AI API 계약 엔지니어링

LLM을 시스템 구성 요소로 쓸 때 발생하는 계약 설계, 장애 복원, 비용 관리, 버전 관리 패턴

ThakiCloud (Hyojung Han)

AI API 계약 엔지니어링

LLM을 시스템 구성 요소로 쓸 때 발생하는 계약 설계, 장애 복원,
비용 관리, 버전 관리 패턴

ThakiCloud (Hyojung Han)

Contents

1 계약이라는 사고법

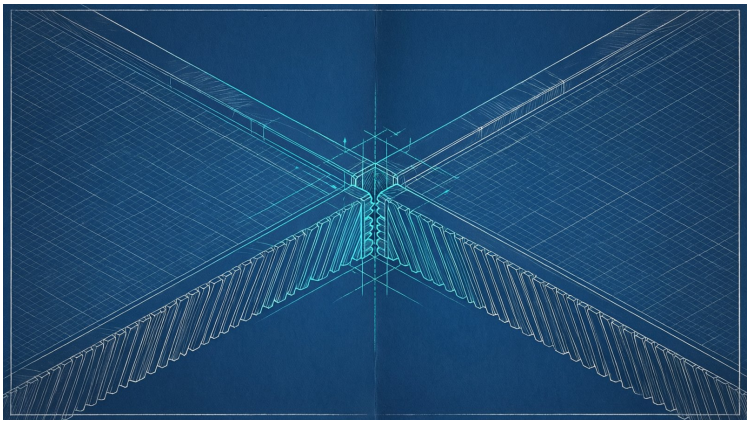

Figure 1.1: 계약이라는 사고법

1.1 서론: 왜 계약인가

AI 모델을 소프트웨어 시스템에 통합할 때 엔지니어들이 가장 흔히 저지르는 실수는 모델을 일반적인 함수처럼 취급하는 것이다. 함수라면 입력과 출력의 경계가 뚜렷하다. 하지만 LLM은 네트워크 서비스에 가깝다. 응답 시간이 일정하지 않고 원하는 형식을 따르지 않을 때도 있으며, 비용은 토큰 단위로 쌓인다. 이런 특성을 이해하지 못하면 시스템은 예기치 않은 장애와 비용 폭증을 겪게 된다.

이 책에서 말하는 계약이란 AI 모델과 소프트웨어 사이의 명확한 약속을 뜻한다. 무엇을 입력으로 받고 어떤 형태로 출력하며 실패할 때 어떻게 행동할지를 명시적으로 정해두는 것이다. 계약이 없으면 시스템은 모델의 변동성에 그대로 노출된다.

1.2 1. LLM은 네트워크 서비스라는 사실

전통적인 소프트웨어 컴포넌트는 동일한 입력에 항상 동일한 출력을 반환한다. 이 결정론적 특성 덕분에 테스트가 가능하고 버그도 예측할 수 있다. 반면 AI 모델은 본질적으로 확률적 시스템이다. 동일한 프롬프트라도 모델의 상태, 온도 파라미터, 서빙 인프라의 부하 상황에 따라 출력이 달라질 수 있다.

더 중요한 사실은 LLM이 외부 네트워크 서비스라는 점이다. API를 호출하는 동안 네트워크 지연, 프로바이더 측 장애, 속도 제한, 토큰 할당량 소진 등 다양한 변수가 개입한다. 이런 현실을 인식하지 못한 시스템 설계는 프로덕션 환경에서 매번 예상과 다른 결과를 만들어낸다.

예를 하나 들어보자. 어떤 시스템이 LLM의 응답을 파싱해서 데이터베이스에 저장한다고 하자. 모델이 간헐적으로 추가 설명을 출력하면 JSON 파싱이 실패하고 전체 파이프라인이 중단된다. 계약이 있었다면 입력 단계에서 출력 형식을 명시하고, 형식에 맞지 않는 응답은 폴백 경로로 처리했을 것이다.

1.3 2. 계약 설계의 세 가지 축

AI API 계약은 세 가지 축으로 구성된다. 입력 계약, 출력 계약, 실패 모드 계약이 그것이다.

입력 계약은 모델에 무엇을 보내고 어떤 전제 조건이 만족되어야 하는지를 정의한다. 프롬프트의 구조, 시스템 메시지, 입력 길이의 상한, 필수 필드 여부 등이 여기에 해당한다. 입력 계약을 지키지 않으면 모델은 예상치 못한 방향으로 응답할 수 있다.

출력 계약은 모델의 출력이 어떤 형태와 구조를 가져야 하는지를 명시한다. JSON 스키마, 특정 필드의 존재 여부, 출력 길이의 범위 등이 포함된다.

출력 계약이 없으면 파싱 로직이 번번이 실패하거나, 나온 그대로 처리했다가 데이터 무결성이 깨지는 상황을 맞는다.

실패 모드 계약은 모델 호출이 실패할 때 시스템이 어떻게 행동해야 하는지를 정의한다. 재시도 정책, 폴백 모델, 캐시된 결과 활용, 사용자에게 표시할 오류 메시지까지 포함한다. 실패 모드를 설계하지 않으면 작은 장애가 전체 시스템 중단으로 번진다.

이 세 축은 독립적이지 않다. 입력 계약을 바꾸면 출력에도 영향을 미치고, 실패 모드를 어떻게 설계하느냐에 따라 입력 검증의 엄격함도 달라진다.

1.4 3. 명시적 계약과 암묵적 계약

소프트웨어 시스템에는 두 가지 종류의 계약이 존재한다. 명시적 계약과 암묵적 계약이다.

명시적 계약은 코드와 문서로 기록된 약속이다. API 스키마, Validation 규칙, 에러 코드, Rate Limit 정책이 이에 해당한다. 팀원 누구나 문서를 확인하면 시스템의 행위를 예측할 수 있다.

암묵적 계약은 코드에 명시되지 않았지만 구성 요소들이 당연하게 전제하는 약속이다. “이 함수는 null을 반환하지 않는다”, “이 API는 100밀리초 내에 응답한다”, “이 모델은 항상 유효한 JSON을 반환한다” 등의 전제가 여기에 속한다.

암묵적 계약은 초기 개발 단계에서는 편의처럼 보인다. 문서를 많이 쓰지 않아도 되고 유연하게 동작하는 것처럼 느껴진다. 그러나 시스템이 복잡해지고 팀이 커지면 이 전제들이 서로 충돌하기 시작한다. 한 팀은 “이 API는 항상 200을 반환한다”고 가정하지만, 실제로는 429가 돌아오는 경우를 다른 팀이 미처 발견하지 못하는 식이다.

LLM을 사용할 때는 특히 암묵적 계약의 함정에 빠지기 쉽다. 모델이 “보통”

유효한 JSON을 반환한다는 전제는 위험하다. 100번 중 1번의 실패가 치명적인 시스템에서는 계약을 명시적으로 정의하고 검증해야 한다.

1.5 4. 계약을 계약답게 만드는 것

계약이 문서에만 존재하면 의미가 없다. 실제로 검증되고 강제되어야 한다. 입력 계약이라면 모델 호출 전에 스키마 검증을 수행하고 계약에 맞지 않으면 사전에 거부해야 한다. 출력 계약이라면 파서가 예외를 삼키지 않고 명시적으로 실패를 보고해야 한다. 실패 모드 계약이라면 재시도 로직과 폴백 경로가 실제로 작동하는지 테스트로 보장해야 한다.

결국 계약 설계란 실패 가능성을 인정하는 일이다. 완벽한 시스템을 꿈꾸지 말고, 모델이 언제 어떻게 실패할 수 있는지를 미리 정의하고, 그에 대한 대응책을 마련하는 것이 안전한 운용의 출발점이다.

1.6 정리

이 장에서는 계약이라는 사고법의 기초를 다루었다. LLM을 네트워크 서비스로 인식하고, 입력과 출력, 실패 모드의 세 축으로 계약을 설계하며, 암묵적 전제보다는 명시적 규칙을 선호해야 한다는 점을 확인했다. 다음 장에서는 이 세 축 중 입력 계약, 특히 프롬프트 계약의 설계 방법을 구체적으로 살펴본다.

2 입력 계약의 설계

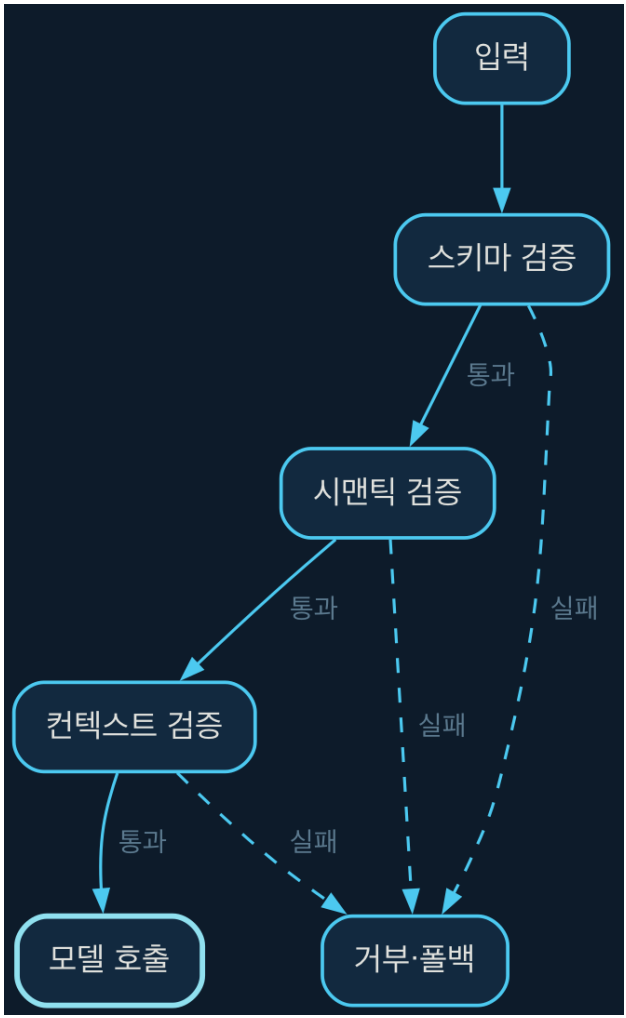

Figure 2.1: 입력 계약의 설계

2.1 서론: 입력이 곧 계약을 만든다

LLM의 출력 품질은 입력의 품질에 크게 좌우된다. 그런데 많은 시스템이 입력 관리를 뒷전으로 미루고 프롬프트 엔지니어링에만 매달린다. 프롬프트 자체보다 중요한 것은 입력 전체의 구조와 유효성이다. 입력 계약이 명확해야 모델의 행동을 예측할 수 있고, 문제가 생겼을 때 원인을 파악할 수 있다.

2.2 1. 프롬프트 계약의 구조화

프롬프트는 자유 형식 텍스트처럼 보이지만 실제로는 구조화된 메시지의 조합이다. 시스템 메시지, 사용자 입력, 컨텍스트 정보가 각각 다른 역할을 맡으며, 이들을 조합하는 방식이 모델의 응답을 좌우한다.

시스템 메시지는 모델의 행동 규범을 정의한다. 역할 지시, 출력 형식, 절대적으로 지켜야 할 사항이 여기에 담긴다. 시스템 메시지가 명확하면 모델은 역할을 일관되게 수행한다. 반면 모호하면 모델은 매번 다른 방식으로 응답한다.

사용자 입력은 실제로 풀어야 할 문제를 전달한다. 여기서 관건은 입도다. 너무 추상적이면 모델은 범론으로 빠지고, 너무 구체적이면 모델의 창의성이 제한된다. 좋은 사용자 입력은 목표와 제약 조건을 명확히 하면서도 모델이 스스로 판단할 여지를 남긴다.

컨텍스트 정보는 모델이 과거 대화를 기억하지 못한다는 한계를 보완한다. 대화 히스토리, 관련 문서, 외부 검색 결과 등이 컨텍스트에 해당한다. 컨텍스트가 충분해야 모델이 일관되게 대화를 이어간다. 그러나 컨텍스트가 너무 길면 모델은 중간 정보를 잊어버리거나 비용이 불필요하게 늘어난다.

2.3 2. 입력 검증 게이트

입력 계약의 핵심은 모델을 호출하기 전에 입력이 유효한지 검증하는 데 있다. 이 단계를 게이트라고 부른다. 게이트를 통과한 입력만 모델로 전달되고, 통과하지 못한 입력은 즉시 거부되거나 폴백 처리된다.

입력 검증은 여러 층으로 구성하는 편이 좋다. 첫 번째 층은 스키마 검증이다. 필수 필드의 존재, 데이터 타입, 문자열 길이의 범위를 확인한다. 이 검증은 비즈니스 로직과 무관하게 항상 적용된다.

두 번째 층은 시맨틱 검증이다. 필드 값이 논리적으로 타당한지를 확인한다. 예를 들어 날짜 필드가 있다면 미래의 날짜가 아닌지, 수량이 있다면 양수인지를 확인한다. 이런 검증은 비즈니스 규칙에 따라 달라진다.

세 번째 층은 컨텍스트 검증이다. 현재 시스템의 상태나 다른 외부 조건을 고려한 검증이다. 예를 들어 사용자의 요금제가 한도에 도달했는지 확인하거나 모델의 현재 부하 상태를 확인하는 일이 여기에 해당한다.

검증 게이트를 설계할 때는 검증에 실패했을 때 어떻게 행동할지를 미리 정해두어야 한다. 즉시 오류를 반환하든, 기본값으로 대체하든, 사용자에게 추가 입력을 요청하든, 이 판단은 시스템 설계자의 몫이다.

2.4 3. 컨텍스트 윈도우 관리 전략

LLM은 무한한 컨텍스트를 처리할 수 없다. 모델마다 고유한 최대 토큰 수가 있으며, 이를 윈도우라고 부른다. 이 윈도우를 초과하면 모델은 오래된 정보를 잊어버리거나 입력 전체를 처리하지 못한다.

컨텍스트 윈도우 관리는 입력 계약 설계의 핵심 요소다. 윈도우를 넘지 않도록 입력의 길이를 관리하고, 넘칠 경우에는 적절한 전략을 선택해야 한다.

첫 번째 전략은 자르기다. 입력의 앞부분이나 말미를 정해진 크기로 자른다. 간단하지만 정보를 잃을 수 있다. 특히 긴 문서라면 핵심 내용이 잘려 나갈 수 있다.

두 번째 전략은 요약이다. 긴 입력을 모델 스스로 요약하게 하고, 요약본을 컨텍스트에 포함시킨다. 정보 손실은 줄어들지만 추가 비용이 들고, 요약 과정에서 세부사항이 사라질 수 있다.

세 번째 전략은 분할이다. 긴 입력을 여러 부분으로 나눠 각각 독립적으로 처리한 후 결과를 합친다. MapReduce 패턴과 비슷하다. 다만 부분 간에 의존성이 있으면 효과가 떨어진다.

네 번째 전략은 정보 선택이다. 임베딩을 활용해 입력 중 현재 작업에 가장 관련된 부분만 선택적으로 포함시킨다. RAG 패턴이 이에 해당한다. 그러나 관련성 판단 자체가 틀릴 수 있다.

어떤 전략을 선택하든 윈도우 크기와 현재 입력 크기를 항상 감시해야 한다. 초과한 뒤에야 아는 것은 이미 늦다. 미리 계산하고 계획해야 비용과 성능을 예측할 수 있다.

2.5 4. 입력 계약을 문서화하고 테스트하기

입력 계약의 마지막 단계는 문서화와 테스트다. 문서에는 입력의 구조, 각 필드의 의미, 유효성 규칙, 예외 상황을 명확히 기록해야 한다. 테스트에서는 정상 입력뿐 아니라 경계값과 예외 상황을 검증해야 한다.

문서는 다른 팀원이 보더라도 입력을 올바르게 구성할 수 있도록 써야 한다. 입력을 구성하는 모든 필드와 제약 조건을 명시하고 예시를 곁들이면 이해하기 쉬워진다.

테스트에서는 주로 세 가지 케이스를 다루어야 한다. 정상 케이스, 경계값 케이스, 예외 케이스다. 정상 케이스는 계약에 맞는 입력이 제대로

처리되는지 확인한다. 경계값 케이스는 길이나 수량이 허용 범위의 경계에 있는 입력을 테스트한다. 예외 케이스는 계약에 어긋난 입력이 적절히 거부되거나 처리되는지를 확인한다.

2.6 정리

이 장에서는 입력 계약을 설계하는 방법을 다루었다. 프롬프트를 시스템 메시지, 사용자 입력, 컨텍스트로 구조화하는 법, 세 층의 검증 게이트를 두는 법, 컨텍스트 윈도우를 관리하는 네 가지 전략을 짚었다. 입력 계약을 명확히 하면 모델의 행동을 예측할 수 있고 문제를 미리 막을 수 있다. 다음 장에서는 출력 계약, 즉 모델의 출력을 어떻게 구조화하고 검증할 것인지를 다룬다.

3 출력 계약의 설계

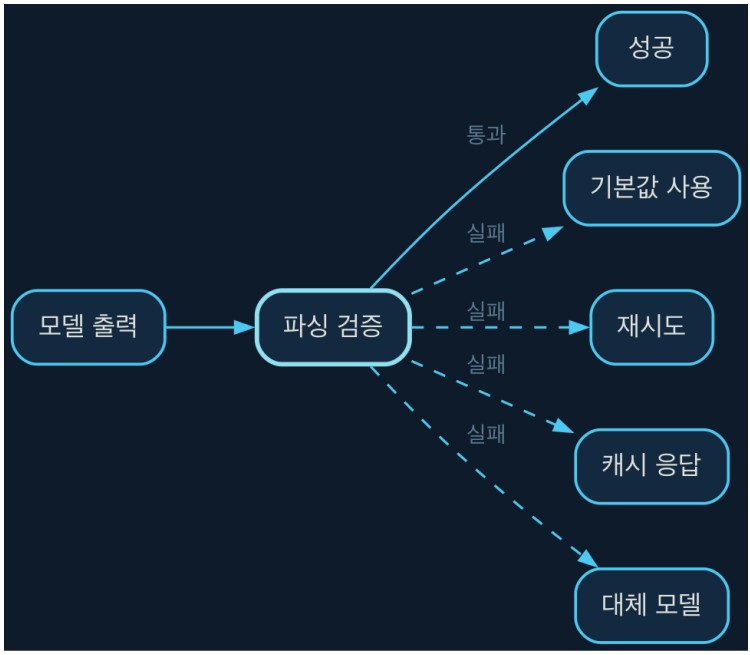

Figure 3.1: 출력 계약의 설계

3.1 서론: 출력은 시스템의 입구다

LLM의 출력은 다음 컴포넌트로 들어가는 입구다. 파싱 로직, 데이터베이스, 다른 API, 사용자에게 보이는 화면까지 이 출력을 받아 움직인다. 출력이 어떻게 구성되느냐에 따라 전체 시스템의 동작이 결정된다. 문제는 모델의 출력이 본질적으로 확정적이지 않다는 데 있다. 같은 입력을 줘도 다양한 형태의 출력이 나올 수 있고, 이 불확실성을 어떻게 다루느냐가 출력 계약 설계의 핵심이다.

3.2 1. 출력 구조화의 세 가지 전략

모델의 출력을 구조화하는 방법은 강제 구조화, 가이드 구조화, 사후 구조화 세 가지로 나뉜다. 세 전략은 각각 다른 trade-off를 만든다.

강제 구조화는 모델이 특정 JSON 스키마 형태로만 출력하도록 강제하는 방법이다. 시스템 메시지에 출력 형식을 명시하고, 필요하면 구조를 요구하는 프롬프트를 추가한다. 장점은 파싱이 쉽다는 것이다. 다만 모델이 구조를 따르지 못하면 그대로 파싱 실패로 이어진다.

가이드 구조화는 모델이 자유롭게 출력하게 두고, 그 뒤에 구조화된 형태로 변환하는 로직을 붙이는 방법이다. LLM이 스스로 출력을 구조화하게 하거나 별도의 파싱 모델을 쓸 수도 있다. 강제 구조화보다 유연한 대신, 변환 과정에서 정보 손실이나 오류가 생길 수 있다.

사후 구조화는 모델의 자유 출력을 정규 표현식이나 파싱 로직으로 나중에 해석하는 방법이다. 가장 유연하지만 가장 취약하다. 모델의 출력이 조금만 달라져도 파서가 바로 실패할 수 있다.

실무에서는 강제 구조화와 가이드 구조화를 조합하는 편이 효과적이다. 모델에게 가능한 한 구조를 따르도록 요구하되, 따르지 않을 경우를 대비해 폴백 로직을 마련해두는 것이다.

3.3 2. 파싱 오류 처리와 폴백 설계

모델이 계약에 맞는 출력을 반환하지 않을 때 무엇을 할지 정해두는 작업이 폴백 설계다. 폴백은 단순히 오류를 로깅하고 끝내는 게 아니라, 시스템이 계속 동작하도록 대안을 제공하는 일이다.

폴백의 첫 번째 선택지는 기본값을 사용하는 것이다. 예를 들어 구조화된 응답에서 특정 필드가 빠진 경우 해당 필드에 기본값을 할당한다. 그러나 기본값이 시스템의 다른 부분에서 예상치 못한 문제를 일으킬 수 있다.

두 번째 선택지는 재시도다. 모델을 다시 호출해 새로운 응답을 받는다. 다만 재시도도 실패할 수 있으니 최대 시도 횟수를 정해두어야 한다. 재시도 사이에 지수 백오프를 두면 일시적인 장애를 효과적으로 넘길 수 있다.

세 번째 선택지는 캐시다. 이전에 모델이 반환한 적당한 응답을 저장해두었다가 실패 시 그것을 쓴다. 다만 오래된 캐시 응답이 지금 작업에 맞지 않을 수 있다.

네 번째 선택지는 대체 모델이나 방식이다. 주 모델이 실패하면 경량 모델이나 규칙 기반 방식으로 바꾼다. 품질은 낮아지지만 최소한의 기능은 유지된다.

어떤 폴백 전략을 고르든 폴백 발생 횟수는 모니터링해야 한다. 폴백이 자주 일어난다면 모델이나 프롬프트를 손봐야 한다는 신호다. 폴백을 처리만 하고 관찰하지 않으면 시스템의 실제 품질을 알 수 없다.

3.4 3. 부분 성공의 처리 패턴

모델의 출력이 부분적으로만 유효한 경우가 있다. 예를 들어 여러 항목을 생성해야 하는데 그중 일부만 성공하고 나머지는 실패하는 경우다. 이런 상황을 어떻게 다룰지가 부분 성공 패턴이다.

완전 성공만 허가하는 방식은 가장 보수적이다. 하나라도 실패하면 전체를 실패로 처리하고, 작업을 중지하거나 재시도한다. 데이터 무결성이 중요한 시스템에서 효과적이지만, 불필요한 재시도가 발생할 수 있다.

부분 성공을 허가하는 방식은 유효한 결과만 활용하고, 실패한 항목은 별도로 기록하거나 사용자에게 보고한다. 유연하지만, 시스템이 불완전한 데이터로 동작할 수 있다는 점에 주의해야 한다.

최선 노력 방식은 유효한 결과를 최대한 활용하면서 실패가 나도 작업을 계속 진행한다. 속도와 완급 조절이 중요한 시스템에서 효과적이지만,

디버깅은 어려워질 수 있다.

어떤 패턴을 고르든 중요한 것은 실패와 성공의 기준을 명확히 정의하는 일이다. 모델의 출력이 어떤 조건을 만족해야 유효하다고 볼지 문서화하고, 조건에 맞지 않는 경우를 어떻게 처리할지도 함께 정해야 한다.

3.5 4. 출력 검증과 모니터링

출력 계약의 마지막 단계는 검증과 모니터링이다. 모델이 반환한 출력이 계약에 맞는지 자동으로 검증하고, 계약 위반이 발생하면 즉시 탐지해야 한다.

검증은 파싱 단계에서 수행하는 것이 가장 효과적이다. JSON 스키마에 맞는지 확인하고, 필수 필드가 존재하는지, 값의 범위가 적절한지를 검사한다. 검증에 실패하면 폴백 로직으로 즉시 전환한다.

모니터링에서는 주로 계약 준수율과 폴백 발생 빈도를 추적한다. 계약 준수율이 낮으면 모델이나 프롬프트를 조정해야 하고, 폴백 발생이 특정 시간대나 입력 패턴과 관련되어 있는지를 분석하면 예상치 못한 문제를 미리 파악할 수 있다.

출력 계약은 한 번 정의하고 끝나는 게 아니다. 시스템이 운영되는 동안 계속 모니터링하면서, 문제가 발견되면 계약을 수정하거나 모델을 조정해야 한다. 계약과 현실 사이의 갭을 줄여나가는 것이 안정적인 시스템 운영의 열쇠다.

3.6 정리

이 장에서는 출력 계약을 설계하는 방법을 다뤘다. 세 가지 구조화 전략과 폴백 설계, 부분 성공 처리 패턴, 검증과 모니터링까지 짚었다. 출력 계약을 명확히 하면 파싱 오류가 줄고 시스템의 예측 가능성이 높아진다. 다음 장에서는 장애가 발생했을 때 시스템이 어떻게 대응해야 하는지 다룬다.

4 장애 복원 설계

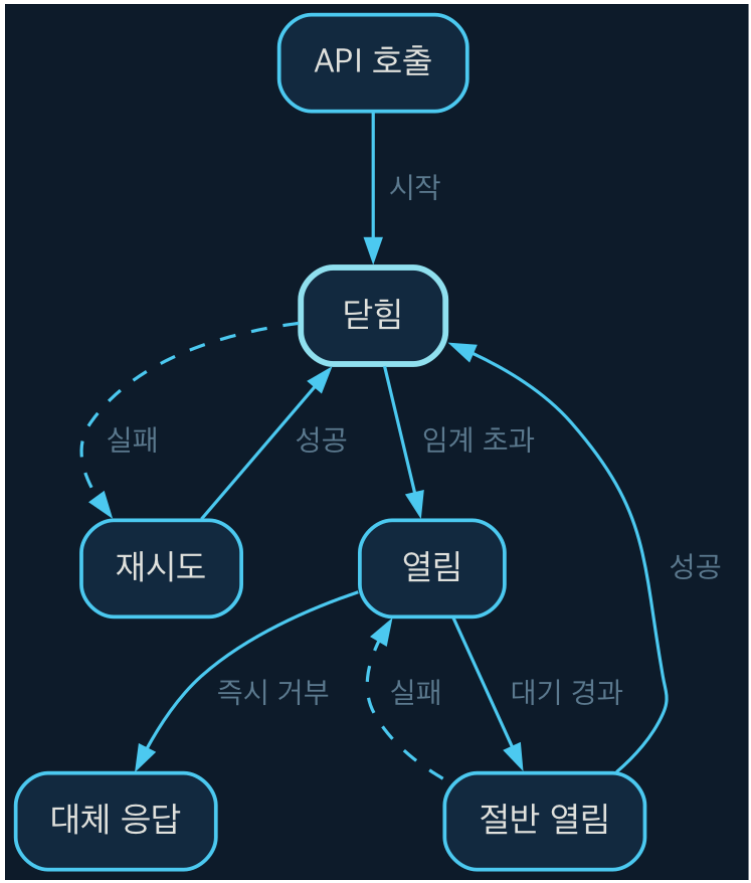

Figure 4.1: 장애 복원 설계

4.1 서론: 장애는 반드시 발생한다

아무리 견고하게 설계해도 장애는 발생한다. 네트워크 단절, API 서버의 일시적 과부하, 속도 제한 초과, 모델 서버의 예기치 않은 종료 등 다양한 원인으로 API 호출이 실패할 수 있다. 장애가 발생했을 때 시스템이 어떻게

행동하느냐가 관건이다. 대비하지 않으면 단일 장애가 전체 시스템의 중단으로 번질 수 있다.

4.2 1. 재시도 전략과 지수 백오프

재시도는 가장 기본적인 장애 복원 수단이다. 실패한 API 호출을 다시 시도해 성공 가능성을 높이는 방식이다. 다만 아무 조건 없이 재시도하면 오히려 문제가 악화될 수 있다. 일시적인 장애 상황에서 무분별한 재시도는 서버에 추가 부하를 주고 속도 제한에 걸릴 위험도 키운다.

가장 기본적인 재시도 전략은 고정 간격 재시도다. 실패한 뒤 정해진 시간만큼 기다렸다가 다시 시도하는 방식이다. 구현은 간단하지만 서버가 이미 과부하 상태라면 재시도가 오히려 상황을 악화시킬 수 있다.

더 나은 전략은 지수 백오프다. 재시도할 때마다 대기 시간을 두 배로 늘리는 방식으로, 첫 번째 재시도는 1초 후, 두 번째는 2초 후, 세 번째는 4초 후로 이어진다. 서버가 스스로 회복할 시간을 벌어주는 동시에 불필요한 재시도도 줄여준다.

재시도 전략을 설계할 때 고려해야 할 파라미터로는 최대 재시도 횟수, 초기 대기 시간, 최대 대기 시간, 재시도할 예외 유형이 있다. 최대 재시도 횟수를 정하지 않으면 무한 재시도로 이어질 수 있고, 초기 대기 시간이 너무 길면 불필요한 지연이 생기며 너무 짧으면 서버에 부하가 집중된다.

재시도할 예외 유형은 신중히 골라야 한다. 네트워크 단절이나 500번대 에러처럼 일시적인 장애라면 재시도가 유효하지만, 400번대 에러처럼 요청 자체가 잘못됐다면 재시도해도 소용없다.

4.3 2. 회로 차단기 패턴

재시도만으로는 충분하지 않은 경우도 있다. 서버가 완전히 다운되었거나 속도 제한에 걸려 있을 때는 재시도가 오히려 추가 부하만 만든다. 이럴 때는 시스템이 장애를 스스로 인정하고 호출을 자동으로 거부해야 하는데, 이것이 회로 차단기의 개념이다.

회로 차단기는 닫힘, 열림, 절반 열림이라는 세 가지 상태를 오간다. 닫힘 상태는 정상 동작 상태로, API 호출이 실패할 때마다 내부 카운터를 늘린다. 열림 상태에서는 모든 API 호출을 즉시 거부하고, 절반 열림 상태에서는 일부 호출만 시도해 서버가 회복됐는지 확인한다.

닫힘 상태에서 실패 횟수가 임계치를 넘으면 회로 차단기가 열리고, 설정된 시간 동안 모든 호출을 거부한다. 이 시간이 지나면 절반 열림 상태로 넘어가 일부 호출만 허용하는데, 호출이 계속 성공하면 닫힘 상태로 돌아가고 실패하면 다시 열린다.

회로 차단기의 핵심은 빨리 실패하는 데 있다. 서버가 이미 장애 상태임을 감지하면 불필요한 재시도 없이 즉시 실패를 반환한다. 그러면 시스템 전체가 장애 서버의 응답을 기다리며 시간을 낭비하지 않는다.

회로 차단기를 구현할 때는 상태 전이 로직뿐 아니라 상태를 어디에 저장할지도 중요하다. 분산 시스템에서는 여러 인스턴스가 동시에 회로 차단기를 운용하므로 상태를 공유하는 메커니즘이 필요하며, 데이터베이스나 Redis 같은 외부 저장소를 활용하는 방법이 있다.

4.4 3. 장애 격리와 Graceful Degradation

시스템의 한 부분에서 장애가 나면 그 영향이 다른 부분으로 번지지 않도록 막아야 하는데, 이를 장애 격리라고 부른다. 격리가 잘 되어 있으면 장애의 영향 범위가 해당 부분으로 제한되고 나머지 시스템은 계속 동작할 수 있다.

장애 격리의 기본 원칙은 필요한 곳에만 격리를 적용하는 것이다. 모든 컴포넌트를 격리하면 시스템만 불필요하게 복잡해진다. 핵심 기능에 직접 영향을 주지 않는 부분부터 격리 전략을 적용하는 편이 효율적이다.

Graceful Degradation은 시스템이 완전하지 않은 상태에서도 최소한의 기능을 유지하는 능력이다. 예를 들어 AI 기반 추천 시스템에서 추천 생성 기능에 장애가 나면 인기 상품 목록이라도 대신 보여준다. 완전한 기능을 제공할 수 없어도 부분적인 기능으로 사용자 경험을 지키는 것이다.

Graceful Degradation을 구현하려면 시스템의 기능을 계층화해야 한다. 가장 중요한 기능부터 순서를 매기고 상위 기능에 장애가 나면 다음 단계의 기능으로 폴백하는 식이다. 이렇게 하면 중요한 기능부터 먼저 작동하도록 보장할 수 있다.

장애 격리와 Graceful Degradation을 설계할 때는 사용자 관점에서 무엇이 중요한지부터 정의해야 한다. 장애 상황에서도 반드시 제공해야 할 기능은 무엇인지, 없어도 되는 기능은 무엇인지를 명확히 가려야 한다. 이를 바탕으로 격리와 폴백의 우선순위를 정한다.

4.5 4. 장애 복원 계획적 실행

장애 복원 전략을 설계만 하고 테스트하지 않으면 실제 장애가 발생했을 때 제대로 동작하지 않는다. 재시도, 회로 차단기, 격리, Graceful Degradation은 모두 자동화된 테스트로 검증해야 한다.

chaos engineering은 의도적으로 장애를 일으켜 시스템의 복원 능력을 테스트하는 접근법이다. API 서버를 일부러 다운시키거나 네트워크 지연을 발생시키거나 속도 제한을 초과하게 만들고, 그 상황에서 시스템이 어떻게 행동하는지 관찰한다.

장애 복원 계획의 목표는 시스템이 장애 상황에서도 사용자에게 예상 가능한 방식으로 응답하게 만드는 데 있다. 장애 자체를 완전히 막을 수는

없지만, 그 영향은 예측 가능하게 만들 수 있다.

4.6 정리

이 장에서는 재시도와 지수 백오프, 회로 차단기, 장애 격리와 Graceful Degradation까지 장애 복원 설계 전반을 다루었다. 장애를 대비하는 데는 비용과 복잡성이 따르지만, 프로덕션 환경에서는 반드시 필요한 투자다. 다음 장에서는 비용 관리와 버전 관리, 즉 시스템을 오래도록 안정적으로 운영하는 방법을 다룬다.

5 비용과 버전의 관리

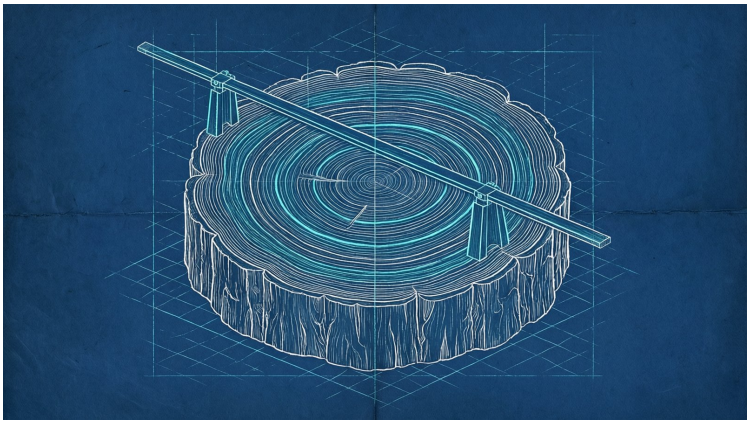

Figure 5.1: 비용과 버전의 관리

5.1 서론: 숨겨진 비용과 만료하는 계약

LLM API를 시스템에 통합할 때 간과하기 쉬운 위험이 두 가지 있다. 하나는 비용이 모르는 사이에 늘어나는 것이고, 다른 하나는 모델 버전이 예기치 않게 바뀌는 것이다. 둘 다 시스템의 동작을 예측하기 어렵게 만들고 예상치 못한 청구서로 이어질 수 있다. 이 장에서는 비용과 버전을 체계적으로 관리하는 방법을 다룬다.

5.2 1. 토큰 사용량 예측과 예산 관리

LLM API의 비용은 토큰 수를 기준으로 계산된다. 입력 토큰과 출력 토큰 각각에 비용이 부과되며, 모델마다 토큰 단가가 다르다. 비용을 예측하려면 먼저 토큰 사용량을 추산할 수 있어야 한다.

토큰 사용량을 추산하는 가장 간단한 방법은 입력과 출력의 문자 수를 세는 것이다. 한글은 영어보다 토큰당 포함되는 글자 수가 적어서, 한글 1글자를 약 2토큰으로 잡으면 대략적으로 추산할 수 있다. 다만 정확한 계산을 원한다면 실제 API 응답에 찍히는 토큰 사용량을 직접 확인하는 편이 낫다.

대부분의 LLM API 제공자는 응답 헤더나 usage 필드에 토큰 사용량을 돌려준다. 이 값을 기록해 쌓아두면 실제 사용량을 파악할 수 있다. 그러면 월간이나 주간 단위로 비용을 예측할 수 있다.

비용 예산관리의 핵심은 예산을 초과하기 전에 감지하고 대응하는 것이다. 예산 한도를 설정하고 사용량이 한도에 근접하면 알림을 발생시킨다. 필요하면 사용량을 줄이는 조치를 즉시 적용할 수 있다. 청구서를 받고 나서 놀라기보다는 미리 관리하는 편이 낫다.

비정상적인 사용량 패턴을 탐지하는 것도 비용 관리에서 중요하다. 평소와 다른 대규모 토큰 소비가 발생하면 경고해야 한다. 버그로 인한 무한 루프나 서비스 공격의 신호일 수 있기 때문이다.

5.3 2. 모델 버전 관리 전략

LLM 제공자는 간헐적으로 모델을 업데이트한다. 새 버전은 종종 다른 동작을 보이고, 이전 버전과 호환되지 않는 출력을 낼 수 있다. 버전 관리를 소홀히 하면 시스템이 모르는 사이에 변하고, 예기치 못한 버그가 발생할 수 있다.

버전 관리의 기본은 사용하는 모델의 버전을 명시적으로 지정하는 것이다. “최신” 버전이 아니라 정확한 버전 번호를 못박는다. 그러면 매번 같은 모델을 쓰게 되고, 예기치 않은 변화에 노출되지 않는다.

모델을 업데이트할 때는 점진적인 접근이 필요하다. 모든 트래픽을 한 번에 새 버전으로 옮기지 않고 일부만 신버전으로 라우팅한다. 이를 canary 배포라고 부른다. 새 버전에서 문제가 발견되면 즉시 구버전으로 되돌릴

수 있다.

버전 업데이트는 자동으로 이루어지지 않게 하는 편이 좋다. 새 버전이 공개됐다고 해서 그것이 이미 시스템에 적용됐다는 뜻은 아니다. 업데이트 여부는 개발자와 운영자가 판단해서 결정해야 한다.

버전 변경 시에는 문서화가 중요하다. 언제 어떤 버전으로 바꿨고 변경 전후에 무엇이 달라졌는지를 기록한다. 그러면 문제가 생겼을 때 원인을 분석하기 쉽고, 필요하면 이전 버전으로 매끄럽게 되돌릴 수 있다.

5.4 3. Provider 전환의 계약 설계

LLM API를 단일 프로바이더에 의존하면 그 프로바이더의 장애나 정책 변경에 취약해진다. 그래서 여러 프로바이더를 함께 쓰는 전략이 필요하다. 이를 multi-provider 전략이라고 부른다.

Provider 전환을 설계하려면 모든 프로바이더에 공통되는 인터페이스를 정의해야 한다. 그러면 프로바이더 하나가 실패해도 다른 프로바이더로 쉽게 옮겨갈 수 있다. 공통 인터페이스는 요청 형식, 응답 형식, 에러 코드까지 포함한다.

Provider 전환 전략은 여러 가지다. 가장 간단한 것은 failover 방식으로 기본 프로바이더가 실패하면 백업 프로바이더로 자동 전환한다. 더 복잡한 방식으로는 load balancing이 있는데 여러 프로바이더에 트래픽을 분산시켜 어느 한쪽에도 과도한 부하가 걸리지 않게 한다.

Provider 전환을 설계할 때 주의할 점 하나는 각 프로바이더의 강점과 약점을 파악하는 것이다. 한 프로바이더는 비용이 낮지만 속도가 느리고, 다른 프로바이더는 속도가 빠르지만 비용이 높을 수 있다. 이런 trade-off를 고려해 트래픽을 분배한다.

Provider 전환은 단순한 기술 구현에 그치지 않는다. 비용, 데이터

개인정보, 컴플라이언스 같은 비즈니스 측면까지 고려해야 한다. 특히 사용자의 데이터를 외부 프로바이더에 보내는 것이 허용되는지 확인해야 한다.

5.5 4. 운영 환경에서의 비용과 버전 모니터링

비용과 버전 관리는 설계 단계에서 끝나지 않는다. 시스템이 운영되는 동안에도 계속 모니터링해야 한다. 사용량, 비용, 모델 버전의 변화를 실시간으로 추적해야 이상 징후를 빨리 포착할 수 있다.

모니터링 시스템에서는 주로 다음 지표를 추적한다. 토큰 사용량, API 호출 횟수, 평균 응답 시간, 에러 발생률, 모델 버전 분포, 프로바이더별 트래픽 비중이다. 이 지표를 대시보드로 시각화하면 현재 시스템 상태를 한눈에 파악할 수 있다.

비용과 버전 관리의 최종 목표는 예측 가능한 시스템 운영이다. 청구서가 예상 밖으로 나오거나 모델의 행동이 바뀌어 문제가 생긴다면 그건 관리의 실패다. 체계적인 모니터링과 빠른 대응이 예측 가능한 운영을 만든다.

5.6 정리

이 책에서는 AI API 계약 엔지니어링의 네 가지 축을 다루었다. 계약 설계의 사고법, 입력 계약, 출력 계약, 장애 복원, 그리고 비용과 버전 관리까지다. LLM을 시스템 컴포넌트로 쓸 때 이 원칙을 적용하면 더 예측 가능하고 안전하며 비용 효율적인 시스템을 만들 수 있다. AI 모델은 강력한 도구이지만, 그 힘을 제대로 쓰려면 공학적 원칙이 뒷받침돼야 한다.