LLM을 신뢰하지 않는 제안자로: 안전한 LLM 통합을 위한 제로 트러스트 아키텍처

> **초안 버전:** 0.1 **대상 학회:** 시스템 / 아키텍처 / LLM 안전성 트랙

초록 (Abstract)

대규모 언어 모델(LLM)이 상태를 관리하는 소프트웨어 시스템에 점점 더 많이 통합되고 있다. 그러나 대부분의 기존 통합 방식은 LLM 출력을 *신뢰할 수 있는 결과*로 암묵적으로 취급하여, 비결정적인 생성 결과가 시스템 상태에 직접 영향을 미치도록 허용한다. 이 가정은 근본적으로 결함이 있다: LLM은 구조적 정확성을 보장할 수 없으며, 생성과 실행이 혼합될 때 실패는 예측 불가능하게 전파된다.

우리는 LLM을 권위 있는 의사결정자가 아닌 *신뢰하지 않는 제안자(untrusted proposer)*로 취급하는 제로 트러스트 아키텍처 패턴인 **Manifesto**를 제안한다. Manifesto에서 LLM 출력은 직접 실행되지 않는다. 대신, 구조적 유효성의 유일한 판정자 역할을 하는 결정론적 헌법 계층(Builder)에 제안(proposal)으로 제출된다. 명시적이고 결정론적인 제약을 만족하는 제안만 수락되며, 그 외의 것들은 거부되거나 해결을 위해 보류된다. 이러한 분리는 LLM의 행동과 관계없이 시스템 안전성과 결정론이 보존되도록 보장한다.

우리의 접근 방식은 세 가지 기여를 한다:

1. **신뢰하지 않는 제안자 패턴(Untrusted Proposer Pattern)**: 비결정적 생성과 결정론적 판단 사이의 원칙적 분리로, 모델 신뢰성에 의존하지 않고 안전한 LLM 통합을 가능하게 한다.
2. **지능 참여 루프(Intelligence-in-the-Loop, ITL)**: 인간 참여 루프(HITL)와 AI 참여 루프(AITL)를 일반화하는 통합 해결 프로토콜로, 책임성과 재현 가능성을 유지하면서 외부에서 모호성을 해결할 수 있게 한다.
3. **자기 호스팅 증명(Self-Hosting Proof)**: 자연어를 스키마로 변환하는 컴파일러를 Manifesto 애플리케이션 자체로 구현하여 아키텍처를 검증함으로써, 복잡하고 다단계이며 실패하기 쉬운 시스템을 이 모델 하에서 안전하게 구축할 수 있음을 보여준다.

우리는 LLM 통합 시스템의 안전성이 모델 정확성을 개선하는 것이 아니라, 실패 상황에서도 견고하게 유지되는 아키텍처 설계에 의존해야 한다고 주장한다. Manifesto는 그러한 설계를 제공한다.

키워드: LLM 통합, 제로 트러스트 아키텍처, 결정론적 시스템, 인간-AI 협업, 소프트웨어 아키텍처

---

1. 서론

1.1 LLM 통합 시스템의 부상

대규모 언어 모델은 독립형 채팅 인터페이스를 넘어 소프트웨어 시스템의 핵심으로 이동했다. 현대 애플리케이션은 LLM을 사용하여 코드를 생성하고, 데이터베이스를 조작하고, 워크플로우를 조율하며, 시스템 상태에 직접 영향을 미치는 결정을 내린다. LangChain, AutoGPT, Semantic Kernel과 같은 프레임워크는 이러한 통합을 일반 개발자들도 접근 가능하게 만들었다.

그러나 이 통합은 근본적인 긴장을 도입한다: 소프트웨어 시스템은 전통적으로 결정론적 동작에 의존하는 반면, LLM은 본질적으로 확률적이다. 함수 호출은 같은 입력에 대해 같은 결과를 반환한다; LLM은 그렇지 않다. 데이터베이스 쿼리는 성공하거나 정의된 오류로 실패한다; LLM은 구문적으로 유효하지 않은 출력, 의미적으로 잘못된 결과, 또는 자신감 있게 환각된 내용을 반환할 수 있다.

1.2 "기본적으로 신뢰" 문제

현재 LLM 통합 프레임워크들은 암묵적인 가정을 공유한다: **LLM 출력은 신뢰할 수 있고 직접 실행할 수 있다.** LLM이 함수 호출을 생성하면, 프레임워크는 그것을 실행한다. LLM이 JSON 객체를 생성하면, 시스템은 그것을 파싱하고 사용한다. LLM이 결정을 내리면, 애플리케이션은 그에 따라 행동한다.

이 "기본적으로 신뢰" 모델은 LLM이 올바르게 동작할 때 작동한다. 하지만 LLM은 항상 올바르게 동작하지 않는다. 그들은 환각한다. 잘못된 형식의 출력을 생성한다. 지시를 오해한다. 그럴듯해 보이지만 의미적으로 유효하지 않은 결과를 생성한다.

신뢰받는 LLM 출력이 실패하면, 실패는 시스템으로 전파된다. 유효하지 않은 JSON은 파서를 충돌시킨다. 잘못된 함수 호출은 상태를 손상시킨다. 환각된 결정은 정의되지 않은 동작으로 이어진다. 시스템은 LLM의 권위에 의문을 제기하지 않았기 때문에 방어책이 없다.

1.3 우리의 접근: 신뢰하지 않는 제안자로서의 LLM

우리는 LLM이 시스템에 통합되는 방식에 대한 근본적인 전환을 제안한다. LLM 출력을 신뢰할 수 있는 결과로 취급하는 대신, 우리는 그것들을 결정론적 검증의 대상이 되는 **신뢰하지 않는 제안**으로 취급한다.

우리 모델에서:

- **LLM은 제안자이다**: 후보 출력을 생성하지만, 실행할 권한이 없다.
- **Builder는 판정자이다**: 구조적으로 유효한 제안만 수락하는 결정론적 검증 계층이다.
- **해결은 외부적이다**: 검증이 정확성을 결정할 수 없을 때(모호성), 결정은 외부 권위자—인간 또는 AI—에게 위임된다.

이 분리는 **LLM 실패가 격리됨**을 보장한다. LLM은 어떤 출력이든—잘못된 형식, 부정확한, 또는 악의적인—생성할 수 있지만, 헌법적 검증을 통과하지 않으면 아무것도 실행되지 않기 때문에 시스템은 안전하게 유지된다.

1.4 기여

이 논문은 세 가지 기여를 한다:

1. **신뢰하지 않는 제안자 패턴** (섹션 3): 비결정적 생성과 결정론적 판단 사이의 분리를 재사용 가능한 아키텍처 패턴으로 형식화한다. 안전성, 결정론, 실패 격리라는 형식적 속성을 정의한다.
2. **지능 참여 루프(ITL)** (섹션 4.4): 인간 참여 루프(HITL)와 AI 참여 루프(AITL) 모두를 포괄하는 모호성 해결을 위한 통합 프로토콜을 도입한다. 이 프로토콜은 누가—인간이든 AI든—해결 결정을 내리든 상관없이 책임성을 보존하고 재생을 가능하게 한다.
3. **자기 호스팅 증명** (섹션 5): 자연어를 스키마로 변환하는 컴파일러를 Manifesto 애플리케이션으로 구현하여 아키텍처를 검증한다. 이 컴파일러는 생성에 LLM을, 검증에 Builder를, 모호성 해결에 ITL을 사용하며—복잡하고

실패하기 쉬운 시스템이 이 모델 하에서 안전하게 구축될 수 있음을 보여준다.

1.5 논문 구성

섹션 2는 제로 트러스트 개념에 대한 배경과 기존 LLM 프레임워크의 신뢰 모델을 분석한다. 섹션 3은 신뢰하지 않는 제안자 패턴을 형식화한다. 섹션 4는 Manifesto 아키텍처를 제시한다. 섹션 5는 자기 호스팅 컴파일러 구현을 설명한다. 섹션 6은 실패 격리와 결정론 속성을 분석한다. 섹션 7은 한계와 향후 연구를 논의한다. 섹션 8은 관련 연구를 조사한다. 섹션 9는 결론을 맺는다.

---

2. 배경 및 동기

2.1 제로 트러스트: 네트워크 보안에서 AI 시스템으로

제로 트러스트 아키텍처는 경계 기반 보안 모델의 한계에 대한 대응으로 네트워크 보안에서 유래했다. 전통적인 접근—"방화벽 내부의 모든 것을 신뢰하라"—는 위협이 진화함에 따라 부적절함이 증명되었다. 제로 트러스트는 이 가정을 뒤집는다: **아무것도 신뢰하지 말고, 모든 것을 검증하라**.

제로 트러스트의 핵심 원칙:

1. **절대 신뢰하지 말고, 항상 검증하라**: 모든 요청은 출처와 관계없이 인증되고 권한이 부여된다.
2. **침해를 가정하라**: 어떤 구성 요소든 손상될 수 있다고 가정하고 시스템을 설계한다.
3. **최소 권한**: 각 작업에 필요한 최소 권한만 부여한다.

우리는 이 원칙들이 LLM 통합에 직접 적용된다고 주장한다:

네트워크 보안	LLM 통합
내부 트래픽을 신뢰하지 마라	LLM 출력을 신뢰하지 마라
모든 요청을 검증하라	모든 생성을 검증하라
침해를 가정하라	환각을 가정하라
최소 권한	최소 LLM 권한

2.2 LLM 실패 모드

LLM 출력은 여러 방식으로 실패할 수 있으며, 각각은 시스템 안전성에 다른 함의를 가진다:

구문적 실패: 출력이 예상 형식에 따라 유효하지 않다.

- 잘못된 형식의 JSON
- 불완전한 응답 (잘림)

- 잘못된 데이터 타입

구조적 실패: 출력이 구문적으로 유효하지만 스키마 제약을 위반한다.

- 필수 필드 누락
- 유효하지 않은 열거형 값
- 타입 불일치

의미적 실패: 출력이 구조적으로 유효하지만 의미적으로 잘못되었다.

- 존재하지 않는 엔티티에 대한 환각된 참조
- 논리적으로 불가능한 상태
- 모순된 정보

적대적 실패: 출력이 시스템을 악용하려 한다.

- 프롬프트 주입
- 지시 무시
- 권한 상승

현재 프레임워크들은 주로 구조화된 생성(제약된 디코딩)을 통해 구문적 실패를 다룬다. 구조적 실패는 스키마 검증을 통해 부분적으로 다뤄진다. 의미적 및 적대적 실패는 아키텍처 수준에서 대체로 다뤄지지 않는다.

2.3 기존 프레임워크의 신뢰 모델

인기 있는 LLM 통합 프레임워크의 암묵적 신뢰 모델을 분석한다:

LangChain / LangGraph: LLM 출력이 직접 도구를 호출하고 상태를 수정한다. 프레임워크는 LLM이 생성한 함수 호출이 유효하고 안전하다고 신뢰한다. 실패 처리는 예방적(실행 전 검증)이 아니라 반응적(예외 포착)이다.

AutoGPT / BabyAGI: 자율 에이전트가 LLM이 생성한 다단계 계획을 실행한다. 시스템은 LLM의 계획과 의사결정을 신뢰한다. 인간 감독은 아키텍처적이 아니라 선택적이다.

DSPy: 최적화와 함께하는 선언적 프롬프팅. 최적화 과정이 신뢰할 수 있는 프롬프트를 생성할 것이라는 신뢰가 있다. 실행을 위한 기본 신뢰 모델은 변경되지 않는다.

Guidance / Outlines: 제약된 생성이 구문적 유효성을 보장한다. 이는 구문적 실패를 다루지만 구조적 및 의미적 정확성에 대해서는 여전히 LLM을 신뢰한다.

Semantic Kernel: 플러그인을 통한 AI 오케스트레이션. LLM 출력이 플러그인 실행을 트리거한다. 신뢰 모델은 LangChain과 유사하다.

이 모든 프레임워크는 공통 패턴을 공유한다: **LLM 출력이 최소한의 검증으로 실행으로 흐른다**. 암묵적 가정은 LLM이 실용적으로 충분히 신뢰할 수 있고, 실패는 재시도나 예외 처리를 통해 처리할 수 있다는 것이다.

2.4 왜 "더 좋은 LLM"이 답이 아닌가

이러한 신뢰 문제는 LLM이 개선되면 사라질 것이라고 주장할 수 있다. 우리는 근본적인 이유로 동의하지 않는다:

확률적 특성: LLM은 확률적 모델이다. 개선되더라도 결정론적 보장을 제공할 수 없다. 99.9% 성공률도 규모에서는 여전히 실패를 의미한다.

적대적 견고성: LLM이 더 유능해지면, 그에 대한 공격도 그렇다. 프롬프트 주입과 탈옥은 모델 능력과 함께 진화한다.

검증 복잡성: 많은 작업에서 정확성을 검증하는 것이 정확한 출력을 생성하는 것보다 쉽다. 이 비대칭성은 검증 기반 아키텍처를 선호한다.

책임 요구사항: 안전이 중요하거나 규제받는 도메인에서 "LLM이 그렇게 말했다"는 허용 가능한 정당화가 아니다. 결정론적 검증은 감사 가능한 결정을 제공한다.

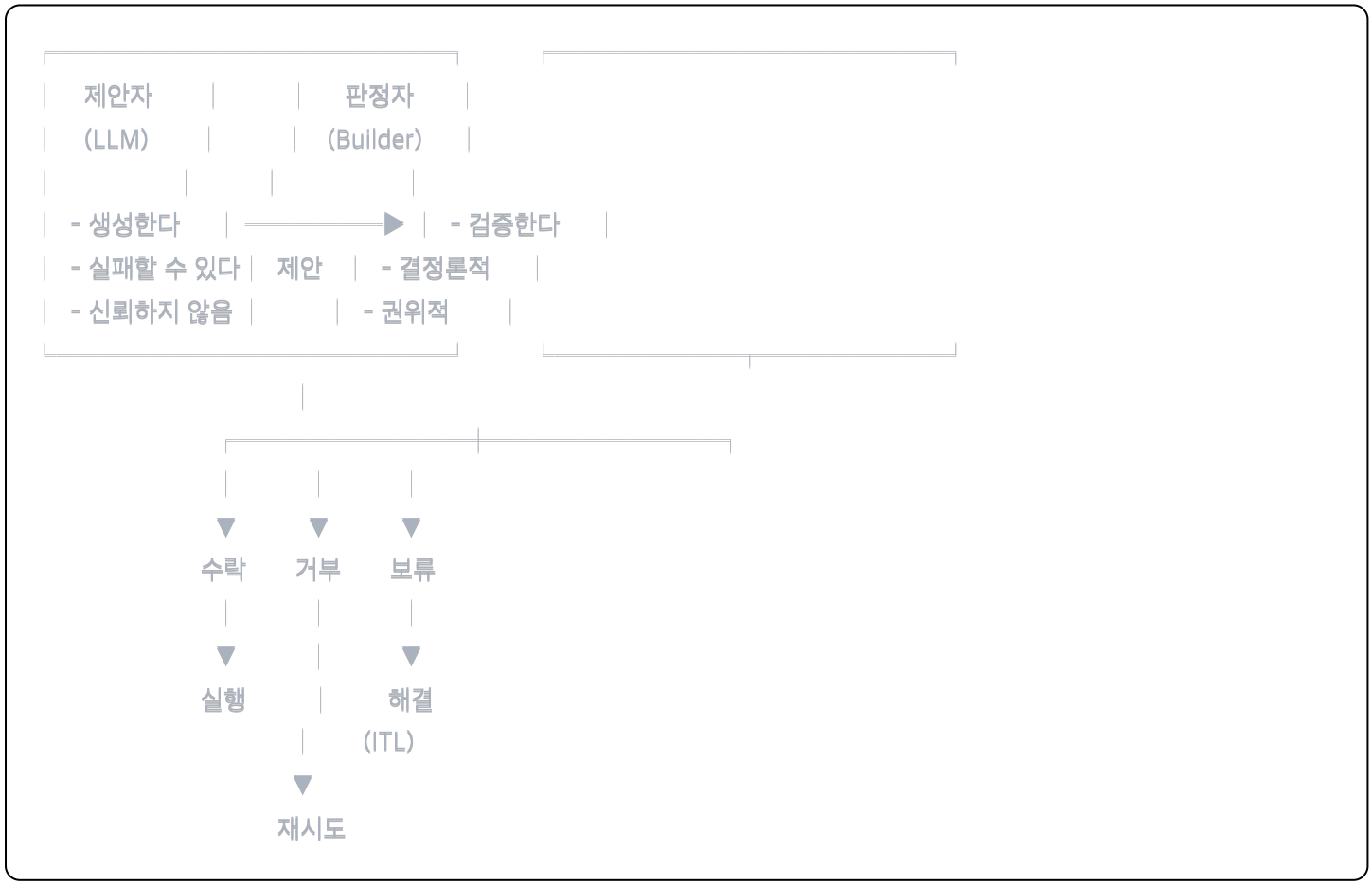
우리는 **안전성이 LLM 신뢰성에 의존해서는 안 된다**고 결론짓는다. 대신, 안전성은 LLM 동작과 관계없이 견고하게 유지되는 아키텍처 설계에서 나와야 한다.

---

3. 신뢰하지 않는 제안자 패턴

3.1 핵심 원칙: 제안과 판단의 분리

신뢰하지 않는 제안자 패턴은 생성과 검증 사이의 분리를 형식화한다:



제안자 (LLM): 후보 출력을 생성한다. 제안자는 권한이 없다—그 출력은 항상 신뢰하지 않는 제안으로 취급된다. 제안자는 시스템 안전성을 손상시키지 않고 어떤 방식으로든 (구문적, 구조적, 의미적, 적대적) 실패할 수 있다.

판정자 (Builder): 결정론적 규칙에 대해 제안을 평가한다. 판정자는 구조적 유효성에 대한 유일한 권위이다. 세 가지 결정 중 하나를 내린다:

- **수락**: 제안이 모든 제약을 만족한다. 실행해도 안전하다.
- **거부**: 제안이 제약을 위반한다. 폐기하고 선택적으로 재시도한다.
- **보류**: 제약이 유효성을 결정할 수 없다 (모호성). 외부 해결에 위임한다.

3.2 제안자 역할

제안자 역할은 *할 수 없는 것*으로 정의된다:

1. **실행할 수 없다**: 제안은 시스템 상태에 직접적인 영향을 미치지 않는다.
2. **검증을 우회할 수 없다**: 모든 제안은 판정자를 통과한다.
3. **자체 검증할 수 없다**: 제안자는 자신의 출력을 인증할 수 없다.

이러한 제약은 **LLM 능력이 시스템 안전성과 직교**함을 의미한다. 더 유능한 LLM은 더 나은 제안을 생성하지만 (더 높은 수락률) 안전성을 손상시킬 수 없다. 덜 유능한 LLM은 더 나쁜 제안을 생성하지만 (더 많은 거부) 역시 안전성을 손상시킬 수 없다.

이 분리가 핵심 통찰이다: 우리는 제안 품질을 위해 LLM을 최적화하면서 안전성은 아키텍처에 의존한다.

3.3 판정자 역할

판정자 역할은 *해야 하는 것*으로 정의된다:

1. **결정론적 평가**: 같은 제안은 항상 같은 판단을 산출한다.
2. **명시적 규칙**: 검증 기준은 학습되는 것이 아니라 정의된다.
3. **완전한 범위**: 모든 제안은 판단을 받는다.

판정자는 우리가 **헌법적 검증**이라고 부르는 것을 구현한다—구조적 유효성을 정의하는 고정된 규칙 집합. 이 규칙들은 헌법과 유사하다: 명시적으로 정의되고, 거의 변경되지 않으며, 권위 있게 해석된다.

헌법적 규칙은 다음을 표현할 수 있다:

- **타입 제약**: 필드 X는 문자열이어야 한다.
- **구조적 제약**: A가 존재하면, B가 존재해야 한다.
- **참조적 제약**: 경로 X는 유효한 엔티티를 참조해야 한다.
- **논리적 제약**: 상태 A와 B는 상호 배타적이다.

헌법적 규칙이 *표현할 수 없는 것*은 의미적 정확성—제안이 사용자가 의도한 것을 *의미하는지* 여부다. 이 한계는 근본적이며 해결 메커니즘을 통해 다뤄진다.

3.4 형식적 속성

신뢰하지 않는 제안자 패턴은 세 가지 형식적 보장을 제공한다:

속성 1: 안전성

> 유효하지 않은 제안은 절대 실행되지 않는다.

증명 스케치: 실행은 판정자 수락 후에만 발생한다. 판정자는 헌법적 규칙을 만족하는 제안만 수락한다. 따라서 유효한 제안만 실행된다.

속성 2: 결정론

> 같은 입력, 해결 결정, 제안이 주어지면, 시스템은 같은 출력을 생성한다.

증명 스케치: 판정자는 결정론적이다. 해결 결정은 기록된다. 기록된 결정이 주어지면, 실행 경로는 완전히 결정된다.

속성 3: 실패 격리

> LLM 실패는 시스템 실패로 전파되지 않는다.

증명 스케치: LLM 출력은 제안이다. 제안은 실행 전에 검증된다. 유효하지 않은 제안은 실행되지 않고 거부된다. 따라서 LLM 실패는 시스템 실패가 아닌 거부로 귀결된다.

이러한 속성은 적대적 행동을 포함한 LLM 동작과 관계없이 유지된다. 아키텍처는 구조적으로 안전하다.

---

4. Manifesto 아키텍처

Manifesto는 완전한 시스템 아키텍처로서의 신뢰하지 않는 제안자 패턴 구현이다. 이 섹션은 그 구성 요소를 설명한다.

4.1 개요

Core: 결정론적 의미 계산기. 상태 스냅샷과 액션이 주어지면, Core는 결과 상태를 계산한다. Core는 부작용이 없다—순수 함수이다.

Builder: 헌법적 검증자. Builder는 도메인 스키마를 정의하고 모든 상태 전환이 정의된 제약을 준수하는지 검증한다. Builder는 신뢰하지 않는 제안자 패턴에서 판정자 역할을 한다.

Host: 실행 런타임. Host는 계산 루프를 관리하고, effects(LLM 호출과 같은 부작용)를 실행하며, 상태 전환을 조정한다. Host는 외부 세계와 상호작용하는 유일한 구성 요소이다.

World Protocol: 거버넌스 계층. World Protocol은 actors(누가 제안할 수 있는가), authorities(누가 승인할 수 있는가), 그리고 그들의 상호작용을 지배하는 규칙을 정의한다. 이 계층이 ITL을 가능하게 한다.

4.2 스냅샷으로서의 상태

Manifesto의 핵심 설계 원칙은 **스냅샷 중심 상태 관리**이다:

```typescript
type Snapshot = {
  readonly state: State;          // 도메인 상태
  readonly computed: Computed;    // 파생된 값
  readonly available: Available;  // 유효한 액션
  readonly explain: Explain;      // 왜 액션이 (불)가능한지
};
```

스냅샷은:

- **불변**: 한번 생성되면, 절대 수정되지 않는다.
- **완전**: 시스템 상태를 이해하는 데 필요한 모든 정보를 포함한다.
- **결정론적**: 같은 입력 상태는 항상 같은 스냅샷을 생성한다.

이 설계는 다음을 가능하게 한다:

- **재생**: 초기 상태와 액션 시퀀스가 주어지면, 모든 스냅샷을 재현한다.

- **디버깅**: 모든 과거 상태를 검사한다.
- **감사**: 상태 전환이 유효했는지 검증한다.

4.3 경계로서의 Effects

Manifesto는 **순수 계산**과 **effects**를 구분한다:

```
순수 (결정론적):
  - 상태 계산
  - 검증
  - 가용성 확인

Effects (비결정론적):
  - LLM 호출
  - 외부 API 호출
  - 데이터베이스 작업
  - 사용자 상호작용
```

Effects는 실행이 아닌 선언이다:

```typescript
// Effect는 flow에서 선언된다
flow.effect('llm:generate', { prompt: '...' })

// Host가 effect를 실행하고 결과를 반환한다
// 결과는 다음 계산의 입력이 된다
```

이 분리는 LLM 통합에 중요하다: **LLM 호출은 effects이고, 그 결과는 제안이다**. LLM 생성의 비결정성은 effect 경계 내에 격리된다. 그 이후에 일어나는 것—검증과 실행—은 결정론적이다.

4.4 지능 참여 루프 (ITL)

판정자가 유효성을 결정할 수 없을 때—진정한 모호성이 있을 때—결정은 외부 권위자에게 위임되어야 한다.
Manifesto는 이 해결을 위한 통합 프로토콜로 **지능 참여 루프(Intelligence-in-the-Loop, ITL)**를 도입한다.

ITL은 두 가지 기존 개념을 일반화한다:

- **HITL (Human-in-the-Loop)**: 인간이 결정을 내린다.
- **AITL (AI-in-the-Loop)**: AI 시스템이 결정을 내린다.

```
ITL (Intelligence-in-the-Loop)
├── HITL (인간)
│   └── 인간이 옵션을 검토하고, 하나를 선택한다
└── AITL (AI)
    ├── 단일 LLM이 결정한다
    ├── 여러 LLM이 투표한다 (합의)
    ├── 헌법적 재판소 (규칙 기반 AI)
    └── 하이브리드 프로토콜
```

핵심 통찰은 **시스템이 누가 모호성을 해결하는지 알 필요가 없다**는 것이다. 알아야 하는 것은:

1. 해결이 필요하다 (옵션이 존재한다)
2. 선택이 이루어졌다 (옵션 ID)
3. 선택이 기록되었다 (재생을 위해)

```typescript
// 시스템 관점
status: 'awaiting_resolution'
options: [{ id: 'a', description: '...' }, { id: 'b', description: '...' }]

// 해결 후 (누가 했는지는 상관없다)
resolve({ selectedOptionId: 'a' })

// 시스템이 계속된다
status: 'continuing'
```

이 설계는 중요한 속성을 가진다:

Actor 중립적: 시스템은 인간과 AI 해결자를 동일하게 취급한다. 해결자 유형에 따른 특별한 권한이나 제한이 없다.

책임 있는: 모든 해결은 선택과 함께 기록된다. 감사 추적이 완전하다.

재생 가능한: 기록된 해결이 주어지면, 시스템 실행을 정확히 재현할 수 있다.

유연한: 핵심 시스템을 수정하지 않고 해결 메커니즘을 변경할 수 있다. 배포 요구사항에 따라 HITL을 AITL로 교체하거나 그 반대로 할 수 있다.

---

5. 자기 호스팅 증명: 컴파일러

Manifesto 아키텍처를 검증하기 위해, 우리는 그것을 사용하여 중요한 시스템을 구현했다: 자연어를 스키마로 변환하는 컴파일러. 이 컴파일러는 자연어로 표현된 인간의 요구사항을 유효한 Manifesto 도메인 스키마로 변환한다.

5.1 왜 컴파일러인가?

컴파일러는 이상적인 검증 대상이다:

복잡성: 컴파일은 여러 단계를 포함하며, 각각 실패 모드가 있다. Manifesto가 이 복잡성을 처리할 수 있다면, 더 단순한 애플리케이션은 분명히 가능하다.

실패하기 쉬움: 자연어 이해는 본질적으로 불확실하다. 컴파일러는 모호한 입력, 잘못된 해석, 유효하지 않은 생성을 처리해야 한다.

다단계: 파이프라인(분할 → 정규화 → 제안 → 검증)은 복잡한 워크플로우가 Manifesto에서 모델링될 수 있음을 보여준다.

자기 참조적: Manifesto를 사용하여 Manifesto 스키마를 생성하는 컴파일러는 아키텍처가 자신을 위한 도구를 구축할 수 있음을 보여준다—가장 강력한 형태의 검증.

5.2 Manifesto 애플리케이션으로서의 컴파일러

컴파일러는 표준 Manifesto 애플리케이션으로 구현된다:

```typescript
const CompilerStateSchema = z.object({
  // 파이프라인 상태
  status: z.enum([
    'idle', 'segmenting', 'normalizing',
    'proposing', 'validating', 'awaiting_resolution',
    'success', 'discarded'
  ]),

  // 입력
  input: z.string().nullable(),
  targetSchema: z.any().nullable(),

  // 중간 결과
  segments: z.array(z.string()),
  intents: z.array(NormalizedIntentSchema),
  currentDraft: z.any().nullable(),

  // 출력
  result: DomainSchemaSchema.nullable(),
  discardReason: DiscardReasonSchema.nullable(),
});
```

컴파일러는 각 상태 전환을 위한 액션을 정의한다:

- `start`: 입력 텍스트로 컴파일 시작

- `receiveSegments`: 분할 결과 처리
- `receiveIntents`: 정규화 결과 처리
- `receiveDraft`: LLM 생성 초안 처리
- `receiveValidation`: Builder 검증 결과 처리
- `resolve`: 모호성에 대한 외부 해결 수락
- `discard`: 이유와 함께 컴파일 포기
- `reset`: 유휴 상태로 복귀

5.3 컴파일 파이프라인

각 LLM 호출은 effect이다:

```typescript
flow.effect('llm:segment', { text: state.input })
flow.effect('llm:normalize', { segments: state.segments, schema: state.targetSchema })
flow.effect('llm:propose', { intents: state.intents, history: state.attempts })
flow.effect('builder:validate', { draft: state.currentDraft })
```

LLM effects는 제안을 반환한다. `builder:validate` effect는 판정자를 호출한다. 검증이 실패하면, 컴파일러는 재시도한다 (최대값까지). 검증이 성공하면, 결과는 유효한 DomainSchema이다.

5.4 해결 흐름

모호성은 여러 지점에서 발생할 수 있다:

- **분할 모호성**: 입력이 여러 방식으로 분할될 수 있다.
- **정규화 모호성**: 세그먼트가 여러 해석을 가질 수 있다.
- **제안 모호성**: 여러 유효한 스키마가 의도를 만족할 수 있다.

LLM이 모호성을 감지하면, 해결 옵션을 반환한다:

```typescript
// LLM 응답
{
  ok: 'resolution',
  reason: '여러 유효한 해석',
  options: [
    { id: 'a', description: '상태 필드로 해석' },
    { id: 'b', description: '계산된 값으로 해석' }
  ]
}
```

컴파일러는 `awaiting_resolution`으로 전환한다:

```typescript
flow.patch(state.status).set('awaiting_resolution')
flow.patch(state.resolutionOptions).set(options)
```

해결은 외부적이다—컴파일러는 인간이 해결하는지 AI가 해결하는지 알지도 신경 쓰지도 않는다. 단순히 기다린다:

```typescript
dispatch({ type: 'resolve', input: { selectedOptionId: 'a' } })
```

이 깔끔한 분리는 같은 컴파일러가 다음에서 작동함을 의미한다:

- 인간 프롬프트가 있는 CLI (HITL)
- AI 해결이 있는 자동화된 파이프라인 (AITL)
- 에스컬레이션 정책이 있는 하이브리드 시스템

5.5 독푸딩 관찰

컴파일러를 구축하면서 여러 통찰이 드러났다:

패턴 검증: 우리가 설계한 모든 아키텍처 패턴이 필요함이 증명되었다. 상태 머신, effects, 검증, 해결—모두 컴파일러 구현에서 사용되었다.

엣지 케이스 발견: 구현은 순수 설계가 놓친 스펙의 엣지 케이스를 드러냈다. 이들은 아키텍처 확정 전에 수정되었다.

복잡성 처리: 컴파일러는 진정으로 복잡하지만 (8개 상태, 9개 액션, 4개 LLM effects, 해결 처리, 재시도 로직) 구현은 깔끔하게 유지되었다. 이는 아키텍처가 확장됨을 시사한다.

성능 특성: 검증 오버헤드는 LLM 지연 시간에 비해 무시할 수 있다. 아키텍처 비용은 최소이다.

---

6. 분석

6.1 실패 격리

다양한 실패 모드가 어떻게 처리되는지 분석한다:

실패 유형	예시	시스템 동작
구문적	LLM의 잘못된 형식의 JSON	파싱 실패 → 재시도
구조적	필수 필드 누락	검증 실패 → 재시도
의미적	잘못된 해석	검증 통과할 수 있음*
적대적	프롬프트 주입	검증이 구조적 위반 포착
LLM 타임아웃	API 실패	Effect 실패 → 재시도 또는 폐기
LLM 거부	모델이 거절	빈 결과 → 재시도 또는 폐기

*구조적으로 유효하지만 의미적으로 잘못된 출력을 생성하는 의미적 실패는 검증을 통과할 수 있다. 이는 논의에서 다루는 근본적인 한계이다.

핵심 관찰: **어떤 실패도 시스템을 충돌시키지 않는다**. 모든 실패는 재시도, 해결 요청, 또는 우아한 폐기로 귀결된다. 시스템은 전체적으로 유효한 상태를 유지한다.

6.2 결정론 보장

Manifesto는 조건부 결정론을 제공한다:

주어진 것:

- 초기 상태
- 액션 시퀀스
- LLM 응답 (기록된)

- 해결 결정 (기록된)

그러면: 최종 상태는 완전히 결정된다.

이는 무조건적 결정론(LLM에서는 불가능)보다 약하지만 다음에 충분하다:

- **디버깅**: 기록을 재생하여 모든 실행을 재현한다.
- **감사**: 특정 상태에 올바르게 도달했는지 검증한다.
- **테스트**: 결정론적 테스트를 위해 기록된 응답을 사용한다.

6.3 오버헤드 분석

Manifesto의 아키텍처 오버헤드는 다음으로 구성된다:

검증 비용: 각 제안은 헌법적 규칙에 대해 검증된다. 일반적인 도메인 스키마의 경우, 검증은 스키마 크기에 대해 O(n)이다—LLM 지연 시간(초 단위)에 비해 무시할 수 있다.

상태 관리: 불변 스냅샷은 메모리 할당이 필요하다. 실제로는 최근 스냅샷의 윈도우만 유지된다. 메모리 오버헤드는 제한된다.

해결 지연: 해결이 필요할 때, 실행은 해결될 때까지 블록된다. HITL의 경우, 이는 상당할 수 있다 (분에서 시간). AITL의 경우, 추가 LLM 호출이다 (초 단위).

작업	일반적인 지연 시간
LLM 생성	1-30초
검증	< 10 밀리초
HITL 해결	분에서 시간
AITL 해결	1-30초

지배적인 비용은 LLM 지연 시간이다. Manifesto의 오버헤드는 그에 비해 무시할 수 있다.

6.4 기존 접근과의 비교

측면	LangChain	AutoGPT	Guidance	Manifesto
신뢰 모델	신뢰	신뢰	제약	**불신뢰**
실패 처리	예외	재시도	문법	**구조적**
결정론	없음	없음	부분	**조건부**
인간-AI 프로토콜	임시	선택적	없음	**ITL**

측면	LangChain	AutoGPT	Guidance	Manifesto
자기 호스팅	아니오	아니오	아니오	**예**
의미적 안전	아니오	아니오	아니오	아니오*

*어떤 접근도 의미적 안전 보장을 제공하지 않는다. 이는 여전히 열린 문제이다.

---

7. 논의

7.1 이 패턴을 언제 사용하는가

신뢰하지 않는 제안자 패턴은 다음 경우에 가장 가치 있다:

안전이 중요할 때: 잘못된 동작이 중대한 결과를 가지는 애플리케이션—금융 시스템, 의료, 인프라 제어.

감사 가능성이 요구될 때: 결정 추적과 재현 가능성을 요구하는 규제 산업.

인간-AI 협업: 인간과 AI 에이전트가 함께 작업하며, 권위와 해결을 위한 명확한 프로토콜이 필요한 시스템.

비결정론이 허용되지 않을 때: 예측 가능하고 재현 가능한 동작을 요구하는 애플리케이션.

이 패턴은 다음에는 불필요할 수 있다:

- "잘못된" 출력이 허용되는 탐색적 또는 창의적 애플리케이션
- 실패가 최소한의 영향을 미치는 낮은 위험 애플리케이션
- 상태 관리가 없는 순수 생성 작업

7.2 한계

구조적 vs. 의미적: Manifesto는 구조적 위반을 포착하지만 의미적 오류는 포착하지 않는다. 스키마는 구조적으로 유효하지만 의미적으로 잘못될 수 있다. 이는 근본적이다—의미적 정확성은 검증이 완전히 포착할 수 없는 의도 이해를 필요로 한다.

검증 표현력: 헌법적 규칙은 검증 언어로 표현 가능해야 한다. 복잡한 도메인 불변식은 인코딩하기 어려울 수 있다.

해결 지연: HITL은 인간 속도의 지연을 도입한다. 실시간 애플리케이션의 경우, 이는 금지적일 수 있다.

LLM 의존성: 안전성은 LLM 신뢰성에 의존하지 않지만, 유용성은 의존한다. 낮은 LLM 성능은 많은 거부와 재시도를 의미하며, 사용자 경험을 저하시킨다.

7.3 향후 연구

의미적 검증 계층: 구조적 제약을 넘어 의미적 제약으로 검증을 확장—잠재적으로 검증을 위해 별도의 LLM을 사용(이는 신뢰 질문을 다시 도입하지만).

AITL 프로토콜: 합의 메커니즘, 신뢰 임계값, 에스컬레이션 정책을 포함한 AI 기반 해결을 위한 형식적 프로토콜.

형식적 검증: 잠재적으로 모델 체킹이나 정리 증명을 사용하여 안전성 속성을 형식적으로 증명.

성능 최적화: 증분 체킹과 캐싱을 통해 검증 오버헤드를 줄임.

---

8. 관련 연구

8.1 LLM 통합 프레임워크

LangChain [Chase 2022]은 프롬프트와 도구의 체인을 통해 LLM 기반 애플리케이션을 구축하는 프레임워크를 제공한다. LLM이 생성한 함수 호출의 신뢰할 수 있는 실행을 가정한다.

LangGraph는 그래프 기반 워크플로우 구성으로 LangChain을 확장하여, 신뢰할 수 있는 실행 모델을 유지하면서 더 복잡한 에이전트 동작을 가능하게 한다.

Semantic Kernel [Microsoft 2023]은 플러그인을 통한 AI 오케스트레이션을 제공하며, LangChain과 유사한 신뢰 모델이다.

AutoGPT [Gravitas 2023]와 BabyAGI [Nakajima 2023]는 자율 에이전트 능력을 보여주며, 최소한의 인간 감독으로 계획과 실행에 LLM을 신뢰한다.

이러한 프레임워크들은 안전성보다 능력을 우선시하며, LLM 신뢰성이 실용적으로 충분하다고 가정한다.

8.2 구조화된 생성

Guidance [Microsoft 2023]와 Outlines [.txt 2023]는 LLM 생성을 지정된 문법을 따르도록 제약하여, 구문적 유효성을 보장한다.

LMQL [Beurer-Kellner et al. 2023]은 타입 제약이 있는 LLM용 쿼리 언어를 제공한다.

이러한 접근들은 구문적 실패를 다루지만 의미적 정확성에 대해서는 여전히 LLM을 신뢰한다. 그들은 Manifesto와 상호 보완적이다—구조화된 생성은 제안 품질을 개선할 수 있고 Manifesto는 안전성을 보장한다.

8.3 LLM 안전성

Constitutional AI [Anthropic 2022]는 자기 비판을 통해 명시적 원칙을 따르도록 모델을 훈련한다. 이는 모델 수준 안전성—LLM을 더 신뢰할 수 있게 만드는 것이다.

RLHF (인간 피드백으로부터의 강화 학습) [OpenAI 2022]는 모델 출력을 인간 선호도와 정렬한다.

이러한 접근들은 LLM 신뢰성을 개선하지만 아키텍처적 안전 보장을 제공하지 않는다. Manifesto는 직교한다—모델 신뢰성과 관계없이 안전성을 제공한다.

8.4 제로 트러스트 아키텍처

제로 트러스트 네트워크 아키텍처 [Kindervag 2010]는 네트워크 보안을 위한 "절대 신뢰하지 말고, 항상 검증하라" 원칙을 확립했다.

BeyondCorp [Google 2014]는 기업 규모의 제로 트러스트 구현을 보여주었다.

우리는 이러한 원칙을 AI 통합에 적용하여, LLM을 실행 전 검증이 필요한 신뢰하지 않는 구성 요소로 취급한다.

---

9. 결론

대규모 언어 모델을 소프트웨어 시스템에 통합하는 것은 근본적인 도전을 제시한다: 신뢰할 수 없는 기반 위에 어떻게 신뢰할 수 있는 시스템을 구축하는가? 현재 접근들은 암묵적으로 LLM 출력을 신뢰하여, 비결정적 실패가 시스템 상태로 전파되도록 허용한다.

우리는 LLM을 신뢰하지 않는 제안자로 취급하는 제로 트러스트 아키텍처인 **Manifesto**를 제시했다. 생성(LLM)과 판단(Builder)을 분리함으로써, 우리는 시스템 안전성이 LLM 신뢰성에 의존하지 않도록 보장한다. 지능 참여 루프(ITL)를 도입함으로써, 우리는 모호성 해결에서 인간-AI 협업을 위한 통합 프로토콜을 제공한다. 자연어 컴파일러를 Manifesto 애플리케이션으로 구현함으로써, 우리는 복잡하고 실패하기 쉬운 시스템이 이 모델 하에서 안전하게 구축될 수 있음을 보여준다.

우리의 핵심 주장은 간단하다: **통합 시스템에서의 AI 안전성은 모델이 아닌 아키텍처에서 와야 한다**. 우리는 LLM을 완벽하게 신뢰할 수 있게 만들 수 없지만, LLM 동작과 관계없이 안전하게 유지되는 시스템을 설계할 수 있다.

LLM이 중요한 시스템에 더 깊이 내장됨에 따라, 안전성에 대한 아키텍처적 접근은 필수적이 될 것이다. Manifesto는 이 미래를 위한 기반을 제공한다.

참고문헌

[전체 인용으로 완성 예정]

- Anthropic. Constitutional AI. 2022.
- Beurer-Kellner et al. LMQL. 2023.
- Chase, H. LangChain. 2022.
- Google. BeyondCorp. 2014.
- Gravitas. AutoGPT. 2023.
- Kindervag, J. Zero Trust Network Architecture. 2010.
- Microsoft. Guidance. 2023.
- Microsoft. Semantic Kernel. 2023.
- Nakajima, Y. BabyAGI. 2023.
- OpenAI. RLHF. 2022.
- .txt. Outlines. 2023.

부록 A: 컴파일러 상태 머신

[전체 상태 머신 다이어그램 - 포함 예정]

부록 B: 검증 규칙 예시

[헌법적 규칙 예시 - 포함 예정]

부록 C: ITL 프로토콜 명세

[형식적 프로토콜 명세 - 포함 예정]

초안 v0.1 끝