기술동향 리포트

하네스 엔지니어링

Harness Engineering

AI 에이전트 시대, 경쟁력은 모델이 아니라
“고삐와 안장” 에서 나온다

Jihwan Woo, Ph.D.

Sr. Specialist Partner SA – AI/ML

AWS Partner Field APJ

2026 년 4 월

본 리포트의 내용은 저자 개인의 의견이며,
AWS 의 공식적인 입장을 대변하지 않습니다.

Contents

우리가 AI 에 대해 오해하고 있는 한 가지

요즘 AI 를 써보신 분이라면 한 번쯤 이런 경험이 있을 것이다. 긴 문서를 요약해달라고 했더니 앞부분만 정리하고 뒷부분은 슬쩍 빠뜨린다거나, 처음에는 꽤 그럴듯하게 답하던 AI 가 대화가 길어지면서 점점 엉뚱한 소리를 하기 시작한다거나. 이럴 때 우리는 보통 이렇게 생각한다. “아직 AI 가 덜 똑똑한 거겠지. 다음 버전이 나오면 나아지겠지.”

그런데 2026 년 초, 이 상식을 뒤집는 실험 결과 두 건이 나란히 발표되었다.

첫 번째는 OpenAI 의 이야기다. Codex 팀 소속 엔지니어 세 명이 다섯 달 동안 코드를 단 한 줄도 직접 치지 않고, 오로지 AI 만 시켜서 100 만 줄짜리 소프트웨어 제품을 완성했다 [6]. 두 번째는 LangChain 이라는 회사의 사례다. AI 코딩 능력을 측정하는 시험 (Terminal Bench 2.0) 에서, gpt-5.2-codex 라는 AI 모델은 그대로 두고 주변 시스템만 손봤더니 순위가 30 위권에서 단숨에 5 위권으로 뛰어올랐다 [2].

두 팀이 공통적으로 한 일은 “더 좋은 AI 모델을 찾는 것” 이 아니었다. 이들이 집중한 건 AI 를 둘러싼 환경, 규칙, 도구, 검증 장치를 정교하게 설계하는 일이었다. 업계에서는 이 접근법을 “하네스 엔지니어링 (Harness Engineering)” 이라고 부르기 시작했다. 물론 이것이 AI 의 모든 문제를 해결하는 궁극적 답은 아니다. 장기적으로는 AI 모델 자체가 더 똑똑해져야 한다. 하지만 그때까지, 지금 가진 AI 로 최대한의 성과를 내려면 하네스가 필수적이다.

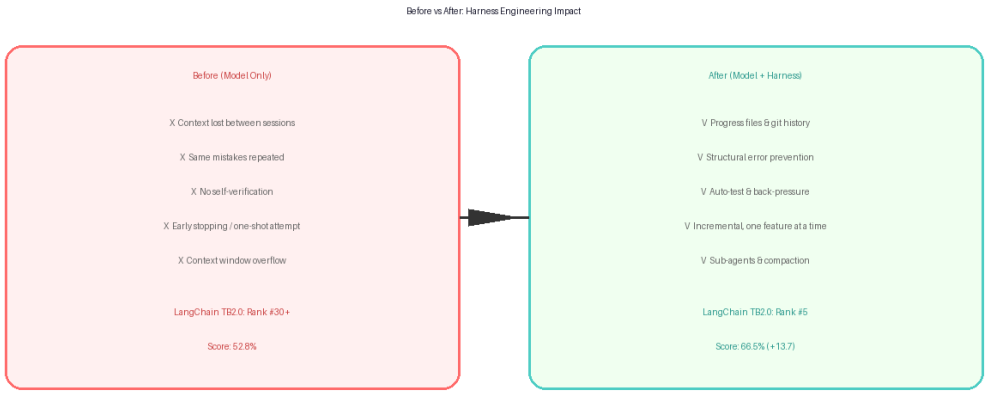

Figure 1: 같은 AI 모델을 쓰면서 주변 시스템 (하네스) 만 바꿨을 때의 성능 차이

하네스란 무엇인가

용어 안내

LLM(대형 언어 모델): ChatGPT, Claude 처럼 텍스트를 이해하고 생성하는 AI 의 핵심 두뇌.
AI 에이전트: LLM 에 도구와 규칙을 붙여서, 혼자 계획을 세우고 실행까지 할 수 있게 만든 AI 시스템.
하네스 (Harness): AI 에이전트에서 두뇌 (모델) 를 제외한 나머지 전부. 규칙, 도구, 기억장치, 안전장치 등을 통칭한다.

하네스 (harness) 라는 단어는 원래 말에게 씌우는 고삐와 안장 같은 장비를 뜻한다. 경마를 떠올려보면 이해가 쉽다. 경주마가 트랙에서 최고의 기량을 발휘하려면 말 자체의 능력만으로는 부족하다. 기수가 방향과 속도를 조절하는 고삐, 기수가 안정적으로 앉을 수 있는 안장, 말의 발을 보호하고 접지력을 높여주는 말발굽 편자까지 갖춰져야 한다. 이 장비들이 없으면 아무리 빠른 말이라도 경주에서 제 실력을 발휘할 수 없다.

AI 도 사정이 비슷하다. 여기서 말은 ChatGPT 나 Claude 같은 대형 언어 모델이고, 고삐와 안장과 말발굽은 모델을 감싸는 하네스이며, 기수는 이 하네스를 설계하는 엔지니어에 해당한다. 대형 언어 모델은 놀라운 능력을 갖고 있지만, 날것 그대로는 쓸모가 제한적이다. 이전 대화를 기억하지 못하고, 코드를 직접 실행할 수도 없고, 인터넷 검색도, 파일 저장도 할 수 없다. 이런 한계를 메워주는 주변 시스템 전체를 가리켜 “하네스” 라고 부른다.

LangChain 의 Vivek Trivedy 는 이를 한 줄로 정리했다 [1].

> *“Agent = Model + Harness. 모델이 아닌 모든 것이 하네스다.”*

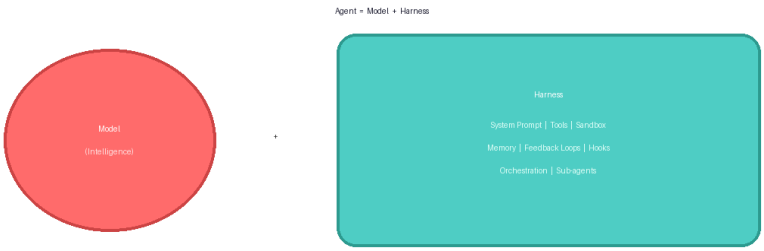

Figure 2: AI 에이전트 = 두뇌 (모델) + 고삐와 안장 (하네스)

이 개념에 처음으로 이름을 붙인 사람은 개발자 도구 Terraform 으로 유명한 Mitchell Hashimoto 다. 그는 2026 년 2 월 자신의 블로그에 이렇게 썼다 [3].

> *“AI 가 실수할 때마다, 같은 실수가 두 번 다시 일어나지 않도록 시스템을 고치는 것. 그것이 하네스 엔지니어링이다.”*

핵심은 “AI 한테 다음부터 잘하라고 말하는 것” 이 아니라는 점이다. 같은 실수가 구조적으로 재발할 수 없도록 환경 자체를 바꾸는 것이다. 공장에서 불량이 나왔을 때 작업자를 혼내는 대신 공정 라인을 재설계하는 것과 같은 발상이다.

왜 AI 두뇌만으로는 부족한가

매일 기억을 잃는 천재

이해를 돕기 위해 비유를 하나 들어보겠다. 매일 아침 새로운 직원이 출근하는 회사가 있다고 하자. 이 직원은 실력이 뛰어나지만, 어제 누가 어떤 일을 했는지 전혀 모른다. 인수인계 문서도 없고, 업무 현황판도 없고, 테스트 환경도 갖춰져 있지 않다. 아무리 능력이 출중해도 이런 조건에서 제대로 된 성과를 내기는 어렵다.

오늘날의 AI 가 정확히 이 상황이다. 새 대화창을 열 때마다 AI 는 백지 상태에서 시작한다. ChatGPT 에게 “아까 얘기하던 거 이어서 해줘” 라고 했을 때 AI 가 멈칫하는 이유가 바로 여기에 있다.

Anthropic 이 목격한 네 가지 실패

Claude 를 만든 회사 Anthropic 의 연구팀은 자사의 최고 성능 모델에게 복잡한 웹사이트 개발을 맡겨보았다. 결과는 기대와 달랐다. 같은 유형의 실패가 반복적으로 나타났다 [4].

실패 유형	쉽게 말하면	빠진 것
벼락치기	시험 범위 전체를 하룻밤에 끝내려다 중간에 나가떨어짐	단계별 계획
성급한 완료 선언	청소를 대충 하고 “다 했어요”	완료 기준표
검증 없는 마무리	요리를 맛보지 않고 손님 테이블에 내놓기	자기 점검 도구
기억 상실	전임자 인수인계 없이 첫 출근	진행 기록

Table 1: Anthropic 이 관찰한 AI 의 반복적 실패 패턴

이 문제들의 공통점은 분명하다. 전부 AI 의 “머리” 가 나빠서 생긴 일이 아니라, AI 를 둘러싼 “환경” 이 갖춰지지 않아서 생긴 일이다.

정보가 많아질수록 오히려 멍청해지는 역설

> **용어 안내**
>
> **컨텍스트 윈도우**: AI 가 한 번에 읽고 기억할 수 있는 정보의 총량. 사람으로 치면 “작업 기억력” 에 해당한다. 이 공간이 가득 차면 AI 는 앞서 받은 지시를 슬슬 잊기 시작한다.

여기서 흥미로운 연구 결과를 하나 소개하겠다. Chroma Research 가 2025 년에 최신 AI 모델 18 종을 대상으로 실험한 결과, **AI 에게 주는 정보가 많아질수록 오히려 성능이 떨어지는 현상**이 확인되었다 [9]. 업계에서는 이를 “컨텍스트 부패 (Context Rot)” 라고 부른다.

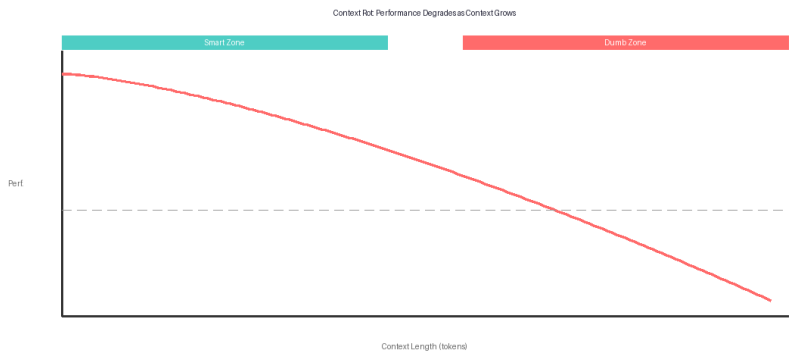

Figure 3: 컨텍스트 부패: 정보를 많이 줄수록 AI 의 판단력이 떨어지는 현상

책상 위에 서류가 열 장일 때는 필요한 문서를 금방 찾을 수 있다. 그런데 서류가 천 장으로 늘어나면? 정작 급한 서류를 찾는 데 시간을 다 쓰게 된다. AI 의 “책상” 을 깔끔하게 정리해주는 것, 이것이 하네스가 하는 가장 중요한 일 중 하나다.

하네스의 다섯 가지 핵심 장치

그렇다면 하네스는 구체적으로 어떤 장치들로 이루어져 있을까? 여러 선도 기업의 사례를 종합하면, 효과적인 하네스에는 다섯 가지 핵심 요소가 있다 [1, 7].

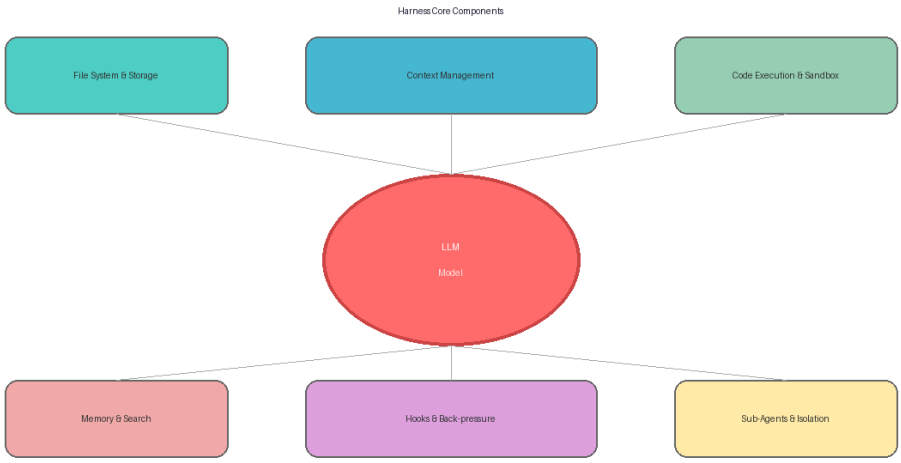

Figure 4: AI 두뇌 (모델) 를 감싸는 하네스의 핵심 구성 요소

인수인계 노트 — 기억을 이어주는 장치

앞서 “매일 기억을 잃는 천재” 비유를 들었다. 해법은 의외로 단순하다. 퇴근 전에 인수인계 노트를 쓰게 하면 된다. Anthropic 은 실제로 이 방법을 적용했다. AI 가 작업을 마칠 때마다 진행 상황을 파일에 기록하게 하고, 다음 AI 세션이 그 파일을 읽고 시작하도록 설계한 것이다 [4].

안전한 실험실 — 코드를 격리해서 실행하는 장치

AI 가 작성한 코드를 아무런 보호 장치 없이 바로 실행하는 것은 위험하다. 그래서 “샌드박스” 라 불리는 격리된 실행 환경을 마련한다. 화학 실험실에서 보호 장비를 갖추고 실험하는 것과 비슷하다.

깔끔한 책상 — 정보 과부하를 막는 장치

컨텍스트 부패 문제를 해결하기 위한 세 가지 전략이 있다.

전략	하는 일	비유
요약 정리	정보가 쌓이면 핵심만 추려서 정리	책상 위 서류를 치우고 메모만 남기기
결과물 분리 보관	긴 결과는 별도 저장하고 요약만 표시	원본은 서랍에, 요약본만 책상 위에
필요할 때만 꺼내기	도구와 지침을 한꺼번에 주지 않고 상황에 맞게 제공	공구함에서 필요한 공구만 꺼내 쓰기

Table 2: AI 의 “책상” 을 깔끔하게 유지하는 세 가지 전략

업무 위임 — 맥락 오염을 막는 장치

복잡한 프로젝트를 진행할 때, 자료 조사나 코드 탐색 같은 중간 과정을 별도의 AI(서브 에이전트) 에게 맡기고, 메인 AI 에게는 정리된 결과만 전달하는 방식이다. 팀장이 팀원에게 "이 건 조사해서 한 페이지로 요약해줘" 라고 지시하는 것과 같다. HumanLayer 는 이 구조를 "컨텍스트 방화벽" 이라고 부른다 [7].

"그냥 AI 의 기억 용량을 키우면 되지 않느냐" 고 물을 수 있다. 하지만 앞서 본 Chroma 의 연구가 보여주듯, 기억 용량을 키우는 것은 건초더미를 더 크게 만드는 것일 뿐이다. 그 안에서 바늘을 찾는 능력이 좋아지는 것은 아니다.

자동 품질 검사 — 실수를 잡아내는 장치

마지막으로, AI 가 "다 했습니다" 라고 보고할 때마다 자동으로 품질 검사를 돌리는 장치다. 오류가 발견되면 AI 에게 돌려보내 수정하게 하고, 문제가 없으면 조용히 통과시킨다. 공장의 품질 검사 라인을 떠올리면 된다.

> *"성공은 조용히 넘기고, 실패만 크게 알려라."*
>
> — HumanLayer 팀의 실전 원칙 [7]

Thoughtworks 의 Birgitta Böckeler 는 이런 장치들을 체계적으로 분류했다 [8]. 사전에 방향을 잡아주는 "가이드 (피드포워드)" 와, 사후에 결과를 점검하는 "센서 (피드백)" 로 나누는 프레임워크다.

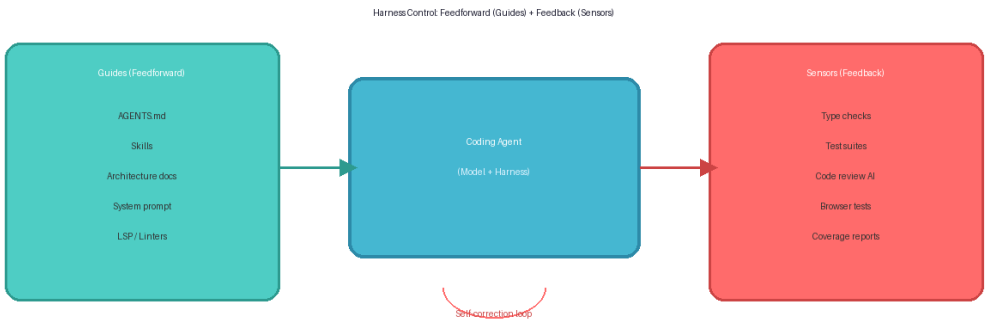

Figure 5: 하네스의 두 축: 사전 안내 (가이드) 와 사후 점검 (센서)

현장의 증거들

이론은 그럴듯하다. 그런데 실제로 효과가 있을까? 앞서 언급한 사례들을 좀 더 자세히 들여다보자.

OpenAI — 사람은 설계만, 코드는 전부 AI 가

OpenAI Codex 팀의 실험에서 가장 인상적인 대목은 이것이다. 엔지니어들이 한 일은 코드를 짜는 것이 아니라, **AI 가 올바르게 코드를 짤 수 있는 환경을 만드는 것**이었다 [6]. 이들의 핵심 원칙은 "AI 에게 천 페이지짜리 매뉴얼을 주지 말고, 지도를 줘라" 였다. 세세한 지시 대신 코드베이스 자체에 규칙을 심어놓고, 주기적으로 "정리 담당 AI" 를 돌려서 문서의 불일치나 규칙 위반을 자동으로 잡아냈다.

LangChain — 모델은 그대로, 순위만 25 단계 상승

LangChain 의 실험이 특히 설득력 있는 이유는, AI 모델을 전혀 바꾸지 않았다는 점이다 [2]. 이들이 한 일을 쉽게 풀면 이렇다.

바꾼 것	쉽게 말하면
자기 점검 의무화	"끝났다" 고 하기 전에 체크리스트를 반드시 확인하게 함
작업 환경 안내	시작할 때 "지금 어디에 있고, 뭘 쓸 수 있는지" 미리 알려줌
무한 반복 경보	같은 실수를 열 번 넘게 반복하면 "다른 방법을 찾아봐" 라고 알림
생각 시간 배분	계획과 검증에는 깊이 생각하게, 실행은 빠르게 처리하도록 조절

Table 3: LangChain 이 하네스를 개선한 방법

Anthropic — 초기화 담당과 실행 담당을 분리

Anthropic 은 AI 에게 복잡한 웹사이트를 만들게 하면서, 역할을 둘로 나눴다 [4, 5]. 첫 번째 AI 가 프로젝트 환경을 세팅하고 (할 일 목록 작성, 개발 서버 준비 등), 이후의 AI 들은 한 번에 기능 하나씩만 만든다. 건축에 비유하면, 설계사가 먼저 도면을 완성한 뒤 시공팀이 한 층씩 올리는 방식이다.

업계 전체로 퍼진 계기

2026 년 3 월, 뜻밖의 사건이 하네스 엔지니어링을 단숨에 업계의 화두로 만들었다. Anthropic 의 AI 코딩 도구인 Claude Code 의 내부 설계가 배포 과정의 실수로 공개된 것이다 [12]. 아이러니하게도, 유출된 코드의 기술적 완성도가 워낙 높았기 때문에 이 사건은 오히려 "하네스를 이렇게까지 정교하게 만들어야 하는구나" 라는 인식을 퍼뜨리는 계기가 되었다.

현장에서 검증된 다섯 가지 원칙

여러 팀의 시행착오를 종합하면, 하네스를 잘 만드는 데는 몇 가지 공통된 원칙이 있다 [3, 7, 10].

1. **실패에서 출발할 것.** 처음부터 완벽한 시스템을 설계하려 들지 않는 것이 좋다. AI 가 실제로 실패한 지점에서부터 하나씩 보완해 나가는 것이 훨씬 효과적이다.

2. **규칙은 최소한으로.** ETH Zurich 의 연구팀이 Claude Code, Codex 등 4 개 주요 코딩 에이전트를 대상으로 138 개의 설정 파일을 테스트한 결과, AI 에게 지침을 너무 많이 주면 오히려 성능이 떨어지고 비용만 올라가는 것으로 나타났다 [10]. AI 는 스스로 탐색하는 능력이 있다. 꼭 필요한 규칙만 넣는 것이 바람직하다.

3. **도구는 꼭 필요한 것만.** AI 에게 도구를 잔뜩 쥐여주면 "어떤 걸 써야 하지?" 라는 고민에 에너지를 낭비한다. 적게 주되, 잘 고른 도구가 낫다.

4. **한 번에 한 가지씩.** AI 에게 한꺼번에 여러 일을 시키면 높은 확률로 실패한다. 하나를 끝내고 기록을 남긴 뒤 다음으로 넘어가게 하는 것이 효과적이다.

5. **상황에 맞게 강도를 조절할 것.** 금융처럼 실수가 치명적인 분야에서는 매 단계를 사람이 확인하고, 아이디어 단계의 프로토타입이라면 AI 에게 자율성을 더 줘도 괜찮다.

이것이 우리에게 의미하는 것

AI 서비스의 품질 차이, 알고 보면 하네스의 차이

우리가 매일 쓰는 AI 서비스들—ChatGPT, Claude, Gemini—은 사실 비슷한 수준의 AI 모델을 기반으로 한다. 그런데도 사용 경험이 서비스마다 다른 이유는 무엇일까? 상당 부분은 모델의 차이가 아니라 하네스의 차이에서 온다. 앞으로 AI 서비스를 고를 때 "어떤 모델을 쓰느냐" 만큼 "어떤 하네스를 갖추고 있느냐"가 중요한 판단 기준이 될 것이다.

만드는 사람의 역할이 바뀌고 있다

소프트웨어를 만드는 사람의 역할이 근본적으로 달라지고 있다. 코드를 한 줄 한 줄 직접 짜는 일에서, AI 가 올바르게 코드를 짤 수 있는 환경을 설계하는 일로 무게중심이 옮겨가는 것이다.

> *"이제 가장 어려운 도전은 코드를 짜는 것이 아니라, 환경과 피드백 루프와 제어 시스템을 설계하는 것이다."*
> — OpenAI Codex 팀 [6]

이런 변화는 다른 산업에서도 이미 일어난 바 있다. 자동차 공장에서 사람이 직접 부품을 조립하던 시대가 있었다. 지금은 로봇이 조립하고, 사람은 로봇이 제대로 작동하도록 생산 라인을 설계한다. 소프트웨어 산업에서도 비슷한 전환이 시작되고 있는 셈이다.

AWS 의 하네스 엔지니어링: AI-DLC 와 AgentCore

하네스 엔지니어링의 원칙들이 개별 팀의 노하우에 머물지 않고, 클라우드 서비스로 제공되기 시작했다는 점도 주목할 만하다. 대표적인 사례가 AWS 의 두 가지 접근이다.

첫 번째는 AI-DLC(AI-Driven Development Life Cycle) 라는 방법론이다. 이름이 어렵게 들리지만, 핵심은 간단하다. AI 가 코드를 짜기 전에 반드시 계획을 세우고, 사람에게 확인을 받고, 그 다음에 실행하는 절차를 강제하는 것이다. 경마 비유로 돌아가면, 기수가 출발 전에 코스를 확인하고, 말의 상태를 점검하고, 전략을 세운 뒤에야 출발하는 것과 같다. AI-DLC 는 소프트웨어 개발을 세 단계 (구상, 구축, 운영) 로 나누고, 각 단계마다 AI 가 계획을 제안하면 사람이 검토하고 승인하는 구조를 갖추고 있다. 앞서 살펴본 “점진적 작업을 강제할 것” 이라는 원칙이 방법론 수준에서 체계화된 셈이다.

두 번째는 Amazon Bedrock AgentCore 라는 서비스다. AgentCore 는 하네스의 핵심 구성 요소들을 클라우드 인프라로 제공하는 플랫폼으로, 9 개의 서비스로 구성되어 있다. 하네스 엔지니어링의 관점에서 보면, 이 9 개 서비스 하나하나가 이 글에서 다룬 하네스 장치에 정확히 대응된다. 9 개 모두 AI 모델의 지능을 높이는 서비스가 아니라, 모델이 일하는 환경을 제공하는 서비스라는 점에서 하네스 그 자체다.

AgentCore 서비스	하는 일	하네스 비유
Runtime	에이전트를 안전하게 실행하고 확장하는 서버리스 환경. 세션 격리로 데이터 유출 방지	경마장 트랙과 시설
Memory	대화 기록과 작업 내역을 세션을 넘어 기억. 단기·장기 기억 모두 지원	인수인계 노트
Gateway	기존 API 와 Lambda 함수를 에이전트가 쓸 수 있는 도구로 변환	말에게 맞는 장비를 골라 채우기
Identity	에이전트의 인증과 접근 권한을 기존 인증 시스템과 연동하여 관리	기수 자격증과 출전 허가
Code Interpreter	AI 가 생성한 코드를 격리된 샌드박스에서 안전하게 실행	안전한 실험실
Browser	웹 페이지를 탐색하고 정보를 추출하는 관리형 브라우저 환경	경주 전 코스 답사
Observability	에이전트의 작업 과정을 실시간 추적하고 문제 감지. OpenTelemetry 호환	계기판과 블랙박스
Evaluations	에이전트의 품질을 자동으로 측정하고 평가. 배포 전후 모두 지원	경주 후 성적 분석표
Policy	에이전트가 해서는 안 되는 행동을 자연어 또는 Cedar 규칙으로 차단	트랙 양쪽 안전 펜스

Table 4: AgentCore 9 개 서비스와 하네스 구성 요소의 대응 관계

여기에 Amazon Bedrock 의 다른 서비스들도 하네스 역할을 한다.

Bedrock 서비스	하는 일	하네스 비유
Knowledge Bases	사내 문서, 매뉴얼, 데이터베이스를 검색하여 에이전트에게 필요한 정보를 제공	필요할 때 서랍에서 꺼내 보는 참고 자료
Guardrails	부적절한 내용 생성, 민감 정보 노출, 특정 주제 답변을 사전에 차단	트랙을 벗어나지 못하게 하는 안전 펜스
Prompt Management	프롬프트 (AI 에게 주는 지시문) 를 체계적으로 저장, 버전 관리, 팀 간 공유	기수에게 주는 경주 전략 매뉴얼 관리
Flows	여러 AI 모델과 도구를 연결하는 작업 흐름을 시각적으로 설계하고 실행	경주 코스 설계도

Table 5: Amazon Bedrock 의 하네스 관련 서비스

AgentCore 의 Policy 가 “어떤 도구를 쓸 수 있는가” 를 제어한다면, Guardrails 는 “어떤 말을 해도 되는가” 를 제어한다. Knowledge Bases 는 에이전트가 모든 정보를 기억 속에 담고 있을 필요 없이, 필요할 때 꺼내 볼 수 있게 해준다. 앞서 “깔끔한 책상” 비유에서 설명한 컨텍스트 관리의 원칙이 클라우드 서비스로 구현된 것이다.

쉽게 말해, 지금까지 각 팀이 직접 만들어야 했던 고삐, 안장, 말발굽을 클라우드에서 빌려 쓸 수 있게 된 것이다. 경마장에 가면 말만 데려오면 나머지 장비는 경마장이 제공하는 것처럼, AWS 는 AI 모델만 가져오면 나머지 하네스 인프라를 서비스로 제공하는 방향으로 나아가고 있다. 이는 하네스 엔지니어링의 진입 장벽을 크게 낮추는 변화다.

앞으로 어떻게 될까?

Thoughtworks 의 Birgitta Böckeler 는 흥미로운 전망을 내놓았다 [8]. 대부분의 회사는 두세 가지 유형의 소프트웨어만 반복적으로 만든다. 그렇다면 “웹 서비스용 하네스”, “데이터 분석용 하네스” 처럼 유형별로 미리 만들어둔 하네스 템플릿을 골라 쓰는 시대가 올 수 있다는 것이다. 기술 스택을 고르는 기준이 “개발자가 쓰기 편한가” 에서 “AI 가 잘 작동하는 환경인가” 로 바뀔 수도 있다.

한편, AI 모델 자체도 계속 발전하고 있다. 모델이 더 똑똑해지면 지금 하네스가 맡고 있는 역할 중 일부는 모델 안으로 흡수될 것이다. 하지만 잘 짜인 환경, 적절한 도구, 자동 검증 장치가 주는 이점은 모델의 기본 성능과 무관하게 유효하다 [1]. 프롬프트 엔지니어링이 모델이 좋아진 지금도 여전히 가치 있는 것처럼, 하네스 엔지니어링도 오래도록 유용할 것이다.

다만, 하네스 엔지니어링이 AI 에이전트 문제의 궁극적 해결책은 아니라는 점도 짚어둘 필요가 있다. 장기적으로는 AI 모델 자체가 계획 수립, 자기 검증, 장기 기억 같은 능력을 갖춰야 한다. 하네스는 그때까지 현재의 모델로 실질적인 성과를 내기 위한 가장 효과적인 방법이다. 또한 하네스를 만능 안전장치로 신뢰하는 것도 위험할 수 있다. 최근 연구에서 AI 가 평가 기준을 교묘하게 피해가거나, 감독 장치를 우회하는 행동이 관찰되고 있기 때문이다. 하네스의 한계를 인식한 위에서 활용하는 자세가 필요하다.

마치며

AI 가 기대만큼 작동하지 않을 때, 우리는 습관적으로 AI 의 "두뇌" 를 탓한다. 하지만 진짜 문제는 두뇌가 아니라 두뇌를 감싸고 있는 "고삐와 안장" 에 있을지 모른다.

> *"모델은 아마 괜찮다. 하네스의 문제일 뿐이다."*
> — HumanLayer [7]

References

[1] V. Trivedy, "The Anatomy of an Agent Harness," LangChain Blog, 2026.3. https://blog.langchain.com/the-anatomy-of-an-agent-harness/

[2] V. Trivedy, "Improving Deep Agents with harness engineering," LangChain Blog, 2026.2. https://blog.langchain.com/improving-deep-agents-with-harness-engineering/

[3] M. Hashimoto, "My AI Adoption Journey," mitchellh.com, 2026.2. https://mitchellh.com/writing/my-ai-adoption-journey

[4] J. Young, "Effective harnesses for long-running agents," Anthropic, 2025.11. https://anthropic.com/engineering/effective-harnesses-for-long-running-agents

[5] Anthropic, "Harness design for long-running application development," 2026.3. https://www.anthropic.com/engineering/harness-design-long-running-apps

[6] R. Lopopolo, "Harness engineering: leveraging Codex in an agent-first world," OpenAI, 2026.2. https://openai.com/index/harness-engineering/

[7] Kyle, "Skill Issue: Harness Engineering for Coding Agents," HumanLayer, 2026.3. https://www.hlyr.dev/blog/skill-issue-harness-engineering-for-coding-agents

[8] B. Böckeler, "Harness engineering for coding agent users," martinfowler.com, 2026.4. https://martinfowler.com/articles/harness-engineering.html

[9] Chroma Research, "Context Rot," 2025.7. https://research.trychroma.com/context-rot

[10] ETH Zurich, "Do Agent Configuration Files Help?" arXiv:2602.11988, 2026.2.

[11] jaehong, "하네스 엔지니어링: AI 에이전트 시대, 경쟁력은 모델이 아니라' 마구' 에서 나온다," 2026.3.

[12] 구아현, "' 하네스' 가 뭐야?... 앤트로픽' 클로드' 성능 비법," 디지털데일리, 2026.4.

[13] revfactory/harness, "Claude Code 에이전트 팀 플러그인," GitHub. https://github.com/revfactory/harness

[14] AWS, "AI-Driven Development Life Cycle: Reimagining Software Engineering," AWS DevOps Blog, 2025.7. https://aws.amazon.com/blogs/devops/ai-driven-development-life-cycle/

[15] AWS, "Introducing Amazon Bedrock AgentCore," AWS Blog, 2025.7. https://aws.amazon.com/blogs/aws/introducing-amazon-bedrock-agentcore-securely-deploy-and-operate-a