델은 코드 자체에만 집중한다. 필요할 때는 상위 에이전트가 검색 결과를 대신 전달하는 구조로 설계하면 된다.

도구 수는 품질이 아니라 입도의 문제다. 거친 입도의 소수 도구가 정교한 입도의 다수 도구보다 낫다. 하나의 도구가 하나의 명확한 행동을 대표하도록 설계하면 모델의 판단 오류는 눈에 띄게 줄어든다.

4 출력 검증 게이트: 모델의 환각을 구조로 막는 법

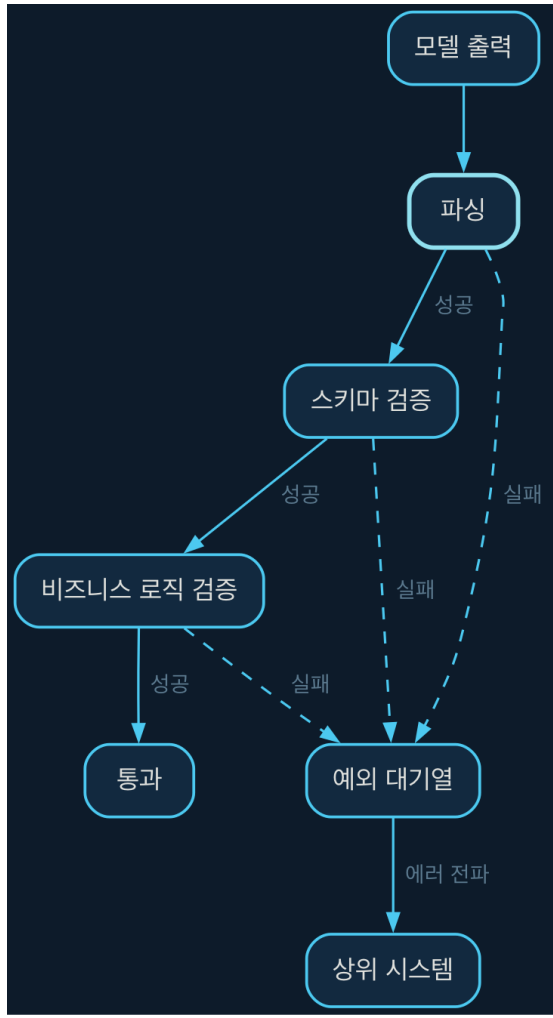

Figure 4.1: 출력 검증 게이트: 모델의 환각을 구조로 막는 법

4.1 검증 게이트는 모델 밖의 코드가 소유한다

LLM의 가장 큰 문제 중 하나는 환각, 즉 그럴듯하지만 실제와 다른 정보를 생성하는 능력이다. 에이전트에서 환각은 치명적이다. 모델이 도구를 사용해 얻은 정보를 잘못 해석하거나, 아예 존재하지 않는 정보를 만들어 출력에 끼워 넣을 수 있다. 이 문제를 풀려면 검증이라는 단계가 필요한데, 여기서 많은 팀이 실수를 한다.

검증 단계를 모델 자체에 맡기려는 시도다. "검증해봐라"라고 시스템 프롬프트에 지시하는 식이다. 하지만 이 방식은 문제가 명확하다. 모델이 스스로 환각을 감지할 수 있다면 애초에 환각을 만들어내지 않았을 것이다. 검증하라는 지시 자체가 논리적으로 모순된다. 모델의 "검증"도 진짜 검증은 아니다. 자신이 만든 내용을 자신이 판단하니 확증편향이 끼어든다.

검증 게이트는 반드시 모델 밖의 코드에서 수행해야 한다. 검증이 에이전트의 일부가 아니라 에이전트의 출력을 확인하는 별도의 층이라는 뜻이다. 이 층은 객관적이다. 모델이 만든 결과물의 정확성을 판단하는 것은 인간이 아니라 정해진 규칙이다.

4.2 출력 스키마를 강제하는 3단계 구조

검증 게이트의 구조를 단계별로 설계하면 다음과 같다.

첫 번째 단계는 **파싱**이다. 모델의 출력을 구조화된 형태로 바꾸는 과정이다. JSON이 예상되면 JSON 파싱을 시도하고, 실패하면 그 출력을 거부한다. 이 단계에서 대부분의 무효 출력이 걸러진다. 모델이 형식을 무너뜨리면 그 출력은 그 자리에서 거부된다.

이어지는 두 번째 단계는 **스키마 검증**이다. 파싱에 성공한 결과가 정해진 스키마에 맞는지 확인한다. 필수 필드가 있는지, 필드 타입이 맞는지, 값의 범위가 유효한지를 검사한다. 여기서 실패한 출력은 모델에게 다시 보내지

않는다. 곧바로 오류로 기록하고 다음 단계로 넘어가지 않는다.

마지막 세 번째 단계는 **비즈니스 로직 검증**이다. 스키마를 통과한 결과가 실제로 유효한지 확인한다. 예를 들어 데이터베이스에서 읽은 값의 합계가 특정 임계치를 넘으면 거부한다. 도메인 지식에 기반한 검증으로, 코드에 정의된 규칙을 따른다.

이 3단계 구조의 핵심은 각 단계가 독립적이라는 데 있다. 파싱 단계는 스키마를 몰라도 동작하고, 스키마 검증 단계는 비즈니스 로직을 몰라도 동작한다. 각 단계는 자신의 책임 범위만 검증한다. 이렇게 하면 검증 로직의 복잡성을 관리하면서도 각 단계의 실패 조건을 명확히 정의할 수 있다.

4.3 재시도 대신 라우팅

검증에 실패한 모델의 출력을 어떻게 처리해야 할까. 대부분의 팀은 재시도 메커니즘을 구현한다. 검증에 실패하면 같은 입력으로 모델을 다시 호출하는 것이다. 그러나 이 접근에는 근본적인 문제가 있다. 재시도는 같은 판단을 다른 시도로 반복하는 것이므로 같은 이유에서 또 실패할 가능성이 높다. 재시도가 무한히 반복되지 않도록 횟수 제한도 필요한데, 결국 그 제한에 도달하면 실패를 보고해야 한다.

보다 효과적인 방법은 재시도 대신 **라우팅**을 쓰는 것이다. 검증에 실패한 출력을 다시 모델에게 보내지 않고 다른 처리 경로로 넘긴다. 예외를 처리하는 채널을 따로 두는 셈이다.

라우팅의 구조는 다음과 같다. 검증에 실패한 출력은 "예외 대기열"로 이동한다. 이 대기열에서 모델은 실패 원인이나 해결 방법을 판단하지 않는다. 대신 또 다른 에이전트나 인간이 그 출력을 직접 확인하고 적절한 조치를 취한다. 이 접근은 에이전트의 역할을 단순화한다. 모델은 정상적인 경우에만 관여하고 비정상적인 경우는 다른 층이 처리한다.

재시도가 필요한 경우도 있다. 다만 재시도는 무조건적이지 않고 조건부여야 한다. "네트워크 타임아웃으로 인한 실패"는 재시도 대상이 될 수 있다. 그러나 "스키마 검증 실패"는 재시도 대상이 아니다. 스키마가 틀렸다는 것은 모델의 판단 자체가 잘못됐다는 뜻이고, 같은 판단을 반복해도 같은 잘못된 결과가 나온다. 이 구분이 재시도 설계의 핵심이다.

4.4 실패를 에러로 전파하는 설계

에이전트가 실패를 마주했을 때 가장 나쁜 행동은 실패를 숨기고 정상 출력을 흉내 내는 것이다. 모델이 출력을 만들지 못했거나 출력이 유효하지 않다는 것을 알면서도 "이런 결과가 나왔습니다"라고 답하는 경우다. 이 패턴은 시스템을 믿을 수 없게 만든다.

실패를 에러로 전파하는 것은 하나의 설계 원칙이다. 출력 검증 게이트에서 걸린 오류는 상위 시스템으로 보고된다. 에이전트를 호출한 클라이언트가 실패를 알아채고 적절히 대응할 수 있어야 한다. 이는 에이전트 자신의 판단이 아니라 시스템 전체의 복원력에 기대는 구조다.

코드 수준에서 보면 다음과 같은 구조가 되어야 한다.

```
try:
 output = agent.execute(input)
 validated = validator.validate(output)
 return validated
except ValidationError as e:
 raise AgentExecutionError(f"Output validation failed: {e}")
except ToolExecutionError as e:
 raise AgentExecutionError(f"Tool execution failed: {e}")
```

에이전트는 실패를 처리하려 들지 않는다. 감지하면 곧바로 상위로 전파할 뿐, 은폐하거나 완화하려 하지 않는다. 이 원칙이 지켜질 때 시스템은 모델의 자기평가가 아니라 에이전트의 출력에 대한 신뢰를 기반으로 하게 된다.

5 라우팅 규칙: 모델에게 선택지를 주지 않는 구조

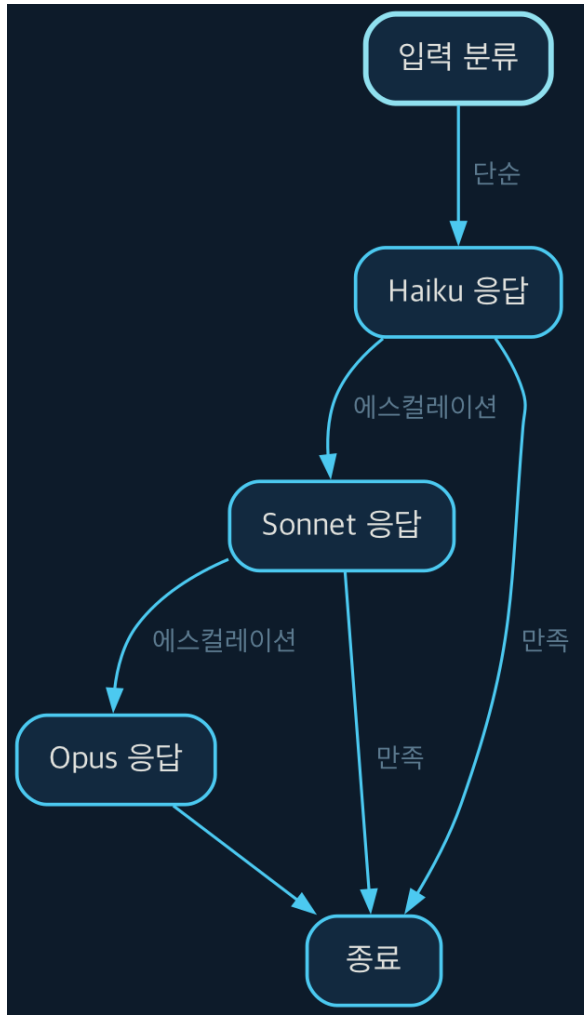

Figure 5.1: 라우팅 규칙: 모델에게 선택지를 주지 않는 구조

5.1 if-then 구조를 프롬프트가 아니라 코드에서 관리해야 하는 이유

에이전트가 복잡한 결정을 내려야 할 때 그 판단 기준을 시스템 프롬프트에 넣는 팀이 있다. "만약 사용자가 결제 관련 질문이면 결제 에이전트로 라우팅하고, 그렇지 않으면 일반 대화로 처리하라." 이런 판단 로직이 프롬프트에 있으면 모델이 그 판단을 대신 수행한다. 하지만 모델의 판단은 확률적이다. 같은 입력이라도 호출마다 다르게 판단할 수 있다. 판단 기준이 아무리 명확해도 모델이 그 기준을 따른다는 보장은 없다.

라우팅 판단은 프롬프트가 아니라 **코드**에서 수행해야 한다. 인간이 판단하면 if-then으로, 시스템이 판단하면 라우팅 함수의 인자로 정의한다. 모델은 라우팅 판단에 관여하지 않는다. 대신 라우팅 결과를 입력으로 받아 그에 맞는 행동만 한다.

이 구조는 세 가지 이점을 준다. 라우팅 결정의 정확도는 100%다. 코드의 조건 분기는 항상 정확한 결과로 평가되기 때문이다. 라우팅 결과도 테스트할 수 있다. 단위 테스트로 모든 분기를 검증하면 된다. 라우팅 기준이 바뀌어도 모델을 재교육할 필요가 없다. 코드만 고치면 바로 반영된다.

5.2 복수 모델을 Harness 차원에서 조합하는 프로덕션 패턴

복잡한 에이전트 시스템에서 모델 하나만 쓰면 비효율적일 때가 많다. 작업 성격에 따라 다른 모델을 써야 하는 경우도 있다. 단순 검색은 Haiku로, 복잡한 분석은 Sonnet으로, 중요한 판단은 Opus로 처리한다. 하지만 이 선택을 모델 자신에게 맡기면 결과를 예측할 수 없다.

하네스 차원에서 복수 모델을 조합하는 게 핵심이다. 모델 선택을

자동화하는 라우팅 규칙을 코드에 정의한다는 뜻이다. 입력의 복잡도, 위험도, 응답 속도 요구치 같은 특성을 보고 적합한 모델을 정한다.

구체적인 패턴은 이렇다. 입력을 분류하는 간단한 규칙 기반 모델을 앞단에 두고, 이 분류기가 입력이 간단한지 복잡한지 판단한다. 그 판단에 따라 복수의 후행 에이전트 중 하나를 고르는데, 각 후행 에이전트는 서로 다른 모델을 쓴다. 전체 시스템에서 보면 여러 모델이 함께 일하지만, 모델을 고르는 핵심 판단 자체는 모델 바깥에서 이루어진다.

이 구조의 실례를 보자. 고객 지원 에이전트에서 첫 번째 응답은 Haiku로 생성한다. Haiku의 응답이 사용자를 만족시키면 거기서 끝이다. 사용자가 추가 질문을 하면 그 질문의 복잡도에 따라 Sonnet 또는 Opus로 에스컬레이션한다. 각 단계에서 "에스컬레이션이 필요한가"라는 판단은 코드가 수행한다. 모델은 판단이 아니라 응답 생성에만 집중한다.

5.3 5축 프레임워크로 에이전트 품질을 지속적으로 측정하고 개선하는 법

에이전트 시스템의 품질은 한 번 설계하고 끝낼 수 있는 게 아니다. 환경이 변하고 사용자의 요구가 변하고 모델의 행동 특성도 업데이트마다 달라진다. 그래서 에이전트 품질을 계속 측정하고 개선하는 구조가 필요하다.

5축 프레임워크는 에이전트 시스템의 품질을 다섯 가지 차원에서 측정하는 방법론이다. 각 축은 서로 독립적이며, 하나라도 부족하면 전체 품질이 떨어진다.

첫 번째 축은 **정확성**이다. 출력이 입력의 의도를 정확히 반영하는가. 도구 호출 결과가 정확히 해석되는가. 환각이 발생하는가. 이 축은 주로 출력 검증 게이트가 관리한다.

두 번째 축은 **일관성**이다. 같은 입력이면 매번 같은 출력이 나오는가.

에이전트의 행동이 시간이 지나면서 변동하는 폭이 허용 범위 안에 있는가. 이 축은 시스템 프롬프트의 안정성이 좌우한다.

세 번째 축은 **적시성**이다. 출력이 제 시간 안에 나오는가. 장기 실행 작업에서 진행 상황 보고가 적절한가. 이 축은 에이전트의 실행 전략이 관리한다.

네 번째 축은 **적절성**이다. 출력이 상황에 맞는 수준인가. 너무 상세하거나 너무 간결하지 않은가. 이 축은 라우팅 규칙이 관리한다.

다섯 번째 축은 **복구력**이다. 실패 상황에서 시스템이 graceful하게 대응하는가. 실패가 상위로 전파되는가. 이 축은 에러 처리 구조가 관리한다.

5축마다 측정 지표를 정하고 정기적으로 수집한다. 수집한 데이터는 에이전트 시스템의 개선 방향을 정하는 근거가 된다. 한 축의 점수가 낮으면 그 축에 해당하는 구성 요소를 중점적으로 고친다.

5.4 하네스 개선의 실체: 반복 사이클

하네스 개선은 프로젝트가 아니라 반복되는 사이클이다. 에이전트를 배포하고 프로덕션에서 실패 케이스를 모은다. 그 케이스를 분석해 하네스의 어떤 부분이 부족했는지 파악하고 그 부분을 고친다. 고친 하네스를 다시 배포하고 모니터링한다. 이 사이클을 계속 돌리는 것만이 에이전트 품질을 높이는 방법이다.

핵심은 실패 케이스를 모으는 데 있다. 실패가 나도 로그에만 남기고 방치하면 개선은 일어나지 않는다. 실패 패턴을 분석해서 그것이 하네스의 어느 부분과 연결되는지 추적할 수 있게 해야 한다. 그러지 않으면 시스템은 실패에서 배울 기회를 영영 잃는다.

모델을 바꾸는 일은 에이전트를 다시 훈련시키는 게 아니라 엔진을

교체하는 것과 같다. 엔진이 강해지면 속도는 빨라지지만, 하네스가 뒷받침하지 않으면 방향을 잘못 잡을 가능성이 크다. 하네스 설계에 쏟은 시간과 노력이 에이전트의 진짜 품질을 결정한다.