The State of Vibe Coding

바이브 코딩의 역사, 기술, 개념

박현우와 참여자들 · 서울대학교 데이터사이언스 특강 〈바이브 코딩〉

2026-09-14

목차

서문

이 책은 바이브 코딩(vibe coding)에 관한 생각들을 한곳에 모으는 시도다. 2025년 2월 Andrej Karpathy의 트윗에서 시작된 이 말은 이제 특정 코딩 방식을 넘어, AI와 함께 일하고 공부하는 방식 전체를 가리키는 렌즈가 되었다. 프롬프트 엔지니어링이 그 전사(前史)였다면, 루프 엔지니어링은 그 다음 장이다. 이 책은 그 흐름 전체를 다룬다.

이 책의 성격

살아 있는 책이다. 서울대학교 데이터사이언스 특강 〈바이브 코딩〉 수업에서 교수와 수강생이 GitHub에서 함께 쓴다. 학기마다 새 판을 내며, 그 사이에도 수시로 고친다. 완성된 책이 아니라 매년 갱신되는 스냅숏, 말 그대로 “the state of” vibe coding이다.

두 가지 형태로 나온다. 웹에서 읽는 판과 조판된 PDF 판이 같은 원고에서 함께 빌드된다.

두 언어로 나온다. 한국어판이 1차 원문이고, 영어판은 AI가 자동 번역한다. 다만 영어로 쓰는 것이 더 편한 저자가 맡은 장은 영어가 원문이 되고 한국어가 번역본이 된다. 각 장의 원문 언어는 장 머리의 메타데이터에 적혀 있다.

다루는 것

- **역사**: 프롬프트 엔지니어링에서 바이브 코딩으로, 다시 에이전틱 코딩과 루프 엔지니어링으로 이어지는 방법론 담론의 진화
- **기술**: 바이브 코딩이 열어주는 기회, 베스트 프랙티스, 글·코드·웹 애플리케이션·이미지·비디오 등 만들 수 있는 것들의 실전
- **원리와 한계**: 이 방식이 왜 작동하는지, 그리고 어디서 무너지는지. AI가 손댄 코드베이스가 블랙박스가 되어가는 문제를 포함한다
- **전망**: AI가 우리가 공부하고 일하는 방식을 어떻게 바꾸는지
- **개념 사전**: 바이브 코딩 주변에서 태어나고 진화하는 개념들의 사전. 부록으로 계속 자란다

기여하는 법

이 책 자체가 수업의 산출물이자 실습장이다. 기여 방법은 저장소의 CONTRIBUTING.md를 보라. 수업 운영 방식은 README와 실라버스에 있다.

> ℹ 현재 판
>
> 2026년판(작업 중). 2026년 2학기 수업과 함께 집필이 진행되고 있다.

제1부 · 바이브 코딩의 역사

1 전사(前史): 사람이 기계에게 말 걸어온 역사

바이브 코딩은 어느 날 갑자기 나타나지 않았다. “사람의 말로 프로그래밍한다”는 꿈은 컴퓨팅의 역사만큼 오래됐고, 그때마다 다른 이름을 달고 돌아왔다. 이 장은 그 반복의 역사에서 시작해, 바이브 코딩의 직접적 전사인 프롬프트 엔지니어링까지 온다.

1.1 자연어 프로그래밍이라는 오래된 꿈

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. “코드를 자연어로 쓰면 프로그래밍 진입 장벽이 사라진다”는 발상은 바이브 코딩이 처음이 아니다. COBOL이 내걸었던 “영어에 가까운 문법”부터 4세대 언어(4GL), CASE(Computer-Aided Software Engineering) 도구까지, 비슷한 약속이 수십 년에 걸쳐 반복되어 왔고 매번 완전히는 실현되지 못했다. 이 꼭지는 그 반복의 역사를 추적하고, 각 시도가 구체적으로 무엇을 약속했고 어디서 좌절했는지를 정리해야 한다. 바이브 코딩을 이 역

사의 최신 반복으로 볼지, 질적으로 다른 단절로 볼지 판단할 근거를 마련하는 것이 이 꼭지의 목표다. 1부 전체에서 가장 먼 기원을 다루는 자리다.

시드 질문

- COBOL 설계자들은 “영어처럼 읽히는 코드”로 실제 무엇을 노렸나, 그 목표는 어느 정도 달성되었나?
- 4GL은 어떤 문제를 풀겠다고 등장했고, 실제 채택은 어떤 궤적을 그렸나?
- CASE 도구의 약속과 1990년대의 실패 사례는 구체적으로 무엇이었나?
- 자연어 프로그래밍의 반복된 좌절에는 공통 원인이 있는가, 아니면 시도마다 다른 원인이었는가?
- 바이브 코딩이 “이번엔 다르다”고 주장할 근거가 있다면 무엇인가, 아니면 같은 패턴의 반복인가?

조사 포인터

- COBOL 원 설계 문서(CODASYL 관련 자료)와 Grace Hopper의 관련 발언 기록을 1차 출처에서 확인할 것
- 4GL과 CASE 도구의 역사를 다룬 소프트웨어공학 교과서(예: Pressman, Sommerville)의 서술을 대조할 것
- 자연어 프로그래밍의 반복된 실패를 다룬 학술 리뷰나 회고 글이 있는지 검색할 것

1.2 엔드유저 프로그래밍: 스프레드시트라는 선례

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 비프로그래머가 프로그래밍을 하는 시대는 이미 있었다. 스프레드시트에서 셀 수식을 쓰는 행위는 형식상 프로그래밍이고, 수십 년간 가장 널리 채택된 엔드유저 프로그래밍 사례다. 이 꼭지는 무엇이 스프레드시트를 성공시켰는지(즉시 피드백, 시각적 표현, 낮은 진입 장벽 등 후보들을 검증할 것)를 분석하고, 이것이 바이브 코딩에 주는 시사점을 도출해야 한다. 동시에 스프레드시트 오류가 실무에서 반복적으로 큰 사고를 낸 반례도 다뤄야 한다. 1부에서 이 꼭지는 “비전문가의 프로그래밍”이라는 주제를 프로그래밍 언어가 아니라 도구 형태의 관점에서 접근하는 대응축이다.

시드 질문

- 엔드유저 프로그래밍이라는 용어는 학계(HCI, 프로그래밍 언어 연구)에서 어떻게 정의되어 왔나?
- 스프레드시트가 다른 자연어·저코드 시도보다 훨씬 널리 채택된 이유는 무엇인가?
- 스프레드시트 오류로 인한 실제 사고 사례(재무, 학술 등)는 어떤 것이 있고 무엇을 시사하는가?
- 스프레드시트의 즉시 실행 피드백 구조는 바이브 코딩의 실행 확인 루프와 어떤 점에서 유사하고 다른가?
- 로우코드·노코드 플랫폼은 스프레드시트의 어떤 성공 요인을 계승했고 어떤 요인을 놓쳤나?

조사 포인터

- 엔드유저 프로그래밍 연구 문헌(HCI/PL 교차 분야, 예: Burnett 등의 연구)을 검색할 것
- 스프레드시트 오류 사례 모음(European Spreadsheet Risks Interest Group의 사고 아카이브 등)을 확인할 것
- VisiCalc, Lotus 1-2-3, Excel의 채택사를 다룬 컴퓨팅사 자료에서 정확한 연도와 수치를 확인할 것

1.3 프롬프트 엔지니어링의 짧은 전성기

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 바이브 코딩 직전 단계에는 프롬프트 엔지니어링이라는, 지금 보면 짧았던 전성기가 있었다. 거대언어모델에게 "무엇을 어떻게 말할 것인가"가 하나의 독립된 기술로 취급되던 시기, few-shot·chain-of-thought 같은 기법이 발견되고 정리되는 과정, 그리고 "프롬프트 엔지니어"라는 직함이 채용 시장에 등장했다가 빠르게 흐려지는 과정을 다뤄야 한다. 이 기법 중심 시대의 어떤 한계가 반복 대화와 실행 확인 중심인 바이브 코딩이라는 다음 단계를 불렀는지가 핵심 질문이다. 1부에서 이 꼭지는 2장(탄생)으로 넘어가기 직전의 마지막 전사를 맡는다.

시드 질문

- "프롬프트 엔지니어링"이라는 말은 언제, 어느 커뮤니티에서 처음 쓰이기 시작했나?
- few-shot, chain-of-thought, 역할 부여 같은 대표 기법은 각각 어떤 문제를 풀려고 나왔나?
- "좋은 프롬프트를 쓰는 기술"과 "모델과 반복 대화하며 만드는 기술"의 경계는 어디서, 왜 무너지기 시작했나?
- 프롬프트 엔지니어 채용 공고는 언제 정점을 찍었고 언제부터 줄었나?
- Karpathy의 Software 2.0(2017) 논의는 프롬프트 엔지니어링 시대를 예견한 것으로 읽을 수 있는가?

조사 포인터

- Anthropic과 OpenAI의 공식 프롬프트 엔지니어링 가이드 문서를 원전으로 확인할 것
- chain-of-thought 관련 초기 학술 논문(Wei et al. 등)의 발표 시점과 핵심 주장을 확인할 것
- 프롬프트 엔지니어 채용 공고의 등장과 소멸을 다룬 언론 기사를 수집할 것
- Karpathy, "Software 2.0" (2017)이 이 시기 담론과 어떻게 연결되는지 재독할 것

2 바이브 코딩의 탄생

2025년 2월 Andrej Karpathy의 트윗 하나가 이름 없던 작업 방식에 이름을 줬다. 이 장은 그 순간을 해부한다. 원문이 말한 것과 말하지 않은 것, 왜 하필 그 시점이었는지, 그리고 말이 퍼지며 뜻이 어떻게 흘러갔는지.

2.1 Karpathy의 트윗과 그 순간의 조건

⚠ 집필 상태

초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 2025년 2월, Andrej Karpathy가 올린 트윗 하나가 이후 한 해 동안 소프트웨어 업계에서 가장 많이 인용된 용어를 낳았다. 이 꼭지는 그 원문을 해부한다. 정확히 무엇을 정의했고 무엇을 정의하지 않았는지, 트윗 작성 시점의 기술적 조건(모델 세대, 에이전트형 코딩 도구의 성숙도, 개발자 커뮤니티의 분위기)이 이 정의의 확산에 어떻게 기여했는지, 왜 하필 그 시점이었는지를 다뤄야 한다. 2장 전체의 출발점이 되는 꼭지다.

> There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists.
>
> — Andrej Karpathy, X, 2025년 2월

시드 질문

- Karpathy의 원문이 실제로 정의한 범위는 무엇이고, 이후 널리 쓰인 뜻과 어떻게 다른가?
- 트윗 발화 시점에 이용 가능했던 코딩 에이전트·도구는 무엇이었나, 그 성숙도가 이 개념의 수용에 어떤 역할을 했나?
- Karpathy는 1년 뒤 회고 트윗에서 이 트윗을 "샤워하다 떠오른 생각을 던진 트윗"이라 불렀다. 이 자기 평가는 실제 영향력과 어떻게 어긋나는가?
- 트윗이 확산되던 초기 며칠간 어떤 반응(환영, 냉소, 반박)이 지배적이었나?

조사 포인터

- 원 트윗(https://x.com/karpathy/status/1886192184808149383)과 1주년 회고 트윗(https://x.com/karpathy/status/2019137879310836075)의 전문과 주요 인용 반응을 수집할 것
- 트윗 발화 시점 전후 주요 코딩 에이전트 도구의 출시·업데이트 이력을 확인할 것
- 위키피디아 개관(https://en.wikipedia.org/wiki/Vibe_coding)에서 초기 수용 관련 서술을 확인할 것

2.2 용어의 확산과 뜻의 표류

⚠ 집필 상태

초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 하나의 트윗에서 시작된 말이 몇 달 사이 원래 정의보다 훨씬 넓은 뜻으로 쓰이게 되었다. 이 꼭지는 그 확산과 뜻의 표류를 다룬다. Simon Willison이 "모든 AI 보조 프로그래밍이 바이브 코딩은 아니다"라고 선을 그은 이유, 좁은 뜻(코드를 읽지 않고 전적으로 맡기는 것)과 넓은 뜻(AI 도구를 쓰는 코딩 전반)이 어떻게 동시에 유통되었는지, Collins 사전이 올해의 단어로 선정하며 이 확산을 어떻게 공인했는지를 다뤄야 한다. 2장에서 §2.1이 발화의 순간을 다룬다면 이 꼭지는 그 이후의 유통 과정을 다룬다.

시드 질문

- Simon Willison이 그은 경계선의 구체적 기준은 무엇인가, 그는 어떤 사례를 "진짜 바이브 코딩"과 "AI 보조 프로그래밍"으로 나눴나?
- 좁은 뜻과 넓은 뜻이 공존하는 상태는 실무 현장에서 어떤 혼란("이거 바이브 코딩이야?" 류의 논쟁)을 낳았나?
- Collins 사전의 선정 사유문은 이 말을 어떻게 정의했나, 그 정의는 원 트윗과 Willison의 구분 중 어느 쪽에 가까운가?
- 시기별 스냅샷(트윗 직후, 몇 달 뒤, 사전 등재 시점)을 비교하면 뜻의 표류에 어떤 패턴이 보이는가?

조사 포인터

- Simon Willison, "Not all AI-assisted programming is vibe coding" (2025-03-19) 전문을 읽을 것
- Collins Word of the Year 2025 발표문(https://www.collinsdictionary.com/us/woty)과 보도(https://www.cnn.com/2025/11/06/tech/vibe-coding-collins-word-year-scli-intl)의 정의 문구를 대조할 것
- 위키피디아 개관(https://en.wikipedia.org/wiki/Vibe_coding)의 편집 이력에서 정의 변화를 추적할 수 있는지 확인할 것

2.3 도구의 계보: 자동완성에서 에이전트까지

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 이 꼭지는 개념이 아니라 도구의 계보를 다룬다. 2021년 공개된 GitHub Copilot의 자동완성형 코딩 지원에서 출발해, 챗 인터페이스 기반 코딩 도구를 거쳐 에이전트형 CLI·IDE 도구로 이어지는 흐름을 정리해야 한다. 핵심 주장은 도구의 형태(자동완성 대 대화 대 자율 실행)가 그 시대의 코딩 방법론을 규정해왔다는 것이다. 바이브 코딩이라는 방법론이 그것을 가능케 한 도구 형태 없이는 나오지 않았을 것이라는 인과 관계를 검증해야 한다. 2장에서 이 꼭지는 개념사(§2.1, §2.2)를 도구사로 보완하는 자리다.

시드 질문

- GitHub Copilot(2021년 프리뷰 공개)은 자동완성이라는 형태를 왜 택했나, 그 형태가 이후 어떤 도구들의 기본값이 되었나?
- 자동완성형에서 챗 기반 코딩 지원(코드 설명, 생성 요청)으로의 전환은 어떤 도구, 어떤 시점에 일어났나?
- 챗 기반에서 에이전트형(파일 시스템 접근, 명령 실행, 자율 루프)으로의 전환은 어떤 기술적 전제를 필요로 했나?
- 각 도구 형태의 전환기마다 개발자 커뮤니티의 반응(환영, 저항)은 어떠했나?
- 도구 형태와 방법론 중 어느 쪽이 다른 쪽을 이끌었나, 아니면 상호 결정적이었나?

조사 포인터

- GitHub Copilot 공개 발표(2021년, 정확한 날짜와 프리뷰·GA 구분을 확인할 것)와 이후 주요 업데이트 이력을 확인할 것

- 대표 에이전트형 코딩 CLI·IDE 도구(Cursor, Windsurf, Claude Code, Devin 등)의 출시 연대기를 정리할 것
- Anthropic, “Building effective agents” (2024-12)에서 도구 형태와 자율성 관련 논의를 확인할 것

3 담론의 진화

바이브 코딩이라는 말이 자리 잡자마자 그것을 다듬거나 넘어서려는 개념들이 쏟아졌다. CHOP, 컨텍스트 엔지니어링, 스펙 주도 개발, 루프 엔지니어링, 바이브 엔지니어링. 이 장은 이 개념들의 계보를 그리고, 개념이 바뀔 때마다 무엇이 병목이었는지를 추적한다. 마지막 절은 이 개념 시장 자체를 분석 대상으로 삼는다.

3.1 CHOP과 에이전틱 코딩: 자율성의 스펙트럼

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 이 꼭지는 Steve Yegge가 바이브 코딩이라는 말이 퍼지기 전부터 쓰던 CHOP(chat-oriented programming)이라는 개념에서 출발해, 이후 정리된 에이전트 자율성 스펙트럼(어디까지 사람이 관여하고 어디부터 에이전트가 자율적으로 움직이는가)까지를 다룬다. CHOP과 바이브 코딩, 에이전틱 코딩이 서로 어떻게 구분되고 겹치는지가 핵심 질문이다. 3장은 바이브 코딩이라는 용어가 자리 잡은 뒤 쏟아진 후속 개념들을 다루는데, 이 꼭지는 그중 가장 먼저 나온 축을 맡는다.

시드 질문

- Steve Yegge는 CHOP을 언제, 어떤 맥락에서 처음 제안했나, 바이브 코딩과의 선후 관계는 어떠한가?
- CHOP과 바이브 코딩은 정의상 어떻게 다른가, 실무자들은 두 말을 구분해서 쓰는가 섞어 쓰는가?
- “에이전틱 코딩”이라는 말은 CHOP, 바이브 코딩과 각각 어떤 관계에 있나?
- 에이전트 자율성을 단계로 나누는 시도(자동완성에서 자율 실행, 자율 검증까지)는 누가, 어떤 기준으로 정리했나?
- Gene Kim과 Steve Yegge가 함께 쓴 단행본은 이 개념들을 어떻게 종합했나?

조사 포인터

- Steve Yegge, O’Reilly 팟캐스트 인터뷰에서 CHOP 관련 발언을 확인할 것
- Gene Kim·Steve Yegge 공저 단행본 “Vibe Coding”의 정확한 출간 시점과 핵심 주장을 확인할 것
- 에이전트 자율성 단계론을 다룬 실무 블로그 글(레벨 표를 제시한 사례)을 찾아 비교할 것

3.2 컨텍스트 엔지니어링

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 이 꼭지는 컨텍스트 엔지니어링이라는 개념의 명명과 정식화를 다룬다. Shopify CEO Tobi Lütke가 2025년 6월 트윗으로 이 말을 붙였고 Karpathy가 이를 승인하며 정식화에 힘을 실었다. 핵심 주장은 프롬프트 하나를 잘 쓰는 기술에서, 에이전트가 매 턴 무엇을 보고 무엇을 못 보는지, 즉 컨텍스트 창 구성 전체를 설계하는 기술로 병목이 옮겨갔다는 것이다. 3장에서 이 꼭지는 프롬프트 엔지니어링(1부)의 직접적 후계 개념을 다룬다.

시드 질문

- Tobi Lütke는 정확히 어떤 트윗에서 이 말을 제안했고, 어떤 문제의식에서 나왔나?
- Karpathy는 어떤 방식으로 이 명명에 동의했나, 그의 승인이 용어 확산에 어떤 역할을 했나?
- "프롬프트 엔지니어링"과 "컨텍스트 엔지니어링"의 실질적 차이는 무엇인가, 겹치는 부분은 어디인가?
- Simon Willison과 Phil Schmid는 각각 이 개념을 어떻게 정리했나, 두 정리 사이에 강조점의 차이가 있는가?
- 컨텍스트 엔지니어링이라는 개념이 실제로 바꾼 개발 관행(CLAUDE.md·AGENTS.md류 문서, 컨텍스트 창 예산 관리 등)은 무엇인가?

조사 포인터

- Tobi Lütke의 명명 트윗(2025-06-19)과 Karpathy의 승인 트윗을 확인할 것
- Simon Willison, 정리 글 (2025-06-27)을 읽을 것
- Phil Schmid, 정리 글을 읽을 것

3.3 스펙 주도 개발

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 이 꼭지는 스펙 주도 개발을 바이브 코딩에 대한 반작용으로 다룬다. GitHub이 2025년 9월 공개한 Spec Kit은 Spec, Plan, Tasks, Implement 네 단계를 명시적으로 밟게 하는 도구다. 핵심 질문은 "코드가 존재한다는 사실조차 잊어버리라"는 바이브 코딩의 태도와, 명세를 먼저 쓰고 그것을 코드보다 우선시하는 스펙 주도 개발의 태도가 어떤 긴장 관계에 있는가이다. 이 대립이 폭포수와 애자일 논쟁의 재상연인지(4부 §10.1에서 다시 다룸)도 짚어야 한다. 3장에서 이 꼭지는 바이브 코딩에 대한 최초의 체계적 반작용을 맡는다.

시드 질문

- GitHub Spec Kit이 명시한 네 단계(Spec, Plan, Tasks, Implement) 각각은 어떤 문제를 겨냥해 설계되었나?
- 스펙 주도 개발을 표방하는 다른 도구·워크플로가 있는가, 있다면 접근 방식이 어떻게 다른가?
- 스펙 주도 개발은 바이브 코딩을 대체하려는 시도인가 보완하려는 시도인가, 실무자들의 실제 사용 패턴은 어느 쪽에 가까운가?
- 이 대립을 폭포수 대 애자일 논쟁(Royce 1970, Agile Manifesto 2001)과 겹쳐 보면 어떤 유사점과 차이점이 드러나는가?
- 스펙 우선 접근이 실제로 코드 품질이나 검증 가능성을 높였다는 증거가 있는가?

조사 포인터

- GitHub, “Spec-driven development with AI” (2025-09) 공식 발표문을 읽을 것
- Winston Royce, “Managing the Development of Large Software Systems” (1970)과 Agile Manifesto (2001)를 원전으로 대조할 것
- Spec Kit 외 유사 워크플로(다른 벤더의 plan-mode류 기능)를 조사해 비교할 것

3.4 루프 엔지니어링과 하네스 엔지니어링

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 이 꼭지는 2026년에 정리된 두 후계 개념, 루프 엔지니어링과 하네스 엔지니어링을 다룬다. 루프 엔지니어링은 Addy Osmani가 2026년 6월 무렵 정리하며 확산시킨 개념으로, 에이전트를 반복시켜 스스로 일을 찾고 검증하게 만드는 설계를 다룬다. 하네스 엔지니어링은 “에이전트는 모델과 하네스의 합”이라는 정식화에서 나온 개념으로, 모델 자체보다 그것을 둘러싼 도구·프롬프트·검증 장치의 설계를 강조한다. 두 개념이 프롬프트 엔지니어링, 컨텍스트 엔지니어링과 함께 병목 이동의 연속선 어디에 놓이는지가 핵심 질문이다.

시드 질문

- Addy Osmani가 정리한 루프 엔지니어링의 핵심 주장은 무엇인가, 그가 인용한 실무 사례는 어떤 것들인가?
- “루프”라는 단위(에이전트가 스스로 일을 찾고, 하고, 검증하고, 기억하는 순환)는 이전의 대화 턴 단위 상호작용과 무엇이 다른가?
- “Agent = Model + Harness”라는 정식화는 누가, 언제 처음 제안했나, 이 정식화가 강조하는 것은 무엇인가?
- 하네스 엔지니어링이라는 말은 실무에서 컨텍스트 엔지니어링과 어떻게 구분되어 쓰이는가, 겹쳐 쓰이는가?
- 루프 엔지니어링과 하네스 엔지니어링 중 어느 쪽이 더 넓은 개념을 가리키는가, 아니면 서로 다른 축인가?

조사 포인터

- adtmag 보도, “Loop Engineering Emerges as Developers Put AI Coding Agents on Repeat” (2026-07-01)에서 Osmani 원 정리의 출처를 역추적할 것
- Faros AI, “harness engineering” 블로그 글을 읽을 것
- Anthropic, “Building effective agents” (2024-12)와의 계보 관계를 확인할 것

3.5 바이브 엔지니어링과 그 반작용들

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 이 꼭지는 Simon Willison이 제안한 바이브 엔지니어링 개념과 그 주변의 재정의 시도들을 다룬다. 바이브 코딩이 “코드를 읽지 않고 전적으로 맡기는” 태도로 널리 이해되자, 이에 대한 반작용으로 빠른 속도를 유지하되 결과에 책임을 지는 태도를 가리키는 말들이 나왔다. 이 꼭지는 바이브 엔지니어링이 바이브 코딩과 정확히 무엇이 다른지, 이런 재정의 시도들

이 실무에서 실제로 구분되어 쓰이는지 검증해야 한다. 3장에서 이 꼭지는 §3.3(스펙 주도 개발)과 함께 바이브 코딩에 대한 반작용 축을 이룬다.

시드 질문

- Simon Willison은 바이브 엔지니어링을 어떤 문제의식에서, 정확히 언제 제안했나?
- 바이브 엔지니어링과 바이브 코딩의 경계는 구체적으로 어디에 그어지나?
- "책임지는 가속"류의 재정의를 시도한 다른 필자나 개념이 있는가, 있다면 Willison의 정의와 비교하면 어떤가?
- 이런 재정의 시도들은 실무 현장에서 실제로 구분되어 쓰이는가, 아니면 바이브 코딩이라는 말로 뭉뚱그려지는가?
- 바이브 엔지니어링이라는 말 자체는 이후 얼마나 확산되었나, 아니면 소수 담론에 머물렀나?

조사 포인터

- Simon Willison, "vibe engineering" 원문 전체와 정확한 발행일을 확인할 것
- Willison의 이전 글 "Not all AI-assisted programming is vibe coding" (2025-03-19)과의 논지 연속성을 대조할 것
- 바이브 엔지니어링을 인용하거나 반박한 후속 글들을 찾아 확산 정도를 가늠할 것

3.6 개념 시장의 관찰: 다음 유행어는 어떻게 만들어지는가

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 이 꼭지는 3장 전체를 메타 분석하는 자리다. 프롬프트 엔지니어링에서 시작해 바이브 코딩, 에이전틱 코딩, 컨텍스트 엔지니어링, 스펙 주도 개발, 루프 엔지니어링, 하네스 엔지니어링, 바이브 엔지니어링까지 이어진 용어들의 탄생, 확산, 소멸(또는 정착) 패턴 자체를 분석 대상으로 삼는다. 핵심 가설은 각 개념의 교체가 실제 병목의 이동(모델 능력 부족에서 컨텍스트 관리 부족, 검증 부족, 루프·시스템 설계 부족으로)을 반영한다는 것이다. 이 가설이 맞다면 다음 유행어가 가리킬 병목도 예측해볼 수 있다. 1부의 마지막 꼭지로서 2부(기술)로 넘어가기 전 개념사 전체를 정리하는 역할을 한다.

시드 질문

- 이 장에서 다룬 개념들을 시간순으로 배열하면 각 개념의 수명(등장부터 다음 개념에 자리를 내주기까지)은 얼마나 되나?
- 각 개념 교체 시점마다 실제로 이동한 병목은 무엇이었다고 주장할 수 있는가, 그 주장의 근거는 무엇인가?
- 이 담론들은 실제 개발 관행을 바꿨는가, 아니면 기존 관행에 새 이름을 붙인 것에 가까운가?
- 개념이 확산되는 경로(트윗 하나, 유력 인사의 승인, 언론 보도, 사전 등재)에 공통 패턴이 있는가?
- 이 분석이 맞다면, 다음 병목과 다음 유행어는 어느 방향일 것으로 예측할 수 있는가?

조사 포인터

- 이 장의 다른 꼭지(§3.1~§3.5)에서 확정한 각 개념의 정확한 등장 시점을 표로 정리해 시간순으로 비교할 것

- 용어 확산 메커니즘 자체를 다룬 언어학·밈 확산 연구가 있는지 검색할 것
- Meir Lehman의 소프트웨어 진화 법칙(1970년대)이 개념 자체의 진화에도 유비될 수 있는지 검토할 것

제2부 · 바이브 코딩의 기술

4 기회와 포텐셜

바이브 코딩이 여는 것은 “코딩을 빨리 하는 것”이 아니라 만들 수 있는 사람과 만들 수 있는 것의 범위다. 이 장은 그 기회를 사람(누구에게 열렸나), 물건(무엇이 새로 가능한가), 경제(비용 구조가 어떻게 바뀌나), 그리고 연구 현장의 순서로 짚는다.

4.1 누구에게 어떤 문이 열렸나

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 바이브 코딩이 가장 먼저 바꾼 것은 누가 소프트웨어를 만들 수 있는가라는 경계다. 비전공자, 도메인 전문가, 1인 창업자, 연구자 각각에게 이 문이 얼마나 넓게 열렸는지, 그 문턱 뒤에 숨어 있던 새로운 장벽은 무엇인지가 이 꼭지의 주제다. “이제 누구나 만든다”는 서술은 성공 사례에서 나온 것이 많고, 실패 사례와 중도 포기 사례는 상대적으로 덜 기록된다. 이 꼭지는 두 흐름을 균형 있게 모아 누구에게 실제로 무엇이 열렸는지 구체적으로 그린다. 4장(기회와 포텐셜)의 도입이며, 뒤이은 §4.2~4.4가 각론을 다룬다.

시드 질문

- 비전공자가 바이브 코딩으로 처음부터 끝까지 완성한 소프트웨어 사례는 무엇이 있고, 어디서 막혔나?
- 도메인 전문가(의료, 법률, 교육 등)가 자기 분야 문제를 직접 코딩으로 푼 사례는 어떤 패턴을 보이나?
- 1인 창업자에게 바이브 코딩은 실제로 팀 규모를 줄여주었나, 아니면 다른 병목(마케팅, 운영)으로 문제를 옮겼을 뿐인가?
- “이제 누구나 만든다”는 주장에 대한 회의적 반론들은 무엇을 근거로 드나?
- 자신의 제작 경험을 이 꼭지의 사례로 기록하라. 처음 바이브 코딩을 시도했을 때 어떤 배경지식이 없었고, 무엇이 실제로 그 부재를 메워주었으며 무엇이 메워주지 못했는가.

조사 포인터

- 비개발자의 제작 후기(블로그, 커뮤니티, Product Hunt류)를 성공과 실패 양쪽에서 수집
- 1인 창업자, 인디 해커 커뮤니티(Indie Hackers 등)의 사례 인터뷰
- 수강생 자신과 동료의 첫 제작 경험을 인터뷰해 사례로 축적

4.2 1인용 소프트웨어, 일회용 소프트웨어

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 소프트웨어를 만드는 한계비용이 낮아지면 나타나는 새로운 카테고리가 있다. 딱 한 사람, 혹은 한 번의 작업을 위해 만들고 버리는 소프트웨어다. 이 꼭지는 이런 개인용, 일회용 도구가 실제로 늘고 있는지, 늘고 있다면 어떤 형태로 나타나는지를 다룬다. 기존 소프트웨어 산업은 재사용과 규모의 경제를 전제로 설계되어 왔는데, 그 전제가 흔들릴 때 무엇이 남고 무엇이 새로 생기는지가 핵심 질문이다. §4.1이 연 문을 통해 들어온 사람들이 실제로 무엇을 만드는지에 대한 답이기도 하다.

시드 질문

- 개인용, 일회용 소프트웨어를 가리키는 용어와 그 논의는 어디서 찾을 수 있나?
- 일회용 소프트웨어(한 번 쓰고 버리는 스크립트, 도구)의 실제 사례는 어떤 영역에서 가장 많이 나타나나?
- 맞춤 소프트웨어의 한계비용이 낮아지면 기존 SaaS, 범용 소프트웨어 시장은 어떻게 반응하나?
- 일회용 소프트웨어에는 유지보수, 보안, 문서화가 얼마나 필요한가? 아예 필요 없다는 주장은 타당한가?
- 자신이 만든 일회용 혹은 개인용 도구를 사례로 기록하라. 무엇을 위해 만들었고, 만든 뒤에는 어떻게 되었는가(계속 쓰는지, 버렸는지, 남에게 공유했는지).

조사 포인터

- “software for one”, “disposable software” 등 키워드로 실무자 블로그와 강연 검색
- 개인 도구 공유 커뮤니티(예: 해커뉴스 Show HN류)에서 유사 사례 수집
- 수강생 각자가 만든 개인용 도구를 설문이나 인터뷰로 모아 정리

4.3 프로토타이핑의 경제학

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 아이디어를 떠올린 순간부터 동작하는 데모를 보여줄 수 있는 순간까지의 거리가 짧아지면, 그 사이에 있던 여러 조직적 관행(기획서, 승인 절차, 목업 도구)의 존재 이유가 흔들린다. 이 꼭지는 프로토타이핑 속도의 변화가 스타트업과 기업 현장에서 실제로 어떤 관행을 바꾸었는지, 바뀌지 않고 남은 것은 무엇인지를 다룬다. 속도만 빨라진 것인지, 프로토타입의 성격 자체가 바뀐 것인지도 구분해서 살펴야 한다. §4.2의 개인용 소프트웨어 논의를 조직과 창업 맥락으로 확장한다.

시드 질문

- 스타트업의 MVP(최소 기능 제품) 제작 기간은 실제로 얼마나 단축되었나? 정량적 근거는 무엇인가?
- 기업 내부에서 프로토타입을 승인받는 절차는 제작 속도가 빨라진 만큼 함께 짧아졌나, 아니면 병목이 다른 곳으로 옮겨갔나?
- “빠른 프로토타입”과 “프로덕션에 바로 배포된 프로토타입”의 경계가 흐려지면서 생기는 문제는 무엇인가?
- 투자 유치나 내부 설득 과정에서 프로토타입의 역할은 어떻게 달라졌나?

- 자신이 아이디어에서 동작하는 데모까지 걸린 시간을 실제로 측정해 사례로 기록하라. 무엇이 병목이었는가(구상, 제작, 검증 중 어디였나).

조사 포인터

- 스타트업 액셀러레이터, VC의 공개 자료에서 MVP 제작 기간 관련 언급 수집(수치는 1차 출처 확인)
- 기업 내부 해커톤, 이노베이션랩 사례 보도
- 수강생 자신의 프로젝트 제작 시간을 기록해 비교 데이터로 사용

4.4 연구자의 바이브 코딩

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 연구자에게 바이브 코딩은 조금 다른 조건에서 작동한다. 데이터 파이프라인, 시각화, 시뮬레이션 코드는 대부분 한 번 쓰고 버리는 것이 아니라 논문 심사와 재현성 검증을 통과해야 한다. 이 꼭지는 연구 현장에서 바이브 코딩이 실제로 어떻게 쓰이고 있는지, “동작하면 된다”는 바이브 코딩의 관행이 “재현 가능해야 한다”는 연구의 요구와 어디서 충돌하는지를 다룬다. 이 책이 서울대 데이터사이언스 수업의 산물이라는 점에서, 이 꼭지는 저자 집단의 실제 작업 방식과도 맞닿아 있다.

시드 질문

- 연구자들은 데이터 전처리, 시각화, 시뮬레이션 코드 작성에 AI 코딩 도구를 어떻게 쓰고 있나? 분야별 차이가 있나?
- 재현성 요구(코드 공개, 환경 고정, 시드 고정)와 바이브 코딩의 빠른 반복 관행은 실제로 충돌하는가, 아니면 도구가 이를 자동으로 해결해주는가?
- 통계, 계량경제 코드에서 AI가 만든 결과를 검증하는 절차는 무엇이 다르게 필요한가?
- 학술지와 학회의 AI 사용 공개 정책은 연구 코드 작성에 어떤 제약을 두고 있나?
- 자신의 연구나 수업 과제에서 데이터 파이프라인, 시각화, 시뮬레이션 중 하나를 바이브 코딩으로 만든 경험을 사례로 기록하라. 결과를 어떻게 검증했는가.

조사 포인터

- 계량경제, 통계 분야 연구자들의 AI 코딩 도구 사용기(블로그, X 스레드)
- 재현성 관련 학술 논의(재현성 위기 관련 문헌과 AI 시대의 재조명)
- 수강생 각자의 연구, 과제 파이프라인 제작 경험을 사례로 정리

5 베스트 프랙티스

바이브 코딩을 잘하는 것과 못하는 것의 차이는 크고, 그 차이는 대부분 배울 수 있다. 이 장은 2025년 이후 실무자들이 쌓아온 관행을 도구가 바뀌어도 남을 원칙 중심으로 정리한다. 루프의 크기, 계획, 컨텍스트, 검증, 복구, 그리고 멈춰야 할 때.

5.1 작게 시작하고 자주 확인하기

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 바이브 코딩 실무자들 사이에서 가장 자주 반복되는 조언은 한 번에 너무 많이 시키지 말라는 것이다. 요청의 단위를 작게 쪼개고 결과를 자주 확인할수록 최종 결과물의 품질이 높아진다는 경험칙이다. 이 꼭지는 이 원칙이 왜 성립하는지, 어느 정도 크기가 적정한지, 이 원칙과 "AI에게 맡기고 결과만 본다"는 바이브 코딩의 본래 정의가 어떻게 공존하는지를 다룬다. 5장(베스트 프랙티스)의 첫 원칙이며, 뒤따르는 §5.2와 §5.4의 전제가 된다.

시드 질문

- "작은 단계"의 적정 크기를 정량적으로 제시한 실무자나 도구 문서가 있는가? (예: 파일 수, 변경 줄 수, 소요 시간)
- 반복 루프를 짧게 가져가는 것과 길게 맡기고 기다리는 것(에이전트에게 긴 태스크를 통으로 위임) 사이의 트레이드오프는 무엇인가?
- "결과를 자주 확인한다"는 것은 구체적으로 무엇을 확인한다는 뜻인가? 코드를 읽는 것과 실행 결과만 보는 것은 다른 원칙인가?
- 이 원칙이 실패하는 경우, 작게 쪼개도 품질이 나빠지는 사례는 어떤 조건에서 나타나나?
- 자신의 바이브 코딩 세션에서 단계를 크게 가져갔을 때와 작게 가져갔을 때를 비교한 경험을 사례로 기록하라.

조사 포인터

- Claude Code, Cursor 등 도구 공식 문서의 권장 워크플로 서술
- 실무자 블로그의 AI와 코딩하는 법 류 글에서 반복 루프 크기 관련 서술 수집
- 수강생 자신의 세션 로그가 있다면 단계 크기와 결과 품질의 관계를 직접 비교

5.2 계획 먼저: 스펙과 계획 문서로 시작하기

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 바이브라는 말의 원래 뜻은 계획 없이 감으로 간다는 것에 가깝지만, 실무에서 자리 잡은 관행은 정반대로 움직였다. 코드를 쓰기 전에 AI에게 먼저 계획 문서나 스펙을 작성하게 하고, 사람이 검토한 뒤에야 실행 단계로 넘어가는 방식이다. 이 꼭지는 이런 계획 우선 워크플로가 어떻게 자리 잡았는지, plan mode 같은 도구 차원의 장치가 무엇을 해결하려 했는지, 이 관행이 §3.3에서 다룬 스펙 주도 개발과 어떻게 다르거나 겹치는지를 다룬다.

시드 질문

- 계획 우선 워크플로를 명시적으로 지원하는 도구 기능(plan mode, 계획 승인 단계 등)은 언제부터 어떤 도구에 등장했나?
- 계획 문서의 적정 상세도는 어느 정도인가? 너무 상세하면 무엇을 잃고, 너무 성글면 무엇이 문제인가?

- 계획 단계에서 사람이 개입해 방향을 바꾼 경험은 실무자들 사이에서 어떻게 보고되나?
- §3.3의 스펙 주도 개발과 이 꼭지의 계획 우선 관행은 같은 것인가, 격식의 정도에서 다른 것인가?
- 자신이 계획 문서를 먼저 쓰게 하고 작업한 경우와 바로 코드를 시킨 경우를 비교한 경험을 사례로 기록하라.

조사 포인터

- Claude Code plan mode, 기타 도구의 계획 및 승인 기능 공식 문서
- 실무자들이 공개한 계획 문서 템플릿이나 프롬프트 예시
- 수강생 자신의 프로젝트에서 계획 문서 유무에 따른 결과 차이를 비교

5.3 컨텍스트 관리의 기술

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 에이전트가 무엇을 알고 무엇을 모르는 상태에서 작업을 시작하느냐가 결과를 크게 좌우한다는 인식은 §3.2에서 다룬 컨텍스트 엔지니어링 개념의 실무판이다. 이 꼭지는 CLAUDE.md, AGENTS.md 같은 프로젝트 지침 파일, 컨텍스트 창의 예산을 어떻게 배분할지, 무엇을 보여주고 무엇을 의도적으로 숨길지에 대한 실무 기술을 다룬다. 이론(§3.2)과 실전(5장) 사이의 다리 역할을 한다.

시드 질문

- CLAUDE.md, AGENTS.md류 프로젝트 지침 파일은 실제로 어떤 내용을 담을 때 효과적이라고 보고되나?
- 컨텍스트 창이 커질수록(대용량 토큰 모델 등) 컨텍스트 관리 기술 자체의 필요성은 줄어드는가, 다른 형태로 남는가?
- “무엇을 숨길 것인가”에 대한 실무자들의 구체적 조언은 무엇인가? (예: 불필요한 파일 제외, 오래된 대화 요약)
- 컨텍스트 관리 실패가 실제 작업 실패로 이어진 사례는 어떻게 보고되나?
- 자신이 프로젝트 지침 파일(CLAUDE.md류)을 작성하고 개선해온 경험을 사례로 기록하라. 무엇을 추가했더니 결과가 나아졌는가.

조사 포인터

- Claude Code, Cursor, GitHub Copilot 등의 프로젝트 지침 파일 공식 문서
- 컨텍스트 엔지니어링 실무 가이드(§3.2 조사 결과와 교차 참조)
- 이 저장소(vibecoding)의 CLAUDE.md 자체를 사례로 분석

5.4 검증 가능한 단위 만들기

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. “동작한다”는 말은 생각보다 헐거운 기준이다. 한 번 실행해서 에러가 안 났다는 뜻일 수도 있고, 모든 테스트를 통과했다는 뜻일 수도 있다. 이 꼭지는 바이브 코딩 실무에서 이 헐거운 기준을 좁혀가는 기술들, 테스트 작성, 실행 결과의 직접 확인, 여러 방법으로 같은 결과를 교차 검증하는 삼각검증 관행을 다룬다. §8.3에서 다룰 검증 비대칭 개념의 실무적 대응이며, 코드는 검증하기 쉽다는 이론이 실제로 어떤 기술로 구현되는지를 보여준다.

시드 질문

- 바이브 코딩 세션에서 테스트를 AI가 직접 작성하게 하는 관행은 신뢰할 만한가? 테스트 자체가 부실하게 작성될 위험은 어떻게 다뤄지나?
- 실행해서 확인한다와 테스트를 통과시킨다는 검증 강도가 다른 두 기준인데, 실무자들은 어떤 조합을 권장하나?
- 삼각검증(같은 결과를 다른 방법으로 재확인)이 실제로 쓰이는 구체적 사례는 무엇인가?
- 검증 단위를 지나치게 잘게 쪼갤 때의 비용(속도 저하, 과도한 테스트 코드)은 어떻게 논의되나?
- 자신의 프로젝트에서 검증 방법을 구체적으로 기록하라. 무엇을 테스트했고, 무엇은 눈으로만 확인했으며, 그 선택은 왜 그렇게 했는가.

조사 포인터

- TDD와 AI 에이전트 결합에 관한 실무 논의(§10.4와 교차 참조 가능)
- 도구별 테스트 자동 생성 및 실행 기능 공식 문서
- 수강생 자신의 프로젝트 검증 절차를 사례로 정리

5.5 되돌릴 수 있게: 버전 관리와 체크포인트

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 에이전트가 코드를 대량으로 고치기 시작하면, 사람이 하나하나 검토하기 전에 이미 여러 파일이 바뀌어 있는 상황이 흔해진다. 이 꼭지는 그런 상황에서 되돌릴 수 있는 상태를 유지하는 기술, 커밋 단위를 어떻게 잡을지, 브랜치를 어떻게 나눌지, 에이전트가 일을 망쳤을 때 어디까지 되돌릴 수 있어야 하는지를 다룬다. §12.1에서 다룰 버전 관리 논의의 실무 매뉴얼에 해당한다.

시드 질문

- 에이전트 작업 단위와 커밋 단위를 일치시키는 관행은 얼마나 널리 퍼져 있나? 구체적 권장 빈도는 무엇인가?
- 에이전트 실행 전 체크포인트를 만드는 도구 기능(자동 커밋, 스냅숏 등)은 어떤 도구에 어떤 형태로 존재하나?
- 브랜치 전략(작업 브랜치, worktree 활용 등)이 에이전트 작업에서 어떻게 달라지나?
- 에이전트가 되돌릴 수 없는 손상(데이터 삭제, 강제 푸시 등)을 낸 사례들은 무엇을 공통 원인으로 지목하나?
- 자신이 에이전트 작업 중 되돌리기를 실제로 사용한 경험을 사례로 기록하라. 무엇이 잘못되었고, 어떻게 복구했는가.

조사 포인터

- Git worktree, 브랜치 전략에 관한 실무 가이드와 도구 문서
- 에이전트 사고(incident) 보고 사례(블로그, 커뮤니티)
- 수강생 자신의 커밋 이력을 사례로 분석(작업 단위와 커밋 단위의 일치 여부)

5.6 언제 멈추고 직접 읽어야 하는가

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 바이브 코딩을 잘하는 사람과 못하는 사람의 차이는 대부분 언제 AI를 믿고 언제 직접 개입해야 하는지를 아는 감각에서 갈린다. 이 꼭지는 그 감각을 가능한 한 명시적인 신호 목록으로 바꾸는 작업이다. 어떤 상황에서 실무자들이 실제로 손을 멈추고 코드를 직접 읽는지, 그 판단을 미루었다가 문제가 커진 사례는 무엇을 보여주는지를 다룬다. 5장의 마지막 꼭지로서 §5.1부터 이어온 실무 원칙들을 이 원칙들이 언제 깨지는가라는 질문으로 마무리한다.

시드 질문

- 실무자들이 공통으로 꼽는 직접 개입해야 하는 신호에는 무엇이 있나? (보안 관련 코드, 데이터 삭제, 결제 로직 등)
- 이 신호를 놓쳐서 문제가 커진 실제 사례(부검, post-mortem)는 어떤 패턴을 보이나?
- 경력이나 도메인 지식에 따라 언제 멈춰야 하는가의 기준이 달라지는가? 초보자와 숙련자의 기준 차이는 무엇인가?
- 팀 단위에서는 이 판단을 개인이 아니라 프로세스(리뷰, 게이트)로 대체하려는 시도가 있는가?
- 자신이 바이브 코딩 도중 실제로 멈추고 직접 코드를 읽은 순간을 사례로 기록하라. 무엇이 멈추게 만들었는가.

조사 포인터

- 바이브 코딩 실패 사례 부검 글, 보안 사고 보고서
- 팀 단위 AI 코딩 거버넌스 정책 문서(§9.5와 교차 참조 가능)
- 수강생 자신의 개입 순간들을 기록해 공통 신호를 도출

6 아티팩트별 실전

"AI로 만든다"는 같은 말이라도 만드는 것이 무엇이냐에 따라 실제 작업은 전혀 다르다. 유형마다 잘 되는 것, 안 되는 것, 사람이 개입해야 하는 지점, 품질을 확인하는 방법이 다르다. 이 장은 그 차이를 유형별로 비교한다. 이 책 자체가 마지막 절의 사례다.

6.1 글쓰기: 보고서, 논문, 문서

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 코드는 실행해서 검증할 수 있지만 글은 그렇지 않다. 이 비대칭이 글쓰기를 바이브 코딩 논의에서 가장 까다로운 아티팩트로 만든다. 이 꼭지는 보고서, 논문, 문서 작성에서 AI 사용이 실제로 어디까지 쓰이는지, 도구인지 대필인지의 경계가 어디에 그어지는지, 학술 글

쓰기의 규범과 AI 특유의 문체(이 저장소가 경계하는 상투구, 과장된 수사 포함) 문제를 다룬다. 6장(아티팩트별 실전)의 첫 꼭지로 §8.3의 검증 비대칭 논의와 직접 연결된다.

시드 질문

- 코드와 달리 글에는 실행해서 확인한다에 해당하는 절차가 없다. 그 공백을 실무자들은 무엇으로 메우나?
- 학술지와 학회의 AI 사용 공개 정책은 어디까지 허용하고 어디부터 금지하나? 분야별 차이는 무엇인가?
- AI 문체로 지목되는 특징(상투적 표현, 과장된 수사, 특정 어휘 패턴)은 어떻게 식별되고 있나?
- 도구로서의 AI 사용과 대필로서의 AI 사용을 가르는 기준을 제시한 논의가 있는가?
- 자신이 이 책이나 다른 글을 AI와 함께 쓴 경험을 사례로 기록하라. 어디까지 AI가 쓰고 어디부터 직접 고쳤는가.

조사 포인터

- 주요 학술지와 학회의 AI 사용 정책 원문(각 저널 홈페이지)
- AI 문체 관련 논의(언어학, 저널리즘 쪽 분석 글)
- 이 책 자체의 집필 과정, 특히 이 스텁이 지키는 문체 규율(브리프 상단 참조)을 사례로 기록

6.2 데이터 분석과 시각화

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 데이터 분석은 바이브 코딩이 특히 빠르게 파고든 영역이다. 탐색적 분석 단계에서 AI에게 코드를 맡기면 그래프와 통계표가 순식간에 나온다. 문제는 그 결과물이 그럴듯해 보이는 것과 실제로 맞는 것이 다를 수 있다는 점이다. 이 꼭지는 탐색적 데이터 분석이 얼마나 가속되었는지, 그럴듯한 그래프가 검증 없이 결론으로 이어지는 함정을 실제 사례로 다룬다. §4.4(연구자의 바이브 코딩)와 맞닿아 있지만, 이 꼭지는 연구뿐 아니라 비즈니스 분석까지 포함한다.

시드 질문

- AI가 생성한 시각화나 통계 분석 코드에서 실제로 발견된 오류 사례(축 왜곡, 통계 오용, 데이터 누출 등)는 무엇이 있나?
- 탐색적 데이터 분석 속도가 빨라지면서 분석가의 역할은 어떻게 바뀌었나? 코드 작성에서 결과 해석과 검증으로 무게중심이 옮겨갔는가?
- 그럴듯한 그래프를 걸러내기 위해 실무자들이 쓰는 구체적 점검 절차는 무엇인가?
- 데이터 분석 도구(노트북 기반 에이전트, 코드 인터프리터류)의 지형은 어떻게 나뉘어 있나?
- 자신이 데이터 분석 결과를 AI로 뽑아낸 뒤 직접 검증한 경험을 사례로 기록하라. 무엇을 검증했고 무엇이 틀려 있었는가.

조사 포인터

- 통계, 데이터 시각화 오류 사례 모음(학계, 저널리즘의 그래프 왜곡 비판 글)
- 코드 인터프리터, 노트북 기반 AI 분석 도구의 공식 문서

- 수강생 자신의 데이터 분석 과제를 사례로 검증 절차를 기록

6.3 웹 애플리케이션: 프론트에서 배포까지

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 웹 애플리케이션은 바이브 코딩의 가장 대표적인 무대다. 프론트엔드 화면 하나를 만드는 것과 백엔드와 데이터베이스를 붙이고 실제로 배포까지 끝내는 것은 난이도가 크게 다르다. 이 꼭지는 아이디어에서 배포까지의 전체 경로를 따라가며 어디까지는 순조롭고 어디서부터 막히는지, 그 병목이 도구의 한계인지 사람의 이해 부족인지를 구분해서 다룬다. §6.1, §6.2가 검증이 어려운 아티팩트를 다뤘다면, 이 꼭지는 검증이 비교적 쉬운(실행하면 보인다) 아티팩트가 왜 그럼에도 완주가 어려운지를 보여준다.

시드 질문

- 풀스택 웹 애플리케이션을 바이브 코딩만으로 기획부터 배포까지 완주한 사례는 실제로 얼마나 흔한가? 중도 포기 사례와 비교하면?
- 프론트엔드, 백엔드, 데이터베이스, 배포 파이프라인 중 바이브 코딩이 가장 잘 통하는 단계와 가장 취약한 단계는 어디인가?
- 인증, 결제, 권한 관리 등 실수 비용이 큰 영역에서 AI 생성 코드의 신뢰도는 어떻게 평가되나?
- 바이브 코딩으로 만든 웹앱 사례 중 실제 사용자를 확보하고 유지한 경우와 데모에 그친 경우의 차이는 무엇인가?
- 자신이 웹 애플리케이션을 기획부터 배포까지 시도한 경험을 사례로 기록하라. 어느 단계에서 가장 오래 막혔는가.

조사 포인터

- 풀스택 바이브 코딩 제작기(블로그, 유튜브, X 스레드)
- 웹 배포 플랫폼(Vercel, Railway 등)의 AI 연동 기능과 공식 사례
- 수강생 자신의 웹 앱 프로젝트 진행 로그를 사례로 정리

6.4 자동화 스크립트와 개인 도구

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 반복 작업을 대신해주는 짧은 스크립트, 여러 도구를 이어붙이는 글루 코드, 개인 워크플로를 자동화하는 CLI 도구는 바이브 코딩이 가장 안정적으로 성공하는 카테고리로 꼽힌다. 이 꼭지는 왜 이 카테고리가 특히 성공률이 높다고 이야기되는지, 실패 위험이 낮은 구조적 이유는 무엇인지, 이 성공 경험이 다른 카테고리(웹 앱, 글쓰기)로 일반화되지 않는 이유를 다룬다. §4.2(1인용 소프트웨어)의 가장 구체적인 실현 형태이기도 하다.

시드 질문

- 자동화 스크립트와 글루 코드가 바이브 코딩에서 성공률이 높다고 이야기되는 근거는 무엇인가? 정량적 조사가 있는가, 일화적 인상인가?
- 이 카테고리의 검증이 쉬운 이유는 무엇인가? (되돌리기 쉬움, 실행 결과가 즉시 보임, 실패 비용이 낮음 등)
- 개인 워크플로 자동화 도구(CLI, 크론잡, 알림 파이프라인 등)의 실제 사례는 어떤 형태로 보고되나?
- 이 카테고리에서 실패하는 경우는 어떤 조건에서 나타나나? (외부 API 변경, 권한 문제 등)
- 자신이 만든 자동화 스크립트나 개인 도구를 사례로 기록하라. 무엇을 자동화했고, 실제로 얼마나 오래 계속 썼는가.

조사 포인터

- 개인 자동화 도구 공유 커뮤니티(해커뉴스 Show HN, 레딧 등) 사례 수집
- 바이브 코딩 성공률에 관한 실증 조사(있다면 1차 출처 확인)
- 수강생 각자가 만든 자동화 스크립트 목록을 사례 데이터로 축적

6.5 슬라이드와 조판물: 이 책의 사례

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 발표 자료와 책 같은 조판물은 시각적 완성도와 구조적 정확성을 동시에 요구한다는 점에서 독특한 검증 문제를 갖는다. 이 꼭지는 슬라이드와 조판물 제작에 AI 코딩 도구가 실제로 어떻게 쓰이는지를 다루면서, 이 책 자체의 빌드 파이프라인을 1차 사례로 삼는다. 이 저장소는 Quarto로 원고를 관리하고 Typst로 PDF를 조판하며, 한국어 원문을 AI로 영어로 번역하는 파이프라인을 쓴다. 담당자는 이 파이프라인이 실제로 어떻게 구성되어 있고 어디서 사람이 개입하는지를 직접 기록해야 한다.

시드 질문

- 이 저장소(vibecoding)의 빌드 파이프라인(`tools/build`, `tools/translate`, Quarto, Typst)은 실제로 어떤 단계로 구성되어 있으며, 각 단계에서 AI는 무엇을 하고 사람은 무엇을 하는가? 저장소의 CLAUDE.md와 tools/ 스크립트를 직접 읽고 실행해 기록하라.
- AI 번역(`en/`의 MACHINE-TRANSLATED 표시가 붙은 파일들)의 품질은 실제로 어느 정도인가? 원문과 대조했을 때 어떤 오류 패턴이 나타나는가?
- 슬라이드 제작 도구(코드 기반 슬라이드 생성기)와 워드, 파워포인트류 GUI 도구에서 AI 활용 방식은 어떻게 다른가?
- 조판물의 동작한다에 해당하는 검증은 무엇인가? (렌더링 성공, 레이아웃 깨짐 없음, 인쇄 규격 준수 등)
- 자신이 이 책 이외의 슬라이드나 문서 조판 작업을 AI와 함께 해본 경험이 있다면 사례로 기록하라.

조사 포인터

- 이 저장소의 `tools/build`, `tools/translate`, `_quarto.yml`, CLAUDE.md를 직접 읽고 실행해 파이프라인을 기록
- Quarto, Typst 공식 문서에서 자동화나 스크립트 연동 관련 서술 확인

- 다른 코드 기반 슬라이드, 조판 도구(마크다운 기반 프레젠테이션 생성기 등) 사례 비교

7 콘텐츠 생성

자연어로 시키고 결과를 보며 고치는 반복은 코드 밖으로도 번졌다. 이미지, 비디오, 오디오, 그리고 이것들을 코드로 엮는 파이프라인까지. 이 장은 콘텐츠 생성의 지형을 정리하고, 코드 생성과 무엇이 같고 무엇이 다른지를 분석한다.

7.1 이미지 생성

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 이미지 생성은 코드 생성과 같은 자연어로 시키고 결과를 보며 고친다는 반복 구조를 공유하지만, 프롬프트 작성 관행과 결과 평가 방식은 상당히 다르다. 이 꼭지는 이미지 생성에서 실제로 쓰이는 프롬프트 관행, 스타일을 일관되게 유지하는 기법, 코드 생성과 비교했을 때 검증 구조가 어떻게 다른지를 다룬다. 7장(콘텐츠 생성)의 첫 꼭지로 §8.3에서 다룰 검증 비대칭 논의를 이미지라는 구체적 아티팩트로 확장한다.

시드 질문

- 이미지 생성 프롬프트 작성 관행(스타일 키워드, 네거티브 프롬프트, 시드 고정 등)은 코드 생성 프롬프트와 어떤 점에서 다른가?
- 여러 이미지에 걸쳐 스타일 일관성을 유지하는 기법은 무엇이 있으며 실제로 얼마나 잘 작동하나?
- 이미지 생성 결과의 품질을 동작한다, 안 한다가 아닌 무엇으로 평가하나? 평가 기준의 주관성 문제는 어떻게 다뤄지나?
- 코드는 실행해서 검증할 수 있지만 이미지는 그럴 수 없다는 비대칭이 프롬프트 작성 관행 자체에 어떤 영향을 주었나?
- 자신이 이미지 생성 도구로 원하는 결과를 얻기까지의 반복 과정을 사례로 기록하라. 몇 번의 시도가 필요했고 무엇을 바꿔가며 수렴했는가.

조사 포인터

- 주요 이미지 생성 모델의 공식 문서와 프롬프트 가이드
- 스타일 제어 기법에 관한 실무자 가이드(정확한 기법명과 최신 상태는 확인할 것)
- 수강생 자신의 이미지 생성 프롬프트 반복 과정을 기록해 사례로 정리

7.2 비디오와 오디오 생성

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 비디오와 오디오 생성은 이미지보다 계산 비용이 크고 시간축이 더해지는 만큼 평가가 더 어렵다. 이 꼭지는 2025년에서 2026년 시점의 비디오 생성 모델 지형, 음성 합성과 음악 생성의 현재 상태, 이 아티팩트들의 품질을 무엇으로 평가하는지를 다룬다. 비디오는 프레

임 간 일관성, 물리 법칙 위반 여부 같은 이미지에는 없던 새로운 실패 양상을 갖는다는 점에서 §7.1(이미지 생성)과 구분된다.

시드 질문

- 2025년에서 2026년 시점에서 주요 비디오 생성 모델은 무엇이며 각각의 강점과 한계는 어떻게 보도되나? (모델명, 출시일은 반드시 1차 출처 확인)
- 비디오 생성에서만 나타나는 실패 양상(프레임 간 일관성 붕괴, 물리 법칙 위반 등)은 어떻게 분류되고 있나?
- 음성 합성과 음악 생성 각각에서 품질이 좋다는 평가는 무엇을 기준으로 내려지나? 사람이 듣고 판단하는 것 외의 정량 지표가 있는가?
- 비디오와 오디오 생성의 저작권 논쟁(학습 데이터, 유명인 목소리 복제 등)은 현재 어디까지 진행되었나?
- 자신이 비디오나 오디오 생성 도구를 써본 경험을 사례로 기록하라. 원하는 결과를 얻기 위해 무엇을 반복해서 조정했는가.

조사 포인터

- 주요 비디오, 오디오 생성 모델의 공식 발표와 기술 블로그(날짜와 모델명은 1차 출처 확인)
- 저작권과 정책 관련 소송, 규제 동향 보도
- 수강생 자신의 생성 경험을 기록

7.3 콘텐츠 파이프라인: 생성을 코드로 엮기

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 개별 이미지나 영상 한 장을 생성하는 것과, 기획부터 생성, 편집, 배포까지를 코드로 엮어 하나의 채널을 운영하는 것은 다른 문제다. 이 꼭지는 콘텐츠 생성을 파이프라인으로 자동화한 사례, 스크립트 작성부터 영상 렌더링, 게시까지 이어지는 자동화된 콘텐츠 제작 과정을 다룬다. §7.1, §7.2가 개별 생성 기술을 다뤘다면 이 꼭지는 그것들을 코드로 오케스트레이션했을 때 무엇이 새로 가능해지는지를 본다.

시드 질문

- 기획, 생성, 편집, 배포를 하나의 파이프라인으로 엮은 콘텐츠 채널 운영 사례는 실제로 어떤 구조를 갖고 있나?
- 파이프라인 자동화는 콘텐츠의 양을 늘리는 대신 질을 어떻게 관리하나? 자동화된 채널과 사람이 매번 개입하는 채널의 결과물 차이는 무엇으로 보고되나?
- 콘텐츠 파이프라인에서 사람이 반드시 개입하는 단계(최종 승인, 편집 등)는 어디이며 왜 그 단계는 자동화되지 않았나?
- 콘텐츠 생성 자동화가 만들어내는 저품질 콘텐츠(low-effort content) 문제는 제3부의 slop 논의와 어떻게 연결되나?
- 자신이 콘텐츠 파이프라인을 직접 구축했거나 구축을 시도한 경험을 사례로 기록하라. 어느 단계까지 자동화했고 어디서 막혔는가.

조사 포인터

- 자동화된 콘텐츠 채널 운영기(유튜브 쇼츠, 뉴스레터 자동화 등 제작 후기)
- 콘텐츠 생성 파이프라인 오픈소스 도구나 프레임워크 사례
- 이 책 저자 집단이나 수강생이 운영하는 콘텐츠 채널이 있다면 그 파이프라인 구조를 사례로 기록

제3부 · 원리와 한계

8 작동 원리

바이브 코딩을 제대로 쓰려면 그 밑에서 무엇이 돌아가는지 대략은 알아야 한다. 이 장은 수학 없이, 그러나 정확하게 설명한다. 다음 토큰 예측이라는 기본 구조, 코딩 에이전트의 해부학, 그리고 왜 하필 코드에서 이 방식이 가장 잘 통했는지.

8.1 다음 토큰 예측에서 코드까지

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. LLM의 기본 동작은 다음에 올 토큰을 예측하는 것뿐이다. 이 단순한 메커니즘에서 코드를 짜는 능력이 어떻게 나오는지, 그리고 왜 코드가 자연어보다 이 메커니즘에 특히 잘 맞는 대상인지를 수식 없이 정확하게 설명하는 것이 이 꼭지의 임무다. 코드의 구문 규칙성, 공개 저장소라는 방대한 학습 데이터, 실행 가능성이 주는 신호가 각각 어떤 역할을 하는지 짚는다. 동시에 이 메커니즘이 어디서 무너지는지, 즉 환각이 코드 생성에서 어떤 형태로 나타나는지도 다룬다. “모델이 이해한다”는 말과 “패턴을 흉내낸다”는 말 사이의 실제 경계가 이 장 전체(8장)의 밑그림이 된다.

시드 질문

- 다음 토큰 예측이라는 단순한 학습 목표에서 코드를 짜는 능력은 구체적으로 어떤 경로로 나타나는가?
- 코드 생성이 자연어 생성보다 학습에 유리한 이유는 무엇인가, 구문 규칙성과 데이터 규모 중 어느 쪽이 더 크게 기여하는가?
- 환각은 코드 생성에서 구체적으로 어떤 형태를 띠는가, 존재하지 않는 API를 지어내는 경우와 틀린 인자를 쓰는 경우는 원인이 같은가?
- “모델이 코드를 이해한다”는 주장과 “패턴을 흉내낼 뿐이다”는 주장은 각각 무엇을 근거로 삼는가?
- 컨텍스트 윈도 크기의 확장은 실제 코딩 작업에서 어떤 성능 개선으로 이어졌는가?

조사 포인터

- Karpathy, “Software 2.0”(2017) 원문을 코드를 데이터로 다루는 관점의 원류로 재독할 것
- 다음 토큰 예측, 트랜스포머 구조에 관한 대중적이면서 정확한 설명 자료를 선별해 인용할 것
- 코드 생성의 환각 현상을 다룬 학술 논문을 우선 검색할 것 (실증 연구가 3부 전체의 기본 태도다)

8.2 코딩 에이전트의 해부학

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 코딩 에이전트가 한 턴에서 실제로 무엇을 하는지 분해한다. 사용자 입력이 프롬프트로 구성되는 과정, 도구 호출(파일 읽기·쓰기, 명령 실행), 실행 결과가 다시 컨텍스트로 들어가는 피드백 루프, 그리고 이 루프가 몇 차례 반복되며 작업이 종료되는지를 단계별로 짚는다. Anthropic의 "Building effective agents"가 제시하는 워크플로와 에이전트의 구분을 이론 축으로 삼고, 오픈소스 코딩 에이전트의 실제 소스 코드를 대조해 이론과 구현의 간극을 확인하는 것이 핵심 작업이다.

시드 질문

- 코딩 에이전트 한 턴을 사용자 입력부터 최종 응답까지 단계별로 분해하면 무엇이 나오는가?
- 도구 호출(tool use)은 모델 쪽에서 무엇을 의미하고, 하네스 쪽에서는 무엇을 준비해줘야 성립하는가?
- Anthropic이 구분하는 "워크플로"와 "에이전트"의 경계는 어디이고, 실제 코딩 에이전트들은 어느 쪽에 더 가까운가?
- 오픈소스 코딩 에이전트의 소스에서 메인 루프는 실제로 어떻게 구현되어 있는가, 문서상 설명과 일치하는가?
- 실행 결과(테스트 실패, 에러 메시지, 셸 출력)가 다시 모델 입력으로 들어가는 피드백 루프가 없으면 무엇이 달라지는가?

조사 포인터

- Anthropic, "Building effective agents"(2024-12) 원문을 정독하고 이 장의 이론 축으로 쓸 것
- 오픈소스 코딩 에이전트(CLI형 도구 여러 종)의 GitHub 저장소에서 메인 루프 코드를 직접 읽을 것
- 하네스 엔지니어링 관련 자료를 찾아 §3.4, glossary의 agent harness 항목과 교차 참조할 것

8.3 검증 비대칭: 왜 하필 코드였나

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 코드가 AI 생성의 첫 주력 분야가 된 이유 중 하나는 검증 가능성이다. 코드는 실행하면 맞았는지 틀렸는지 적어도 부분적으로 즉시 알 수 있지만, 산문이나 그림은 그렇지 않다. 이 비대칭이 왜 생기는지, 테스트·컴파일·타입 체크 같은 자동 검증 신호가 모델 학습과 에이전트 루프 설계에서 실제로 어떤 역할을 하는지, 그리고 이 검증 이점이 글쓰기나 다른 창작 분야로 일반화되려면 어떤 조건이 필요한지를 다룬다. §6.1(글쓰기)의 "검증이 어려운 아티팩트" 논의와 대칭을 이루는 꼭지다.

시드 질문

- “코드는 실행하면 검증된다”는 명제는 정확히 무엇을 검증하고 무엇을 검증하지 못하는가?
- 자동 검증 신호(테스트 통과, 컴파일 성공, 타입 체크)는 에이전트 루프 설계에 구체적으로 어떻게 반영되는가?
- 검증 비대칭 가설을 뒷받침하거나 반박하는 실증 연구는 무엇이 있는가?
- 이 검증 이점이 글쓰기, 이미지, 데이터 분석 같은 다른 산출물로 일반화되려면 어떤 조건(형식화 가능성, 채점 가능한 기준의 존재)이 필요한가?
- 검증 가능한 작업 위주로 자동화가 앞서가는 현상은 장기적으로 어떤 편향을 만드는가?

조사 포인터

- 검증 가능한 보상(verifiable reward) 기반 학습을 다룬 실증 연구를 찾아 확인할 것
- §5.4(검증 가능한 단위 만들기), §6.1(글쓰기)과 교차 참조해 중복 없이 역할을 나눌 것
- 소프트웨어 테스팅 이론과 AI 평가(evals) 문헌이 만나는 지점을 확인할 것

9 한계와 리스크

바이브 코딩의 약속 뒤에는 계산서가 있다. 이 장은 그 계산서를 정직하게 편다. 이해의 공동화(블랙박스가 되어가는 코드베이스), 보안, 품질 부채, 전문가 양성 경로의 단절, 그리고 책임의 문제까지. 낙관도 공포도 아닌 근거가 기준이다.

9.1 블랙박스가 되어가는 코드베이스

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI가 빠르게 짜는 코드는 그만큼 빠르게, 아무도 전체를 이해하지 못하는 상태로 쌓인다. 이 꼭지는 이해의 공동화 문제를 다룬다. 개별 커밋은 동작하지만 전체 아키텍처를 파악하고 있는 사람이 조직에서 사라지는 과정, 그리고 그런 코드베이스 위에 다시 AI가 코드를 얹을 때 무슨 일이 일어나는지를 살핀다. “이해 없이 동작하는 코드”가 언제 문제가 되고 언제 그렇지 않은지 구분하는 기준을 세우는 것이 이 꼭지의 핵심 논증이다. 9장 전체(한계와 리스크)의 도입에 해당한다.

시드 질문

- “이해 없이 동작하는 코드”가 실제로 사고나 장애로 이어진 사례가 보고되어 있는가?
- 코드베이스 이해도와 유지보수 비용의 관계를 다룬 실증 연구는 무엇을 보여주는가?
- AI가 이미 AI가 짠 코드 위에 다시 코드를 얹을 때, 오류나 스타일 불일치가 누적되는 현상은 확인되는가?
- 조직은 “아무도 전체를 모르는” 상태를 어떻게 측정하거나 완화하려 하는가, 문서화 의무나 온보딩 관행이 실제로 바뀌었는가?
- 개인 프로젝트(1인 개발)와 팀 코드베이스에서 이 문제의 양상은 어떻게 다르게 나타나는가?

조사 포인터

- 프로그램 이해 가능성(program comprehension), 코드베이스 복잡도를 다룬 학술 논문을 우선 찾을 것
- 장애 부검(postmortem) 보고서, 유지보수 경험담 등 실무 현장 보고를 수집할 것
- Parnas(1972)의 모듈 분해 논거를 다루는 §11.1과 교차 참조할 것

9.2 보안: 새로운 공격면

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI가 생성한 코드는 사람이 짠 코드와 다른 종류의, 혹은 다른 빈도의 취약점을 만드는가. 이 꼭지는 생성 코드의 보안 취약점을 다룬 실증 연구, 에이전트가 외부 콘텐츠(검색 결과, 문서, 이슈 코멘트)를 읽다가 악성 지시를 실행하게 되는 프롬프트 인젝션, 그리고 모델이 존재하지 않는 패키지 이름을 지어내고 공격자가 그 이름을 먼저 등록하는 slopsquatting을 다룬다. 수치와 사례는 반드시 1차 출처로 확인한 뒤 인용한다.

시드 질문

- AI 생성 코드의 취약점 비율을 사람이 짠 코드와 비교한 실증 연구는 무엇이 있고, 방법론은 신뢰할 만한가?
- 프롬프트 인젝션은 코딩 에이전트 맥락에서 구체적으로 어떤 공격 경로를 만드는가?
- slopsquatting은 실제로 보고된 사례가 있는가, 피해 규모는 어느 정도로 추정되는가?
- 코딩 에이전트에 부여되는 실행 권한(셸 명령, 파일 쓰기, 네트워크 접근)은 보안 모델을 어떻게 바꾸는가?
- 벤더(Anthropic, GitHub 등)는 이런 공격면에 대해 어떤 완화책을 공식 문서로 제시하고 있는가?

조사 포인터

- 생성 코드 보안 취약점을 정량 분석한 학술 논문을 우선 검색할 것
- 프롬프트 인젝션, slopsquatting을 다룬 보안 업체·연구자의 실증 보고서를 확인할 것
- glossary의 slopsquatting 항목(G14)과 교차 참조하되 역할을 나눠 중복을 피할 것

9.3 품질 부채와 유지보수

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 빠르게 생성된 코드는 종종 동작은 하지만 고치기는 어렵다. 이 꼭지는 품질 부채가 기존 기술 부채 개념과 어떻게 같고 다른지, AI의 생성 속도가 부채 누적 속도를 어떻게 바꾸는지, 그리고 처음부터 다시 짜는 편이 언제 더 싼 선택이 되는지를 다룬다. 리라이트 비용을 재무적으로 따지는 시도가 있다면 확인해 소개한다. §11.2(기술 부채, 재발견 관점)와 짝을 이루되 이 꼭지는 실무·재무 관점에 무게를 둔다.

시드 질문

- AI가 생성한 코드는 사람이 짠 코드에 비해 유지보수 비용이 다르다는 실증 연구가 있는가?
- 기술 부채 은유를 AI 생성 코드에 적용할 때 무엇이 맞고 무엇이 맞지 않는가?
- “동작하지만 고칠 수 없는 코드”를 판별하는 구체적 기준(복잡도 지표, 리뷰 소요 시간 등)은 실무에서 무엇이 쓰이는가?

- 리라이트와 점진적 리팩토링 중 어느 쪽이 더 싼지 판단하는 재무적 프레임은 무엇이 제안되어 있는가?
- 코드 생성 속도의 증가는 리뷰·테스트 역량의 증가 속도를 앞지르고 있는가, 데이터로 확인되는가?

조사 포인터

- Cunningham(1992)의 기술 부채 원 발표(OOPSLA experience report)를 원문으로 확인할 것
- 소프트웨어 유지보수 비용, 리팩토링 대 재작성 판단에 관한 학술 논문을 우선 검색할 것
- §11.2와 역할을 나눌 것: 이 꼭지는 실무·재무 관점, §11.2는 개념사 관점

9.4 주니어의 역설: 사다리가 사라진다

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI가 가장 잘 대신해주는 업무는 대개 주니어가 맡아 성장하던 업무다. 이 꼭지는 초급 업무의 자동화가 전문가로 가는 사다리를 끊는다는 우려를 다룬다. 주니어 개발자 채용 데이터, 인턴십과 신입 공고의 변화를 찾아, 이 문제가 실제로 데이터로 확인되는지 아니면 과장된 서사인지를 가른다. §13.3(전문성의 사다리)과 짝을 이루되 이 꼭지는 노동시장·채용 데이터 쪽에 무게를 둔다. 수치는 반드시 원 데이터로 확인한 뒤 인용한다.

시드 질문

- 주니어 개발자 채용 공고 수, 인턴십 규모는 최근 실제로 어떻게 변했는가, 신뢰할 만한 데이터 출처는 어디인가?
- “주니어가 AI로 더 빨라졌다”는 관찰과 “주니어 채용이 줄었다”는 관찰은 같은 현상의 다른 면인가, 별개의 현상인가?
- 기업들은 신입 채용 감소를 AI 때문이라고 공식적으로 인정한 적이 있는가, 아니면 경기나 과잉 채용 조정 같은 다른 요인이 더 크게 작용하는가?
- “사다리가 사라진다”는 우려를 반박하거나 다른 설명을 제시하는 연구는 무엇이 있는가?
- 신입 채용 감소에 대한 기업·업계 차원의 대응(주니어 전담 프로젝트 유지, 멘토링 재설계 등) 사례가 보고되어 있는가?

조사 포인터

- 채용 공고와 고용 통계를 다루는 노동경제학 실증 연구를 우선 검색할 것, 수치는 반드시 원 데이터로 확인한 뒤 인용할 것
- 업계 서베이(개발자 채용 동향 보고서 등)와 학술 연구를 구분해 신뢰도를 다르게 취급할 것
- §13.3(전문성의 사다리), §14.2(직업 지형의 변화)와 역할을 나눌 것: 이 꼭지는 코딩 사다리에 한정

9.5 책임과 서명: 누가 이 코드에 사인하는가

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI가 짠 코드에서 사고가 났을 때 책임은 누구에게 있는가. 이 꼭지는 코드 리뷰와 배포 승인이라는 전통적 서명 관행이 AI 생성 코드 앞에서 어떻게 재정의되는지, 실제 기업·기관이 도입한 AI 코드 거버넌스 정책(리뷰 의무, 생성 코드 표기, 책임자 지정)이 무엇인지, 그리고 사고 사례에서 책임이 실제로 어떻게 배분되었는지를 다룬다. 9장의 마지막 꼭지로 앞선 문제들(이해의 공동화, 보안, 품질 부채)이 조직 차원에서 누구의 책임으로 귀결되는지 묻는다.

시드 질문

- AI 생성 코드로 인한 장애나 사고에서 책임이 실제로 어떻게 귀속되었는가, 공개된 사례가 있는가?
- 기업·기관이 공식화한 AI 코드 거버넌스 정책은 리뷰 의무, 승인 절차, 생성 코드 표기와 관련해 구체적으로 무엇을 요구하는가?
- 전통적 코드 리뷰의 승인이라는 관행은 AI 생성 코드 앞에서 형식적으로만 유지되고 실질은 비어가는가?
- 오픈소스 프로젝트는 AI 생성 기여물에 대해 어떤 표기·심사 규칙을 도입하고 있는가?
- 법적·계약적 책임 논의(제조물 책임과의 유비 등)는 어느 수준까지 진행되어 있는가?

조사 포인터

- 기업·정부 기관이 발표한 AI 코드 거버넌스 정책 문서를 1차 출처로 수집할 것
- 오픈소스 프로젝트의 AI 기여 정책(컨트리뷰션 가이드라인)에서 실제 사례를 확인할 것
- 코드 리뷰의 목적론을 다루는 §10.3과 교차 참조하되 역할을 나눌 것: 이 꼭지는 책임·거버넌스에 한정

제4부 · 소프트웨어 엔지니어링의 재발견

10 방법론의 재발견

지난 50년의 소프트웨어 엔지니어링은 AI 시대에 하나씩 다시 발견되고 있다. 이 부(部)의 각 꼭지는 고전 개념 하나를 잡고 원전을 읽은 뒤, AI 시대의 재등장과 대조한다. 이 장은 일하는 방식의 방법론들이다. 폭포수와 애자일, 페어 프로그래밍, 코드 리뷰, TDD.

10.1 폭포수와 애자일, 다시

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 스펙 주도 개발과 바이브 코딩의 대립이 폭포수와 애자일 논쟁의 재상연인지 묻는 꼭지다. 3단 구조로 쓴다. 먼저 Royce(1970)의 원논문이 실제로 무엇을 주장했는지, Agile Manifesto(2001)의 4가치와 12원칙이 원문으로 무엇을 말하는지 정확히 읽는다. 다음으로 GitHub Spec Kit류 스펙 주도 개발과 프롬프트로 시작해 반복 확인하는 바이브 코딩 실무가 각각 어느 쪽에 가까운지 최근 사례를 모은다. 마지막으로 반복 주기의 길이, 명세의 위치처

럼 정말 같은 지점과 에이전트라는 새 행위자가 양쪽 모델의 전제를 바꾸는 지점을 가려 논증한다.

시드 질문

- Royce(1970) 원논문을 실제로 읽으면 그가 정말 폭포수를 순수 순차 모델로 옹호했는가, 아니면 이미 반복의 필요성을 경고하고 있었는가? (원전 확인)
- Agile Manifesto의 4가지 가치와 12원칙 중 어떤 조항이 오늘날 바이브 코딩의 정당화 논리로 가장 자주 인용되는가? (원전 확인)
- 스펙 주도 개발(Spec Kit 등) 지지자들은 자신의 워크플로를 폭포수의 부활로 규정하는가, 아니면 애자일 내부의 한 변형으로 규정하는가? (재발견 사례 수집)
- 프롬프트로 시작해서 반복 확인하는 바이브 코딩 실무는 애자일의 반복 주기와 어떤 점에서 같고 다른가, 특히 반복 단위 시간이 초 단위로 줄었을 때도 이론이 성립하는가?
- 폭포수와 애자일 논쟁 당시의 비판(문서 과잉, 계획의 허구성)이 오늘날 스펙 주도 개발에 대한 비판과 얼마나 겹치는가?
- 실무자들이 두 워크플로를 섞어 쓰는 사례가 있다면 어떤 기준으로 나누는가? (재발견 사례 수집)

조사 포인터

- Royce 1970 원논문(저자명·연도로 검색해 원문 확인)
- agilemanifesto.org 원문(4가치, 12원칙)
- GitHub Spec Kit 발표 글과 실무자들의 채택·비판 블로그
- 이 책 §3.3(스펙 주도 개발), §2.2(용어의 확산)과의 접점, 중복 서술 피할 것

10.2 페어 프로그래밍: 짝이 사람이 아니게 될 때

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Beck의 XP(1999)가 페어 프로그래밍을 정당화한 논거를 AI를 짝으로 삼는 오늘날의 실무와 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 XP 원문에서 상시 리뷰, 지식 전파, 역할 교대 같은 페어링 논거를 정확히 읽는다. 다음으로 도구가 스스로를 "AI 페어 프로그래머"로 소개하는 방식과 실무자들의 워크플로 묘사를 사례로 모은다. 마지막으로 대등한 두 행위자, 상호 학습, 사회적 압력이라는 사람 페어링의 전제가 사람과 AI 관계에서 성립하는지 성립하지 않는지 논증한다.

시드 질문

- Beck이 "Extreme Programming Explained"(1999)에서 페어 프로그래밍을 옹호한 핵심 논거는 정확히 무엇이었나, 단순한 결함 감소를 넘어선 부분까지 확인할 것 (원전 확인)
- XP의 페어링은 운전자와 관찰자 역할 교대를 전제하는데, AI와의 협업에서 이 역할 교대가 실제로 일어나는가, 아니면 사람이 항상 관찰자 역할에 고정되는가?
- AI 페어 프로그래밍이라는 표현을 처음 쓴 제품이나 문서는 무엇인가, 그 표현이 XP의 원래 의미를 어디까지 계승하고 어디서부터 은유로 미끄러지는가? (재발견 사례 수집)
- 사람과 사람의 페어링 연구가 보고한 효과(결함 감소, 학습 효과) 중 사람과 AI 페어링에서도 재현된다고 주장하는 최근 연구나 사례가 있는가? (재발견 사례 수집)

- 페어 프로그래밍의 사회적 메커니즘, 즉 부끄러움과 동료 압력과 즉각적 설명 요구는 AI 상대에게는 작동하지 않는데, 이 차이가 실제 작업 품질에 어떤 영향을 주는가?
- XP 실천가들 사이에서도 있었던 페어 프로그래밍 반론(피로도, 비용)이 AI 페어 시대에는 어떻게 다르게 제기되는가?

조사 포인터

- Beck, “Extreme Programming Explained”(1999) 원문 중 페어 프로그래밍을 다룬 장
- 페어 프로그래밍 실증 연구(결함율, 소요 시간을 측정한 논문류)
- GitHub Copilot, Cursor 등 도구의 페어 프로그래머 자기 소개 문구와 발표 자료
- 실무자 블로그의 AI와 페어하는 워크플로 묘사

10.3 코드 리뷰: Fagan 인스펙션에서 AI 리뷰어까지

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Fagan의 인스펙션(1976)이 세운 코드 리뷰의 원래 목적을 LLM 기반 자동 리뷰 도구와 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 Fagan 인스펙션의 형식적 절차, 즉 사전 준비와 역할별 참가자와 결함 분류와 재작업 확인을 원전에서 정확히 읽는다. 다음으로 AI 리뷰어 도구가 실제로 무엇을 검사하고 무엇을 놓치는지 사례를 모은다. 마지막으로 결함 발견, 지식 전파, 조직 규범 확립이라는 리뷰의 세 기능 중 AI 시대에 무엇이 남고 무엇이 빠지는지 논증한다.

시드 질문

- Fagan(1976)의 인스펙션 절차에서 참가자 역할(모더레이터, 저자, 리뷰어, 기록자)과 결함 분류 체계는 정확히 어떻게 정의되어 있었나? (원전 확인)
- Fagan 인스펙션이 명시적으로 방지하려 한 것, 즉 저자가 스스로를 방어하는 상황이나 형식 없는 즉흥 리뷰는 오늘날 AI 리뷰 도구에서 어떤 형태로 재현되거나 사라지는가? (원전 확인)
- 코드 리뷰의 전통적인 세 목적, 즉 결함 발견과 신참에 대한 지식 전파와 팀 코딩 규범 확립 중 AI 리뷰어는 무엇을 대체하고 무엇을 대체하지 못하는가? (재발견 사례 수집)
- AI가 작성한 코드를 AI가 리뷰하는 구조는 Fagan이 전제한 저자와 리뷰어의 독립성 원칙과 어떻게 충돌하는가?
- 실무에서 AI 리뷰 도구가 지적한 내용을 사람이 실제로 반영하는 비율에 대한 데이터가 존재하는가? (재발견 사례 수집)
- Fagan 인스펙션 이후 경량 코드 리뷰로 넘어가면서 이미 한 번 형식성이 후퇴했는데, AI 리뷰어의 등장은 형식성을 다시 높이는 방향인가 낮추는 방향인가?

조사 포인터

- Fagan, “Design and Code Inspections to Reduce Errors in Program Development”, IBM Systems Journal(1976) 원문
- GitHub Copilot code review, Claude Code /code-review 등 도구 공식 문서
- 코드 리뷰 실증 연구(대형 기술 기업의 코드 리뷰 관행을 다룬 논문류)
- 이 책 §9.3(품질 부채)과 겹치지 않도록 리뷰 절차와 목적 자체에 집중할 것

10.4 TDD의 부활: 테스트가 스펙이 될 때

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Beck의 TDD(2002)가 제시한 red-green-refactor 사이클과 테스트가 곧 명세라는 논거를 에이전트 코딩에서 테스트를 먼저 작성해 가드레일이자 검증 신호로 쓰는 최근 관행과 대조하는 꼭지다. 검증이 병목이라는 이 책의 관찰(§8.3)과 직결된다. 3단 구조로 쓴다. 먼저 TDD 원전의 사이클과 논거를 정확히 읽는다. 다음으로 에이전트 워크플로에서 테스트 우선 프롬프팅이 실무 관행으로 자리잡은 과정을 사례로 모은다. 마지막으로 Beck이 강조한 설계 도구로서의 TDD와 AI 출력 검증 도구로서의 테스트가 같은 것인지 다른 것인지 논증한다.

시드 질문

- Beck이 "Test-Driven Development: By Example"(2002)에서 제시한 red-green-refactor 사이클의 정확한 순서와, 테스트를 설계 도구라고 부른 맥락은 무엇인가? (원전 확인)
- Beck의 TDD 논거는 결함 감소보다 설계 압력, 즉 작은 단위로 나누고 인터페이스를 먼저 생각하는 데 방점이 있었는데, 이 설계 압력 논거가 AI가 코드를 쓰는 상황에서도 똑같이 성립하는가, 아니면 사람이 코드를 쓸 때만 유효했던 논거인가?
- 테스트를 먼저 작성하게 하고 에이전트가 통과시키게 하는 워크플로가 도구 문서나 실무 관행에서 언제부터 명시적으로 등장했는가? (재발견 사례 수집)
- TDD가 원래 전제한, 테스트를 쓰는 사람과 코드를 쓰는 사람이 같다는 조건이 에이전트가 테스트와 구현을 모두 작성하는 상황에서는 어떻게 깨지는가, 이것이 검증 신호로서의 유효성을 훼손하는가?
- TDD 부활론자들이 인용하는 성공 사례와, AI가 테스트를 통과시키기 위해 테스트 자체를 조작하는 실패 사례가 둘 다 보고되는가? (재발견 사례 수집)
- Kent Beck 본인이 최근 AI 코딩과 TDD의 관계에 대해 공개적으로 언급한 적이 있는가, 있다면 무엇이라 했는가?

조사 포인터

- Beck, "Test-Driven Development: By Example"(2002) 원문(서문, 초반 예제 장)
- 에이전트 코딩 도구의 테스트 우선 권장 문서, 실무자 블로그
- 이 책 §8.3(검증 비대칭)과의 접점, 중복 서술 피할 것
- Kent Beck 최근 인터뷰나 블로그에서 AI 코딩 관련 언급이 있는지 확인

11 원리의 재발견

방법론 아래에는 원리가 있다. 추상화와 정보 은닉, 기술 부채, 조직과 시스템 구조의 동형성, 본질적 복잡성과 우연적 복잡성, 그리고 사람과 시간의 교환 법칙. 반세기 전의 논문들이 컨텍스트 윈도와 에이전트의 시대에 어떻게 다시 읽히는지가 이 장의 내용이다.

11.1 추상화와 정보 은닉: Parnas가 옳았던 이유, 다시

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Parnas(1972)의 정보 은닉 기준, 즉 모듈을 기능이 아니라 변경될 가능성이 있는 결정을 기준으로 나눠야 한다는 논거를 컨텍스트 창이 유한한 AI 에이전트 시대의 코드베이스 설계와 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 Parnas 원논문이 비교한 두 가지 모듈 분해 방식을 정확히 읽는다. 다음으로 AI 에이전트가 코드를 다루는 방식이 이 기준을 다시 요구하는 사례를 모은다. 마지막으로 사람의 인지 한계를 위해 고안된 원칙이 모델의 컨텍스트 한계를 위해서도 같은 형태로 유효한지 논증한다.

시드 질문

- Parnas(1972)가 논문에서 비교한 두 가지 모듈 분해 방식, 즉 흐름도 기준 분해와 정보 은닉 기준 분해는 정확히 어떤 예제로 제시되었나? (원전 확인)
- Parnas가 정보 은닉의 근거로 든 것은 변경 용이성이었고 이는 사람 프로그래머의 인지 부담을 줄이기 위한 논거였다. 이 논거가 컨텍스트 창이 유한한 AI 에이전트에게도 그대로 적용되는가, 아니면 에이전트의 한계는 성격이 달라 다른 분해 기준이 필요한가? (원전 확인)
- 에이전트 코딩 도구의 공식 문서나 실무 가이드가 에이전트가 다루기 좋은 코드베이스 구조를 명시적으로 권장하는 사례가 있는가, 있다면 그것이 Parnas의 정보 은닉 기준과 얼마나 겹치는가? (재발견 사례 수집)
- AI 친화적 리포지토리 구조를 표방하는 리팩토링 사례나 논의(작은 파일, 명확한 파일명, 순수 함수 선호 등)를 얼마나 찾을 수 있는가? (재발견 사례 수집)
- Parnas 이후 발전한 다른 모듈화 원칙(응집도와 결합도, SOLID 등)과 비교했을 때, AI 시대의 재발견이 정보 은닉 하나에 집중되는 이유는 무엇인가?
- 컨텍스트 엔지니어링(이 책 §3.2)에서 말하는 무엇을 보여줄 것인가의 문제와 Parnas의 모듈 경계 문제는 같은 문제의 다른 이름인가, 다른 문제인가?

조사 포인터

- Parnas, “On the Criteria to Be Used in Decomposing Systems into Modules”(1972) 원문
- 이 책 §3.2(컨텍스트 엔지니어링), §5.3(컨텍스트 관리의 기술)과의 접점, 중복 서술 피할 것
- 에이전트 코딩 도구의 리포지토리 구조 권장 가이드(공식 문서, 엔지니어링 블로그)
- 응집도와 결합도 등 이후 모듈화 이론과의 비교 자료

11.2 기술 부채: 은유가 이자율을 만났을 때

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Cunningham이 OOPSLA 1992 experience report에서 처음 쓴 기술 부채 은유의 원래 의미를 생성 속도가 부채 누적 속도를 크게 높인 AI 코딩 시대의 재소환 사례와 대조하는

꼭지다. 3단 구조로 쓴다. 먼저 Cunningham의 원래 발언과 이후 은유가 오용되어 온 경위를 정확히 읽는다. 다음으로 AI 시대의 부채 관련 논의, 생성 코드의 유지보수 비용을 다룬 실증 조사를 사례로 모은다. 마지막으로 의도적이고 상환 계획이 있는 부채와 무분별한 부채라는 원래 은유의 구분이 AI 생성 코드에도 적용 가능한지 논증한다.

시드 질문

- Cunningham이 1992년 OOPSLA experience report에서 실제로 사용한 표현과 맥락은 정확히 무엇이었나, 그가 나중에 이 은유의 오용에 대해 언급한 적이 있다면 그 내용은 무엇인가? (원전 확인)
- Cunningham의 원래 논지는 빚을 지는 것 자체가 나쁘다는 것이 아니라 빚을 인식하고 상환 계획을 세워야 한다는 것에 가까웠는데, 이 조건부적 성격이 오늘날 기술 부채라는 말이 쓰이는 방식에서 얼마나 유지되고 있는가? (원전 확인)
- AI가 빠르게 생성한 코드에 대해 기술 부채가 크게 늘고 있다는 주장을 실증적으로 뒷받침하는 조사나 데이터가 존재하는가? (재발견 사례 수집)
- 이자율 은유를 문자 그대로 확장해서 AI 생성 코드의 이자율을 정량적으로 논의한 사례가 있는가, 있다면 무엇을 측정 지표로 삼았는가? (재발견 사례 수집)
- 부채를 의도적으로 지는 경우(빠른 프로토타이핑을 위해)와 무분별하게 지는 경우(검증 없이 AI 출력을 그대로 반영)의 구분이 AI 코딩 워크플로에서 실제로 관찰되는가?
- 기술 부채 개념이 정량화 시도(부채 지표, 정적 분석 도구류)를 거쳐온 역사가 있는데, AI 생성 코드에 대해서도 유사한 정량화 시도가 있는가?

조사 포인터

- Cunningham의 OOPSLA 1992 experience report 원문(정확한 제목과 인용은 확인할 것)
- Cunningham 본인의 이후 인터뷰에서 은유의 오용에 대한 코멘트가 있는지 확인
- 이 책 §9.3(품질 부채와 유지보수)과 역할 분담. 그 꼭지는 현재 실태에, 이 꼭지는 은유 자체의 역사와 재소환에 집중
- AI 생성 코드 유지보수 비용에 관한 최근 실증 연구나 업계 보고서

11.3 Conway의 법칙과 인간-AI 조직

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Conway(1968)의 원논문이 제시한 명제, 즉 시스템 설계가 그것을 만든 조직의 소통 구조를 반영한다는 관찰을 사람과 에이전트가 뒤섞인 조직의 코드 구조와 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 Conway 원논문의 정확한 주장과 근거를 읽는다. 다음으로 사람과 AI가 혼합된 조직의 코드 구조 관련 사례를 모은다. 마지막으로 Conway가 전제한 소통 비용이라는 메커니즘이 사람과 근본적으로 다른 행위자인 AI 에이전트에게도 같은 방식으로 작동하는지 논증한다.

시드 질문

- Conway가 1968년 논문에서 실제로 사용한 사례와 논증, 즉 위원회의 설계 산출물이 위원회 자신의 구조를 닮는다는 관찰은 정확히 무엇이었나? (원전 확인)

- Conway의 법칙이라는 이름과 역 Conway 전략(원하는 아키텍처에 맞게 조직을 먼저 설계하는 전략)이라는 표현은 Conway 본인이 만든 것인가, 후대에 붙여진 것인가? (원전 확인)
- 한 사람이 여러 에이전트를 동시에 지휘하는 작업 구조에서, 조직 구조가 코드 구조를 닮는다는 명제는 무엇을 닮는다고 봐야 하는가, 지휘자 1인의 인지 구조인가 에이전트들의 통신 프로토콜인가?
- 멀티에이전트 시스템을 운용하는 조직에서 역 Conway 전략을 의도적으로 적용한 사례, 즉 원하는 소프트웨어 아키텍처를 얻기 위해 에이전트 조직 구조를 먼저 설계한 사례가 보고된 적이 있는가? (재발견 사례 수집)
- Conway의 법칙이 성립하는 메커니즘으로 흔히 꼽히는 소통 비용은 사람 사이에서는 회의와 문서화와 신뢰 형성으로 이루어지는데, 에이전트 사이의 소통 비용은 무엇으로 구성되며 이것이 법칙의 성립 여부를 바꾸는가?
- 멀티에이전트 프레임워크의 설계 자체가 이미 Conway의 법칙을 의식하고 있는가, 즉 도구 설계자들이 이 법칙을 인용하며 아키텍처를 정당화한 사례가 있는가? (재발견 사례 수집)

조사 포인터

- Conway, “How Do Committees Invent?”(1968) 원논문(원 게재지 확인)
- 역 Conway 전략 용어의 출처(Conway 본인인지 후대 정리자인지 확인)
- 멀티에이전트 코딩 시스템 설계 문서, 실무자들의 에이전트 조직 운용 후기
- 이 책 14장(일의 재구성) 관련 꼭지와의 접점, 역할 중복 피할 것

11.4 No Silver Bullet, 다시 읽기

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Brooks(1986)가 제시한 본질적 복잡성과 우연적 복잡성의 구분, 그리고 생산성을 10배 높일 단일 기술은 없다는 주장을 LLM 코딩 도구가 실제로 어느 쪽 복잡성을 줄이는지와 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 Brooks 원논문의 두 복잡성 구분과 그가 검토한 유망 기법들에 대한 평가를 정확히 읽는다. 다음으로 LLM 코딩 도구의 생산성 관련 실증 연구를 사례로 모은다. 마지막으로 Brooks가 은총알이 아니라고 판단한 근거가 LLM에도 그대로 적용되는지, 아니면 LLM이 그가 상상하지 못한 종류의 해법인지 논증한다.

시드 질문

- Brooks가 “No Silver Bullet”(1986)에서 구분한 본질적 복잡성과 우연적 복잡성의 정확한 정의는 무엇이었나, 각각의 예로 든 것은 무엇이었나? (원전 확인)
- Brooks는 논문에서 이미 인공지능을 은총알 후보로 검토하고 기각했는데, 그가 기각한 근거는 정확히 무엇이었나, 그 근거는 오늘날의 LLM에도 여전히 유효한가? (원전 확인)
- LLM 코딩 도구의 생산성 향상을 측정한 실증 연구는 그 향상이 본질적 복잡성을 줄인 것인지 우연적 복잡성(타이핑, 문법, 보일러플레이트)을 줄인 것인지 구분해서 보고하는가? (재발견 사례 수집)
- LLM이 마침내 은총알이라는 취지의 주장과, LLM도 결국 우연적 복잡성만 줄일 뿐이라는 반론이 실제로 맞서고 있는가, 있다면 각각의 핵심 논거는 무엇인가? (재발견 사례 수집)

- Brooks가 은총알이 없다고 결론 내린 이유 중 하나는 소프트웨어의 본질적 어려움(복잡성, 정합성, 변경 가능성, 비가시성)이 표현 형식을 바꿔도 사라지지 않는다는 것이었는데, 자연어 프롬프트라는 새 표현 형식이 이 네 어려움 중 어느 것을 실제로 줄이는가?
- Brooks 본인이 이후 인터뷰나 후기 저작에서 자신의 1986년 예측을 되짚어본 발언이 있는가, 있다면 무엇을 재확인하거나 수정했는가?

조사 포인터

- Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering"(1986) 원문
- 본질적 복잡성과 우연적 복잡성 구분을 다룬 이후 소프트웨어 공학 문헌
- LLM 코딩 생산성에 관한 실증 연구(기업 및 학계 조사)
- 이 책 4장(기회와 포텐셜), 8장(작동 원리)과의 접점, 이 꼭지는 Brooks의 프레임을 통한 재평가에 집중

11.5 Mythical Man-Month: 사람-월에서 에이전트-시간으로

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Brooks(1975)의 핵심 법칙, 즉 지연된 프로젝트에 인력을 추가하면 더 늦어진다는 주장과 그 근거를 에이전트를 추가로 병렬 투입하는 최근 실무와 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 Brooks의 법칙과 의사소통 오버헤드 근거를 원전에서 정확히 읽는다. 다음으로 여러 에이전트를 동시에 돌리는 병렬 운용 사례를 모은다. 마지막으로 사람을 추가하는 것과 에이전트를 추가하는 것이 의사소통 오버헤드라는 메커니즘 면에서 같은지 다른지 논증한다.

시드 질문

- Brooks(1975)가 지연된 프로젝트에 인력을 추가하면 더 늦어진다는 법칙을 도출한 정확한 논증은 무엇이었나, 특히 의사소통 경로 수가 인원수의 어떤 함수로 증가한다고 계산했나? (원전 확인)
- Brooks의 논증은 신참 훈련 비용과 작업이 순차적으로 분할되지 않는 어려움도 근거로 들었는데, 이 두 근거 각각이 에이전트를 추가로 투입하는 상황에는 어떻게 적용되거나 적용되지 않는가? (원전 확인)
- 여러 에이전트를 병렬로 동시에 돌려 같은 프로젝트의 다른 부분을 각각 맡기는 방식이 실제로 전체 완료 시간을 줄이는지, 아니면 사람의 조정 비용으로 상쇄되는지에 대한 사례나 보고가 있는가? (재발견 사례 수집)
- Brooks의 법칙이 성립하는 핵심 메커니즘이 사람 사이의 의사소통 오버헤드라면, 에이전트 사이에는 이에 대응하는 오버헤드가 존재하는가, 존재한다면 무엇으로 측정되는가(토큰 비용, 컨텍스트 공유 실패, 충돌하는 변경 등)?
- Brooks의 법칙이 에이전트에는 적용되지 않는다거나 반대로 그대로 적용된다고 명시적으로 주장하는 실무자나 연구자의 글이 존재하는가? (재발견 사례 수집)
- Brooks가 같은 책에서 함께 제시한 외과 수술팀 모델과 개념적 무결성의 중요성은 에이전트 오케스트레이션 논의에서 어떻게 재조명되는가?

조사 포인터

- Brooks, “The Mythical Man-Month”(1975) 원문(해당 장 확인)
- 멀티에이전트 병렬 코딩 운용에 관한 실무자 글, 도구 문서(병렬 서브에이전트, 여러 워크트리 동시 운용 등)
- 이 책 §8.2(에이전트 해부학)와의 접점, 역할 중복 피할 것

12 도구와 관행의 재발견

버전 관리, 지속 통합, 문서화, 오픈소스 협업. 사람들의 협업을 위해 만들어진 도구와 관행이 에이전트가 끼어든 루프에서 새 역할을 얻는다. 이 장은 그 역할 전환을 다룬다.

12.1 버전 관리: 이력에서 안전망으로

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Git이 협업 도구로 만들어졌던 역사적 배경과, 에이전트 코딩에서 그 쓰임이 커밋 단위의 실험 되돌리기로 이동하는 현상을 대조하는 꼭지다. Git의 출현 연도나 창시 서사는 확정된 것만 쓰고 나머지는 조사 지시로 넘긴다. 3단 구조로 쓴다. 먼저 버전 관리 시스템의 역사, 특히 분산 모델로의 전환이 협업 문제를 해결하기 위한 것이었다는 통상적 서사를 검증 가능한 만큼만 확인한다. 다음으로 에이전트 워크플로에서 커밋과 브랜치와 워크트리가 안전망으로 쓰이는 최근 관행을 사례로 모은다. 마지막으로 여러 사람의 병렬 작업을 조정하는 도구에서 한 사람이 에이전트의 실험을 되돌리는 도구로 쓰임이 이동했을 때 Git 설계의 어떤 속성이 여전히 유효하고 어떤 속성이 과잉인지 논증한다.

시드 질문

- Git이 만들어진 역사적 배경에 대해 확정적으로 말할 수 있는 사실은 무엇이고, 정확한 연도나 세부 서사 중 확인이 필요한 부분은 무엇인가? (원전 확인, 연도는 함부로 단정하지 말 것)
- 버전 관리 시스템이 애초에 해결하려 한 문제, 즉 여러 사람의 동시 편집 충돌과 이력 추적은 에이전트 코딩에서 실제로 관찰되는 쓰임, 즉 에이전트가 코드를 망쳤을 때 되돌리기와 얼마나 겹치는가?
- 에이전트에게 자주 커밋하게 하라거나 체크포인트마다 커밋을 만들라는 실무 조언이 도구 문서나 실무자 글에서 언제, 어떤 형태로 등장했는가? (재발견 사례 수집)
- 여러 에이전트가 동시에 다른 워크트리에서 작업하는 관행은 Git이 원래 지원하려 한 여러 사람의 병렬 작업과, 한 사람이 여러 에이전트의 병렬 작업을 조정하는 것 사이에서 어느 쪽에 더 가까운가?
- 커밋 메시지나 리뷰용 diff처럼 사람이 사람에게 설명하기 위한 Git의 협업 관행이 에이전트가 커밋을 생성하는 상황에서는 어떤 형태로 바뀌는가, 사람 독자를 여전히 전제하는가 아니면 다른 에이전트나 자기 자신을 위한 기록으로 바뀌는가?
- 버전 관리 이전 시대의 문제, 즉 공유 파일이나 잠금 기반 시스템의 충돌 문제가 에이전트의 통제되지 않은 대량 수정에서 재현되는가? (재발견 사례 수집)

조사 포인터

- Git 공식 역사 문서, 초기 발표 자료(정확한 날짜와 경위는 확인할 것, 기억으로 단정 금지)
- 에이전트 코딩 도구의 커밋과 체크포인트 관련 공식 문서
- git worktree, 브랜치 전략에 관한 실무자 글

- 이 책 §5.5(되돌릴 수 있게)와 역할 분담. 그 꼭지는 실무 기법에, 이 꼭지는 도구의 역사적 쓰임 변화에 집중

12.2 CI/CD와 게이트: 사람 없는 루프의 검문소

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 지속적 통합의 원래 논거를 사람 리뷰어가 매 커밋을 볼 수 없는 에이전트 생성 코드 시대에 CI 게이트가 사실상 유일한 자동 검문소가 되는 현상과 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 CI/CD의 원래 논거, 즉 빈번한 통합이 통합 지옥을 방지한다는 관행과 배포 파이프라인 자동화가 조직 성과와 상관관계를 갖는다는 실증 결과를 정확히 읽는다. 다음으로 에이전트가 생성한 코드에 대해 CI를 최후 방어선으로 쓰는 실무 관행을 사례로 모은다. 마지막으로 사람이 커밋하고 사람이 실패를 해석한다는 CI의 전제가 에이전트가 커밋하고 실패도 스스로 해석하는 루프로 바뀌었을 때 무엇이 달라지는지 논증한다.

시드 질문

- Fowler가 CI에 관한 글에서 제시한 핵심 관행, 즉 하루에 여러 번 통합하고 자동화된 빌드로 통합을 검증하는 관행의 정확한 내용은 무엇인가? (원전 확인)
- Humble과 Farley가 “Continuous Delivery”(2010)에서 제시한 배포 파이프라인 개념과, Forsgren 등이 “Accelerate”(2018)에서 실증한 배포 빈도와 안정성의 상관관계는 핵심적으로 무엇을 보였나? (원전 확인)
- 에이전트가 코드를 작성하고 커밋하고 CI가 실패하면 로그를 읽고 스스로 고치는 워크플로가 실무에서 얼마나 정착되어 있는가, 이를 명시적으로 다루는 도구 문서나 사례가 있는가? (재발견 사례 수집)
- CI가 원래 전제한 테스트 실패가 사람에게 신호를 보내 사람이 원인을 진단하는 구조에서, 에이전트가 신호도 받고 진단도 하는 구조로 바뀌었을 때 CI 게이트의 설계, 즉 무엇을 테스트할지와 얼마나 엄격하게 막을지는 어떻게 달라져야 하는가?
- 높은 성과 조직은 배포 빈도가 높다는 Accelerate의 결과가 에이전트 생성 코드로 배포 빈도가 크게 높아진 상황에서도 같은 방향으로 성립하는가, 아니면 배포 빈도만 높고 안정성은 떨어지는 역전이 관찰되는가? (재발견 사례 수집)
- CI 게이트를 우회하거나 통과만을 목표로 코드를 조작하는 에이전트의 실패 사례가 보고된 적이 있는가?

조사 포인터

- Martin Fowler의 CI 관련 글(원제와 요지 확인)
- Humble·Farley, “Continuous Delivery”(2010) 원문
- Forsgren·Humble·Kim, “Accelerate”(2018) 원문의 핵심 지표(배포 빈도, 리드타임, 변경 실패율, 복구 시간)
- 이 책 §8.3(검증 비대칭), §5.4(검증 가능한 단위 만들기)와의 접점, 역할 중복 피할 것

12.3 문서화: 리터레이트 프로그래밍에서 CLAUDE.md까지

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Knuth(1984)의 리터레이트 프로그래밍이 제시한 이상, 즉 코드를 사람이 읽는 설명적 산문과 함께 짜야 한다는 논거를 AGENTS.md와 CLAUDE.md류 파일이 만드는 새로운 문서화 장르와 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 Knuth의 이상, 즉 WEB 시스템과 프로그램은 사람에게 설명하기 위해 쓰여야 한다는 원문의 취지를 정확히 읽는다. 다음으로 AGENTS.md와 CLAUDE.md라는 새 관행이 언제 어떻게 표준화되었는지 사례를 모은다. 마지막으로 문서의 독자가 사람에서 AI로 바뀌었을 때 Knuth가 강조한 설명의 서사성과 코드와 산문의 통합 중 무엇이 남고 무엇이 무의미해지는지 논증한다.

시드 질문

- Knuth가 “Literate Programming”(1984)에서 제시한 핵심 주장, 즉 프로그램을 사람에게 설명하는 문학 작품으로 취급해야 한다는 논거의 정확한 표현과 근거는 무엇인가? (원전 확인)
- Knuth의 WEB 시스템이 실제로 어떻게 작동했는지, 코드와 설명 산문을 어떤 순서로 섞어 짜고 사람이 읽는 문서와 컴파일 가능한 코드라는 두 산출물로 어떻게 나뉘어 나오는지 정확히 확인할 것 (원전 확인)
- AGENTS.md, CLAUDE.md 같은 파일이 AI 에이전트가 읽는 프로젝트 설명서로 정착한 것은 언제부터, 어떤 도구나 관행에서 시작되었는가? (재발견 사례 수집, 확인 안 되면 조사 지시로 남길 것)
- Knuth의 리터레이트 프로그래밍은 코드와 설명을 물리적으로 한 파일에 섞어 짜는 것이 핵심이었는데, AGENTS.md류는 코드와 분리된 별도 파일이다. 이 물리적 분리가 서사의 순서가 코드의 실행 순서를 따를 필요가 없다는 Knuth의 논거와 양립하는가, 아니면 정반대의 설계인가?
- 리터레이트 프로그래밍이 실무에서 널리 채택되지 못한 이유로 흔히 꼽히는 것들(도구 부족, 유지보수 부담 등)이 AGENTS.md류 문서에서는 어떻게 회피되거나 똑같이 재현되는가?
- AI가 스스로 자신이 읽을 AGENTS.md를 생성하거나 갱신하는 관행이 있다면, 이것은 사람이 사람을 위해 설명한다는 리터레이트 프로그래밍의 저자성 전제를 어떻게 바꾸는가?

조사 포인터

- Knuth, “Literate Programming”(1984), The Computer Journal 게재 원문
- AGENTS.md 표준화 관련 공지나 발표(여러 도구가 공통으로 채택하게 된 경위, 정확한 시점은 확인할 것)
- Claude Code, Cursor 등의 CLAUDE.md·AGENTS.md 공식 문서
- 이 책 §5.3(컨텍스트 관리의 기술)과 역할 분담. 그 꼭지는 실무 기법에, 이 꼭지는 문서화 장르의 역사적 계보에 집중

12.4 오픈소스 협업 모델: 성당, 시장, 에이전트

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. Raymond의 “The Cathedral and the Bazaar”(에세이 1997, 책 1999)가 제시한 성당 모델과 시장 모델의 대비, 그리고 충분히 많은 눈이 있으면 모든 버그는 얕다는 명제를 AI 기여자가 오픈소스 프로젝트에 대량으로 등장하는 최근 현상과 대조하는 꼭지다. 3단 구조로 쓴다. 먼저 Raymond의 원래 논거, 즉 리누스의 법칙과 초기 공개와 빈번한 릴리스와 사용자를 공동 개발자로 대우하기를 정확히 읽는다. 다음으로 AI가 생성한 이슈 트리아지나 PR이 오픈소스 프로젝트의 리뷰 병목과 신뢰 문제를 어떻게 바꾸는지 사례를 모은다. 마지막으로 눈이 많으면 버그가 얕다는 명제가 그 눈이 사람이 아니라 AI일 때도 성립하는지, 신뢰라는 시장 모델의 숨은 전제가 AI 기여자 앞에서 무너지는지 논증한다.

시드 질문

- Raymond가 “The Cathedral and the Bazaar”에서 제시한 리누스의 법칙, 즉 충분히 많은 눈이 있으면 모든 버그는 얕다는 명제의 정확한 원문 표현과, 이를 뒷받침하기 위해 든 근거는 무엇이었나? (원전 확인)
- Raymond가 시장 모델의 성립 조건으로 제시한 것들, 즉 잦은 릴리스와 사용자를 공동 개발자로 대우하기와 프로젝트 리더의 조정 역할은 정확히 무엇이었나, 그중 이것 없이는 시장 모델이 작동하지 않는다고 명시한 전제 조건이 있었는가? (원전 확인)
- 대량의 AI 생성 PR이나 이슈가 오픈소스 프로젝트에 쏟아지는 현상이 실제로 보고된 사례나 메인테이너들의 공개 대응 정책이 있는가? (재발견 사례 수집)
- 충분히 많은 눈이 사람이 아니라 AI 리뷰어나 AI 기여자로 채워질 때, Raymond가 전제한 눈의 자격, 즉 신뢰할 만한 판단력과 프로젝트에 대한 이해가 여전히 성립하는가, 아니면 눈의 숫자만 늘고 질은 떨어지는 역전이 관찰되는가?
- 오픈소스 메인테이너들이 AI 생성 기여를 다루기 위해 새로 도입한 정책, 예를 들어 기여 가이드라인 변경이나 자동 라벨링이나 AI 기여 공개 의무화의 구체적 사례를 얼마나 찾을 수 있는가? (재발견 사례 수집)
- Raymond의 성당 모델, 즉 소수의 신뢰받는 개발자가 폐쇄적으로 개발하는 방식이 AI 시대에 오히려 재평가받는 흐름이 있는가, 있다면 시장 모델의 위기에 대한 반작용으로 볼 수 있는가?

조사 포인터

- Raymond, “The Cathedral and the Bazaar”(에세이 1997, 책 1999) 원문
- 주요 오픈소스 프로젝트의 AI 기여 관련 공개 정책 문서, 메인테이너 블로그
- 이 책 9장(한계와 리스크) 관련 꼭지, §12.3(문서화)과의 접점
- AI slop 관련 논의는 개념 사전 G13(slop/workslop)과 역할 분담, 이 꼭지는 오픈소스 협업 모델 자체에 집중

제5부 · 공부와 일

13 공부의 재구성

배워야 할 것이 바뀌면 배우는 방법도 바뀐다. 이 장은 커리큘럼 논쟁, 교육 현장의 실험들, 그리고 전문성이 자라는 경로의 변형을 다룬다. 이 수업 자체가 하나의 실험 사례다.

13.1 무엇을 배워야 하는가

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI가 문법과 상용구를 대신 써주는 시대에 무엇을 가르치고 배워야 하는가. 이 꼭지는 프로그래밍 교육의 무게중심이 문법 암기에서 문제 정의, 분해, 검증 능력으로 옮겨가고 있다는 주장을 검토하고, 실제 커리큘럼 논쟁에서 어떤 입장들이 맞서는지 정리한다. 결론을 미리 내리지 않고 각 입장의 근거를 인터뷰와 1차 자료로 확인하는 것이 이 꼭지의 임무다. 5부 전체의 도입에 해당하며, 이 수업 자체도 이 논쟁의 한 실험적 답이라는 점을 염두에 둔다.

시드 질문

- "문법 암기는 이제 중요하지 않다"는 주장과 "기초 문법 없이는 검증도 못 한다"는 반론은 각각 어떤 근거를 대는가?
- 문제 정의, 분해, 검증 능력을 명시적으로 가르치려는 커리큘럼 시도는 실제로 존재하는가, 어떤 형태를 띠는가?
- 협업, 신뢰 형성, 타인의 관점 추론, 불확실성 대응 같은 인간 역량은 별도 교과 내용으로 가르쳐야 하는가, 아니면 팀 과제와 토론 같은 교수법으로 길러야 하는가?
- 데이터사이언스 교육과 전통적 컴퓨터공학 교육에서 이 논쟁의 양상은 어떻게 다른가?
- 교육자들은 AI 시대에 무엇을 가르쳐야 하는가에 대해 실제로 얼마나 합의하고 있는가, 의견이 크게 갈리는 지점은 어디인가?
- 이 수업의 커리큘럼 설계(세미나형, 책 공동 집필) 자체는 이 논쟁에서 어떤 입장을 취하고 있는가?

조사 포인터

- 대학·부트캠프의 커리큘럼 개편 사례를 1차 문서(강의계획서, 학과 발표)로 확인할 것
- David Deming의 The Value of Human Skills in an AI Economy 웨비나(2026-07-28)를 출발점으로, 사회적 학습의 비교우위와 인간 역량을 교수법으로 다뤄야 한다는 주장을 원 논문과 후속 연구로 검증할 것
- 프로그래밍 교육 연구자, 현직 교수·강사를 인터뷰해 1차 자료로 수집할 것
- §13.2(교육 현장의 실험들)와 역할을 나눌 것: 이 꼭지는 "무엇을" 가르칠 것인가의 논쟁, §13.2는 실제 실험 사례

13.2 교육 현장의 실험들

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI 시대 교육 현장은 대체로 세 갈래로 갈렸다. AI 사용을 금지하는 쪽, 필수로 쓰게 하는 쪽, 평가 방식 자체를 바꾸는 쪽이다. 이 꼭지는 각 접근을 택한 구체적 사례를 찾아 실제로 어떤 결과를 냈는지 비교한다. 이 수업 자체(세미나형, 책 공동 집필, AI 전면 허용)도 하나의 실험 사례로 포함해 자기 기록하는 것이 이 꼭지의 특수한 임무다. 인터뷰와 1차 데이터 수집이 핵심 방법이다.

시드 질문

- AI 사용을 금지한 교육 기관·강좌의 사례는 무엇이 있고, 금지가 실제로 지켜졌는가?
- AI 사용을 필수화하거나 전면 허용한 사례는 무엇이 있고, 평가 방식은 어떻게 바뀌었는가?
- 평가 방식 자체를 개편한 사례(구술시험 부활, 과정 평가 강화 등)는 어떤 결과를 보고했는가?
- 이 세 갈래 접근을 비교한 교육학 연구나 대규모 서베이는 존재하는가?
- 이 수업의 설계와 진행 과정 자체를 실험 사례로 어떻게 기록할 것인가, 수강생 인터뷰로 무엇을 확인할 수 있는가?

조사 포인터

- 국내외 대학·고등학교의 AI 정책 공식 발표문을 1차 출처로 수집할 것
- 교육학 학술지의 실증 연구를 우선 검색할 것
- 이 수업의 강의계획서, 운영 기록, 수강생 인터뷰를 직접 1차 자료로 수집할 것

13.3 전문성의 사다리: 초보자는 어떻게 전문가가 되는가

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. 전문가는 쉬운 일을 반복하며 실력을 쌓는다는 것이 의도적 수련(deliberate practice) 이론의 기본 전제다. AI가 그 쉬운 일을 대신 해준다면 이 사다리는 어떻게 되는가. 이 꼭지는 의도적 수련 이론의 핵심 주장을 정리하고, AI 보조 환경에서 초보자가 실제로 어떻게 학습하는지에 관한 연구와 관찰을 찾아 충돌 지점을 논증한다. §9.4(주니어의 역설)와 짝을 이루되 이 꼭지는 학습 이론 쪽에 무게를 둔다. 경력이 다른 개발자를 직접 인터뷰하는 것이 중요한 방법이다.

시드 질문

- 의도적 수련 이론이 요구하는 조건(즉각적 피드백, 난이도의 점진적 상승, 반복)은 AI 보조 학습 환경에서 충족되는가, 훼손되는가?
- AI 도구를 쓰며 배운 초보자와 쓰지 않고 배운 초보자의 실력 습득 속도와 질을 비교한 연구가 있는가?
- 같은 AI 도구를 쓰더라도 처음부터 보조받는 방식과 먼저 스스로 풀고 기준선을 만든 뒤 피드백에 쓰는 방식은 장기 학습 성과가 다른가?
- “쉬운 일을 AI에 위임하면 검증 능력만 남고 생성 능력은 퇴화한다”는 우려는 근거가 있는가?
- 전문가들은 자신의 학습 경로를 돌이켜볼 때 AI가 있었다면 무엇이 달라졌을 것이라고 말하는가?
- 의도적 수련 이론 자체에 대한 최근의 학술적 비판이나 수정 논의는 무엇이 있는가?

조사 포인터

- 의도적 수련 관련 원 연구와 후속 비판 문헌을 학술 데이터베이스에서 확인할 것
- AI 보조 학습에 관한 교육심리학·컴퓨터과학교육 실증 연구를 우선 검색할 것
- David Deming의 The Value of Human Skills in an AI Economy 웨비나(2026-07-28)에서 제시한 "먼저 생산적 어려움을 겪고 이후 AI를 사고 파트너로 쓰라"는 주장을 검증할 실험 연구를 추적할 것
- 경력 3년 미만과 10년 이상 개발자를 각각 인터뷰해 1차 자료로 비교할 것

14 일의 재구성

바이브 코딩은 코딩 이야기로 시작했지만 끝은 일의 이야기다. AI와 함께 일하는 사람의 하루, 직업 지형의 실제 변화, 그리고 "AI가 다 해주면 나는 무엇을 하는 사람인가"라는 질문까지. 예측이 아니라 이미 일어난 변화의 관찰에서 출발한다.

14.1 AI와 일하는 하루

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI를 깊게 쓰는 실무자의 하루는 코드를 직접 치는 시간보다 위임하고 검토하고 여러 작업을 병렬로 굴리는 시간으로 재구성되고 있다. 이 꼭지는 예측이 아니라 관찰에서 출발한다. AI와 깊게 일하는 사람들을 직접 인터뷰해 하루가 어떻게 바뀌었는지 1차 자료로 기록하는 것이 이 꼭지의 핵심 방법이다. 14장(일의 재구성)의 도입에 해당한다.

시드 질문

- AI를 깊게 쓰는 개발자·연구자의 하루 일과를 시간 단위로 재구성하면 무엇이 눈에 띄게 달라졌는가?
- 위임과 검토의 순환은 실제로 어떻게 굴러가는가, 검토에 드는 시간은 생성에 드는 시간보다 늘었는가 줄었는가?
- 여러 에이전트를 병렬로 굴리는 작업 방식은 실제로 생산성을 높이는가, 관리 부담을 늘리는가?
- 하루 중 AI에 맡기지 않고 반드시 직접 하는 일은 무엇으로 남아 있는가, 그 기준은 무엇인가?
- 이런 변화는 직군(개발자, 연구자, 창업자)에 따라 어떻게 다르게 나타나는가?

조사 포인터

- AI를 깊게 쓰는 실무자·연구자를 직접 섭외해 심층 인터뷰할 것. 이 꼭지의 핵심 자료는 인터뷰다
- 인터뷰 대상은 직군, 경력, 사용 도구가 다르도록 다양화할 것
- 기존에 발표된 "하루 기록" 형태의 1차 에세이나 블로그가 있다면 참고하되 인터뷰를 대체하지 않을 것

14.2 직업 지형의 변화: 데이터로 보기

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI가 개발자 직업 지형을 바꾸고 있다는 주장은 많지만 근거의 질은 제각각이다. 이 꼭지는 채용 공고, 임금, 고용 규모에 관한 실증 연구를 찾아 과장된 서사와 실제로 확인되는 변화를 가른다. 이 책에서 수치를 인용해도 되는 유일한 방법은 원 데이터와 방법론을 직접 확인하는 것이다. 이 꼭지는 특정 수치나 결론을 미리 정하지 않는다. 담당자는 어떤 수치도 확인 없이 단정하지 말고, 모든 주장을 실증 연구와 원 데이터로 검증한 뒤에만 써야 한다.

시드 질문

- 개발자 채용 공고 수, 요구 스킬의 변화를 추적한 신뢰할 만한 데이터 출처는 어디이고, 그 방법론은 어떠한가?
- 개발자 임금 추이에 관한 통계는 AI 도입과 인과적으로 연결지을 수 있는가, 아니면 상관관계에 그치는가?
- 언론에서 자주 인용되는 채용 감소 관련 수치는 원 출처가 어디이고, 그 방법론은 검증할 만한가?
- 국가별, 산업별로 고용 데이터의 변화 양상은 어떻게 다른가?
- 실증 연구들 사이에 결론이 엇갈리는 지점이 있다면 그 원인은 방법론 차이인가, 관측 시점 차이인가?

조사 포인터

- 노동경제학 학술 논문과 공식 고용 통계(정부·국제기구 발표 원자료)를 1차 출처로 확인할 것
- 채용 플랫폼이 자체 발표하는 리포트는 방법론과 표본의 한계를 반드시 확인하고 명시할 것
- 이 꼭지의 모든 수치는 원 데이터를 직접 확인한 뒤에만 쓸 것. 확인하지 못한 수치는 단정하지 말고 “확인할 것”으로 남길 것

14.3 정체성 질문: 나는 무엇을 하는 사람인가

> ⚠ 집필 상태
>
> 초안 전. 담당·상태는 저장소 루트의 TOPICS.md에서 확인.

브리프. AI가 코드의 상당 부분을 대신 쓴다면, 그 코드를 만든 사람은 스스로를 무엇을 하는 사람이라고 정의하는가. 이 꼭지는 개발자, 연구자, 창작자의 자기 정의가 실제로 어떻게 바뀌고 있는지, 그리고 장인정신이나 숙련이라는 오래된 담론이 이 변화 앞에서 어디로 가는지를 다룬다. 통계가 아니라 당사자의 목소리를 1차 자료로 삼는 것이 이 꼭지의 방법이다. 14장의 마지막 꼭지로 책 전체를 정체성 질문으로 닫는다.

시드 질문

- “나는 코드를 짜는 사람”에서 다른 자기 정의로 옮겨갔다면, 그 새로운 정의(설계자, 검토자, 오케스트레이터 등)는 인터뷰에서 실제로 어떻게 표현되는가?
- 장인정신, 숙련이라는 담론은 AI 보조 제작 앞에서 폐기되는가, 다른 형태로 재정의되는가?

- 이 정체성 변화에 대한 반응은 세대별, 경력별로 어떻게 다르게 나타나는가?
- “만드는 사람”이라는 자기 정의를 그대로 유지하려는 사람들은 무엇을 근거로 그렇게 말하는가?
- 사진, 음악 제작 등 도구 자동화를 먼저 겪은 분야의 정체성 논쟁에서 참고할 유사 사례가 있는가?

조사 포인터

- 개발자·창작자를 대상으로 정체성 변화에 관한 심층 인터뷰를 직접 수행할 것. 이 꼭지의 핵심 자료는 인터뷰다
- 사진, 음악 등 앞서 자동화를 겪은 분야의 정체성 논쟁 문헌을 비교 자료로 확인할 것
- §13.3(전문성의 사다리)과 교차 참조하되 역할을 나눌 것: 이 꼭지는 정체성과 자기 서사, §13.3은 학습 경로

부록

15 개념 사전

바이브 코딩 주변에서 태어나고 진화하는 개념들의 사전이다. 항목마다 정의, 유래, 현재 상태를 적는다. 개념은 계속 나타나고 뜻이 바뀌므로 이 부록은 영원히 미완성이며, 그래서 이 책에서 가장 자주 갱신되는 부분이다. 항목 하나가 곧 기여 단위다. 새 항목 추가와 담당은 저장소의 TOPICS.md에서 한다.

> 항목 형식
>
> 정의 두어 문장, 유래(누가 언제), 현재 상태. 유래의 날짜와 인물은 1차 출처를 확인하고 링크를 단다. 확인하지 못한 것은 “유래 확인 필요”로 표시한다.

15.1 바이브 코딩 (vibe coding)

자연어로 AI 코딩 에이전트에게 지시해 소프트웨어를 만드는 방식이다. 좁은 뜻으로는 생성된 코드를 직접 읽지 않고 실행 결과만 보며 반복하는 작업 태도를 가리키고, 넓은 뜻으로는 AI 보조 개발 전반을 가리킨다.

Andrej Karpathy가 2025년 2월 트윗에서 만든 말이다. 원문에는 “fully give in to the vibes, embrace exponentials, and forget that the code even exists”라는 표현이 들어 있다 (원 트윗). Karpathy는 2026년 2월 1주년 회고 트윗에서 이 말을 “shower of thoughts throwaway tweet”이라고 돌아봤다 (회고 트윗).

2025년 11월 Collins 사전 올해의 단어로 선정되며 일반어가 됐다 (발표, 보도). Simon Willison은 “코드를 읽지 않는 방식”과 “AI 보조 개발 전반”을 구분하자고 제안했지만 (글), 실제 사용에서는 두 뜻이 섞여 쓰인다.

확장 포인터

- Willison 글 전문을 읽고 그가 구분한 스펙트럼을 정리할 것
- Karpathy 1주년 회고 트윗 전문에서 원 트윗 당시 맥락(어떤 모델·도구를 쓰고 있었는지) 확인할 것
- 위키피디아 개관 문서(링크)와 대조해 뜻의 표류 사례를 추가로 모을 것

15.2 프롬프트 엔지니어링 (prompt engineering)

원하는 출력을 얻기 위해 모델에 주는 입력, 즉 프롬프트를 설계하는 기술이다. few-shot 예시 구성, chain-of-thought 유도 등 특정 기법을 발견하고 체계화하는 작업을 포함한다.

GPT-3 공개(2020) 이후 한동안 하나의 직무처럼 다뤄졌다. 정확히 누가 언제 이 말을 직무 명칭으로 굳혔는지는 확인이 필요하다.

모델 성능이 좋아지고 상호작용이 대화형, 에이전트형으로 바뀌면서 독립 기술로서의 위상은 흐려졌다. 이 책의 관점에서는 바이브 코딩의 전사(前史)에 해당한다.

확장 포인터

- “프롬프트 엔지니어”라는 직무명이 언제 등장하고 채용 공고에서 언제 사라졌는지 시계열 확인
- few-shot, chain-of-thought 등 대표 기법의 원 논문 확인
- 직무 소멸을 보도한 기사 수집 (§1.3 섹션과 연계)

15.3 컨텍스트 엔지니어링 (context engineering)

프롬프트 한 줄이 아니라 모델이 보는 컨텍스트 전체, 즉 파일, 문서, 대화 이력, 도구 출력을 설계하는 기술이다. 프롬프트 엔지니어링의 병목이 “말을 어떻게 하나”에서 “무엇을 보여주나”로 이동했음을 반영한다.

Tobi Lütke(Shopify CEO)가 2025년 6월 19일 트윗으로 이름을 붙였고 Karpathy가 이를 승인하는 트윗을 남겼다 (Karpathy 트윗). Simon Willison과 Phil Schmid가 각각 개념을 정리하는 글을 썼다 (Willison, Schmid).

2025년 하반기부터 널리 쓰이기 시작했고, 현재는 CLAUDE.md, AGENTS.md 같은 컨텍스트 파일 설계 관행을 가리키는 실무 용어로 자리 잡았다.

확장 포인터

- Lütke 원 트윗 전문과 스레드 반응 확인
- Willison·Schmid 두 정리 글의 정의 차이 비교
- §3.2, §5.3 섹션에서 다루는 실무 사례와 교차 확인

15.4 에이전틱 코딩 (agentic coding)

AI가 한 번 답하고 끝나는 것이 아니라 도구를 호출하고 코드를 실행하고 결과를 확인한 뒤 다음 행동을 정하는 루프를 돌며 작업하는 방식이다. 바이브 코딩을 실제로 가능하게 만든 기술 형태다.

"에이전틱 코딩"이라는 표현을 누가 언제 처음 썼는지는 확인이 필요하다. 다만 이 루프 구조 자체를 정리한 참고 문헌으로 Anthropic의 "Building Effective Agents"(2024년 12월)가 있다 (링크).

CLI형 코딩 에이전트(Claude Code 등)와 IDE 통합형 에이전트가 이 방식을 구현하는 대표 도구로 꼽힌다. §2.3, §8.2 섹션에서 도구 계보와 루프 구조를 각각 다룬다.

> ℹ 확장 포인터
>
> - "agentic coding" 용어의 최초 사용례 확인
> - Anthropic "Building Effective Agents"의 워크플로/에이전트 구분과 이 책의 용법 대조
> - §8.2(코딩 에이전트의 해부학)와 정의 정합성 확인

15.5 CHOP (chat-oriented programming)

채팅 인터페이스로 대화하듯 AI에게 지시해 코드를 만드는 방식을 가리키는 말이다. 바이브 코딩과 인접한 개념으로, 코드를 직접 다루기보다 대화로 작업을 진행한다는 점을 강조한다.

Steve Yegge가 바이브 코딩이라는 말이 나오기 전에 만든 용어다 (O'Reilly 팟캐스트). 정확한 최초 사용 시점은 확인이 필요하다. Yegge는 Gene Kim과 함께 "Vibe Coding"이라는 단행본을 공저했는데, 정확한 출간 정보는 확인이 필요하다.

바이브 코딩이라는 말이 대중화된 뒤로는 CHOP이 더 이전 세대 용어, 또는 바이브 코딩의 동의어에 가깝게 쓰이는 경우가 많다.

> ℹ 확장 포인터
>
> - CHOP 최초 사용 시점과 맥락(어떤 글, 어떤 발표)을 원 출처에서 확인
> - Kim·Yegge 공저 "Vibe Coding" 출간일, 출판사 확인
> - CHOP과 바이브 코딩을 같은 말로 쓰는 사례와 구분해 쓰는 사례를 각각 수집 (§3.1과 연계)

15.6 바이브 엔지니어링 (vibe engineering)

바이브 코딩의 속도를 유지하면서도 결과물의 책임을 지는 작업 방식을 가리키는 말이다. 검증, 테스트, 리뷰 같은 공학적 절차를 바이브 코딩 워크플로에 다시 결합하자는 제안에 가깝다.

Simon Willison이 제안한 개념이다 (글). 2025년 가을로 알려져 있으나 정확한 발행일은 확인이 필요하다.

바이브 코딩의 "검증 문제"에 대한 반작용 개념 중 하나로, §3.5에서 이런 반작용 개념들을 함께 다룬다.

확장 포인터

- Willison 글의 정확한 발행일과 정의 원문 확인
- vibe engineering과 spec-driven development, TDD 부활론(§ 10.4) 등 인접 반작용 개념과의 관계 정리
- Willison 이후 이 용어를 쓴 다른 저자·기사 존재 여부 확인

15.7 루프 엔지니어링 (loop engineering)

프롬프트를 사람이 매번 치는 대신, 에이전트가 일을 찾고 수행하고 검증하고 기록하는 루프 자체를 설계하는 기술이다. 프롬프트 엔지니어링의 후계 개념으로 이야기된다.

2026년 6월 Addy Osmani의 정리 글을 계기로 널리 퍼졌다 (보도).

이 책의 관점에서는 바이브 코딩의 다음 장에 해당하는 개념이다. § 3.4에서 하네스 엔지니어링과 함께 다룬다.

확장 포인터

- Osmani 원문 글(정리 글의 원 출처) 찾아 인용 확인
- loop engineering과 agent harness 개념이 어디서 겹치고 어디서 갈리는지 정리
- 이 용어를 쓰는 실제 도구·워크플로 사례 수집 (§ 3.4와 연계)

15.8 에이전트 하네스 (agent harness / harness engineering)

에이전트를 실제로 작동하게 만드는 실행 환경, 즉 도구 접근, 권한, 루프 제어, 컨텍스트 관리 장치 전체를 가리키는 말이다. “Agent = Model + Harness”라는 정식화에서, 모델의 성능과 별개로 하네스 설계가 에이전트 품질을 좌우한다는 관점을 담는다.

프롬프트 엔지니어링에서 컨텍스트 엔지니어링을 거쳐 하네스 엔지니어링으로 이어지는 3단계론으로 정리된 바 있다 (Faros AI 블로그). 이 정리 글 이전에 누가 “하네스”라는 말을 에이전트 실행 환경에 처음 썼는지는 확인이 필요하다.

Claude Code 같은 CLI형 코딩 에이전트 도구가 하네스의 대표 사례로 꼽힌다. § 3.4, § 8.2 섹션과 겹친다.

확장 포인터

- “harness”라는 말이 에이전트 문맥에서 쓰이기 시작한 최초 시점 확인
- Faros AI 정리 글의 3단계론(프롬프트→컨텍스트→하네스)을 원문에서 확인하고 이 책 § 3.4와 대조
- 오픈소스 에이전트 하네스 사례(§ 8.2에서 다루는 소스 읽기 대상)와 연결

15.9 스펙 주도 개발 (spec-driven development)

AI에게 바로 코드를 시키지 않고 먼저 명세, 즉 스펙을 함께 작성한 뒤 그 명세를 기준으로 생성과 검증을 진행하는 방식이다. “코드를 읽지 않는” 바이브 코딩의 검증 문제에 대한 대표적 대응으로 이야기된다.

GitHub이 2025년 9월 Spec Kit을 공개하며 Spec, Plan, Tasks, Implement 4단계 흐름을 제시했다 (GitHub 블로그).

폭포수 개발의 명세 우선 원칙이 AI 시대에 재발견된 사례로 볼 수 있다. §3.3, §10.1에서 각각 개념 확산과 폭포수 논쟁 재상연을 다룬다.

> **i** 확장 포인터
>
> - Spec Kit 이전에 유사한 "스펙 먼저" 도구·제안이 있었는지 확인
> - Spec Kit의 4단계 흐름과 §10.1이 다루는 폭포수 모델의 단계 구분을 비교
> - Spec Kit 채택 사례나 비판 글 수집

15.10 서브에이전트 (subagent / 멀티에이전트)

메인 에이전트가 특정 작업을 별도의 독립된 에이전트 인스턴스에 위임하는 구조, 또는 그렇게 위임받아 실행되는 개별 에이전트를 가리키는 말이다. 위임받은 서브에이전트는 자신의 컨텍스트 창과 도구 접근 범위를 따로 가지며, 작업을 마치면 결과만 메인 에이전트에 반환하는 방식이 일반적이다. 여러 서브에이전트를 동시에 또는 순차로 조율하는 구조를 멀티에이전트 시스템이라 부른다.

용어의 정확한 유래는 확인이 필요하다.

컨텍스트 창을 아끼면서 작업을 병렬화하는 수단으로 여러 코딩 에이전트 도구가 이 구조를 채택하고 있다. §8.2에서 다루는 에이전트 해부학의 구성 요소 중 하나다.

> **i** 확장 포인터
>
> - 서브에이전트 개념을 처음 제품화한 도구와 시점 확인
> - 서브에이전트 간 컨텍스트 격리 방식이 도구마다 어떻게 다른지 비교
> - §8.2(코딩 에이전트의 해부학)와 정의 정합성 확인

15.11 휴먼 인 더 루프 (human-in-the-loop)

자동화된 시스템의 실행 과정에 사람의 확인이나 개입 지점을 명시적으로 두는 설계를 가리키는 말이다. AI 코딩 에이전트 맥락에서는 파일 수정, 명령 실행, 배포처럼 되돌리기 어려운 행동 전에 사람의 승인을 받도록 하는 장치를 뜻한다.

AI 코딩 이전부터 자동화·제어 시스템 분야에서 쓰이던 일반 용어이며, 정확한 최초 유래는 확인이 필요하다.

코딩 에이전트 도구에서는 권한 프롬프트, 계획 모드(plan mode), 변경 사항 승인 UI 등의 형태로 구현된다. §5.6에서 다루는 위임의 한계선 논의와 직결된다.

> **i** 확장 포인터
>
> - 자동화·제어 분야에서 이 용어가 쓰인 최초 문헌 확인
> - 코딩 에이전트별 human-in-the-loop 구현 방식(승인 UI, plan mode 등) 비교
> - §5.6(언제 멈추고 직접 읽어야 하는가)과의 관계 정리

15.12 환각 (hallucination)

언어 모델이 사실이 아니거나 존재하지 않는 내용을 그럴듯하게 생성하는 현상을 가리키는 말이다. 코딩 맥락에서는 존재하지 않는 함수, 라이브러리, API를 실재하는 것처럼 생성하는 경우가 대표적이다.

머신러닝 분야에서 먼저 쓰이던 말이 대형 언어 모델로 옮겨온 것으로 알려져 있으나, 정확한 최초 용례와 시점은 확인이 필요하다.

코딩 에이전트가 존재하지 않는 패키지명을 생성하는 경우가 공급망 공격으로 이어질 수 있다는 점에서 슬롭스쿼팅(G14)과 직결된다. §9.2에서 보안 맥락으로 다룬다.

> ℹ 확장 포인터
>
> - “hallucination”이 언어 모델 오류를 가리키는 말로 쓰이기 시작한 최초 논문·글 확인
> - 코드 생성 맥락에서의 환각 실증 연구(존재하지 않는 패키지 생성 비율 등) 수집
> - §9.2(보안: 새로운 공격면)와 슬롭스쿼팅 항목을 교차 확인

15.13 슬롭 (slop / workslop)

품질 검토 없이 대량으로 생성된, 낮은 품질의 AI 산출물을 가리키는 말이다. workslop은 그 중에서도 업무 맥락, 즉 보고서나 문서, 코드처럼 동료나 조직이 소비해야 하는 산출물에 한정해 쓰이는 표현이다.

유래 확인 필요. 누가 언제 이 말을 만들었는지, workslop이라는 변형어가 별도로 언제 등장했는지 모두 원 출처를 찾아 검증해야 한다.

생성 속도가 검토 속도를 앞지르면서 나타난 현상을 가리키는 말로 통용되고 있으며, §9.3(품질 부채와 유지보수)에서 다루는 문제와 연결된다.

> ℹ 확장 포인터
>
> - “slop”과 “workslop” 각각의 최초 용례와 유래를 원 출처에서 확인
> - workslop을 다룬 실증 연구(생산성 손실 추정치 등)가 있는지 확인
> - §9.3과 사실관계 대조

15.14 슬롭스쿼팅 (slopsquatting)

AI 코딩 에이전트가 환각으로 만들어낸, 실재하지 않는 패키지명을 공격자가 미리 등록해 두는 공급망 공격 기법을 가리키는 말이다. 개발자가 에이전트의 제안을 그대로 설치 명령에 넣으면 공격자가 등록한 악성 패키지가 설치된다.

유래 확인 필요. 누가 언제 이 용어를 만들었는지 원 출처를 찾아 검증해야 한다. typosquatting(오타를 노린 패키지명 선점)에서 파생된 조어로 보이나 이 계보 역시 확인이 필요하다.

생성 코드의 신뢰 문제를 보여주는 대표 사례로 보안 논의에서 자주 언급된다. §9.2(보안: 새로운 공격면)에서 다루는 핵심 사례다.

확장 포인터

- slopsquatting 용어의 최초 사용례(보안 연구, 블로그, 논문 등) 확인
- 실제 관측된 slopsquatting 공격 사례와 피해 규모 수집
- §9.2와 사실관계 대조

15.15 MCP (Model Context Protocol)

AI 모델이 외부 도구, 데이터 소스, 서비스와 통신하는 방식을 표준화한 프로토콜이다. 이 표준을 따르는 서버를 구현하면 서로 다른 AI 애플리케이션이 같은 도구 연동 코드를 재사용할 수 있다.

Anthropic이 공개한 프로토콜이다. 정확한 공개 시점은 확인이 필요하다.

코딩 에이전트가 파일 시스템 바깥의 도구(브라우저, 데이터베이스, 사내 시스템 등)에 접근하는 표준 경로로 자리 잡았다. 실무에서는 MCP 서버 대신 직접 API 호출로 전환하는 사례도 나타나고 있다.

확장 포인터

- MCP 공개 정확한 날짜와 최초 발표 자료(Anthropic 공식 블로그·문서) 확인
- MCP 이전에 존재하던 도구 연동 방식(함수 호출, 플러그인 등)과의 차이 정리
- MCP 채택 현황(어떤 회사·도구가 지원하는지) 수집

15.16 RAG (retrieval-augmented generation)

모델이 답을 생성하기 전에 외부 문서나 데이터베이스에서 관련 내용을 검색해 프롬프트에 포함시키는 기법이다. 모델의 학습 데이터에 없는 최신 정보나 비공개 문서를 답변에 반영할 수 있게 한다.

유래 확인 필요. 최초 제안 논문과 저자, 발표 시점을 원 출처에서 확인해야 한다.

코딩 에이전트에서는 사내 코드베이스나 문서를 검색해 컨텍스트에 넣는 용도로 쓰이며, 컨텍스트 엔지니어링(G3)의 구체적 구현 기법 중 하나로 다뤄진다.

확장 포인터

- RAG를 처음 제안한 논문의 저자·발표 시점을 원 출처에서 확인
- 코딩 에이전트 맥락에서의 RAG 활용 사례(코드 검색, 문서 검색) 수집
- 컨텍스트 엔지니어링 항목과의 관계 정리

15.17 토큰과 컨텍스트 윈도 (tokens / context window)

토큰은 언어 모델이 텍스트를 처리하는 최소 단위로, 단어보다 작을 수도 여러 단어를 묶을 수도 있다. 컨텍스트 윈도는 모델이 한 번에 참조할 수 있는 토큰의 최대 개수를 가리킨다.

일반 기술 용어로, 특정 인물이나 시점에 귀속되는 유래는 없다.

코딩 에이전트가 다룰 수 있는 코드베이스의 크기, 대화의 길이, 컨텍스트에 넣을 수 있는 파일 수를 모두 이 한계가 좌우한다. 컨텍스트 윈도가 커질수록 컨텍스트 엔지니어링(G3)의 중요성이 오히려 부각된다는 관찰이 있다. §5.3에서 실무적 관리 기술을 다룬다.

> ℹ 확장 포인터
>
> - 주요 모델별 컨텍스트 윈도 크기 변화 추이(수치는 반드시 공식 문서로 확인)
> - 토큰화 방식(BPE 등)이 코드에 어떻게 특히 적용되는지
> - §5.3(컨텍스트 관리의 기술)과 사실관계 대조

15.18 evals (평가)

AI 모델이나 에이전트의 출력을 정해진 기준에 따라 채점하는 절차, 또는 그 채점에 쓰이는 테스트 집합을 가리키는 말이다. 소프트웨어의 유닛 테스트에 대응하는 개념으로, 모델 성능이나 회귀 여부를 정량적으로 확인하는 데 쓰인다.

일반 기술 용어로, 특정 인물이나 시점에 귀속되는 유래는 없다.

코딩 에이전트 평가에서는 벤치마크 점수뿐 아니라 실제 저장소에서의 작업 성공률을 재는 evals가 함께 쓰인다. §5.4(검증 가능한 단위 만들기)와 맞닿아 있다.

> ℹ 확장 포인터
>
> - 코딩 에이전트용 대표 벤치마크(예: 저장소 기반 작업 성공률 측정)의 방법론 확인
> - evals와 전통적 소프트웨어 테스트의 차이점(비결정성 처리 등) 정리
> - §5.4와 사실관계 대조

15.19 가드레일 (guardrails)

AI 시스템의 입력과 출력에 제약을 걸어 의도하지 않은 행동이나 위험한 결과를 막는 장치를 가리키는 말이다. 프롬프트 필터링, 출력 검증, 실행 권한 제한 등 여러 층위에서 구현된다.

일반 기술 용어로, 특정 인물이나 시점에 귀속되는 유래는 없다.

코딩 에이전트에서는 파일 접근 범위 제한, 위험 명령 차단, 실행 전 승인 요구 같은 형태로 구현되며 휴먼 인 더 루프(G11)와 겹치는 부분이 많다. §9.2, §9.5에서 각각 보안과 책임 소재 맥락으로 다룬다.

> ℹ 확장 포인터
>
> - 코딩 에이전트 도구별 가드레일 구현 방식(권한 시스템, 샌드박스 등) 비교
> - 가드레일 우회 사례(프롬프트 인젝션 등)와의 관계 정리
> - §9.2, §9.5와 사실관계 대조

15.20 소프트웨어 2.0 / 3.0 (Software 2.0 / 3.0)

소프트웨어 2.0은 사람이 직접 규칙을 코드로 짜는 대신 데이터로 신경망 가중치를 학습시켜 프로그램을 만드는 패러다임을 가리키는 말이다. 소프트웨어 3.0은 이 계보를 이어, 자연어 프롬프트로 모델의 행동을 프로그래밍하는 단계를 가리키는 말로 쓰인다.

소프트웨어 2.0은 Andrej Karpathy가 2017년에 쓴 글에서 만든 개념이다 (링크). 소프트웨어 3.0이라는 후속 개념을 누가 언제 정식화했는지는 확인이 필요하다.

소프트웨어 2.0은 딥러닝 전반을 가리키는 말로 정착했고, 소프트웨어 3.0은 바이브 코딩 담론에서 자연어를 프로그램으로 보는 관점을 뒷받침하는 개념으로 쓰이고 있다.

> ℹ 확장 포인터
>
> - “Software 3.0” 개념을 누가 언제 처음 정식화했는지 원 출처 확인 (Karpathy의 이후 발표·글 포함)
> - 소프트웨어 1.0/2.0/3.0 3단계 구분이 처음 어디서 나왔는지, 3.0이 2.0 글에 이미 있었는지 원문 확인
> - §8.1(다음 토큰 예측에서 코드까지)과 개념적으로 어떻게 연결되는지 정리

16 읽기 자료

수업과 집필의 공통 기반이 되는 자료 목록이다. 발표 담당자는 자기 챕터의 자료를 여기에 추가한다. 규칙: 1차 출처를 우선하고, 확인한 날짜를 적고, 요약 한 줄을 단다.

16.1 원전

- Andrej Karpathy, 바이브 코딩 원 트윗 (X, 2025-02). 용어의 탄생. “fully give in to the vibes, embrace exponentials, and forget that the code even exists.”
- Andrej Karpathy, 1주년 회고 트윗 (X, 2026-02). 본인이 돌아본 용어의 확산 과정. “shower of thoughts throwaway tweet.”
- Andrej Karpathy, Software 2.0 (Medium, 2017). 신경망이 코드를 대체한다는 논의의 원류.

16.2 담론

- Simon Willison, Not all AI-assisted programming is vibe coding (but vibe coding rocks) (2025-03-19). 용어의 좁은 뜻과 넓은 뜻을 가른 대표 글.
- Collins Dictionary, Word of the Year 2025 발표 (2025-11). 일반어로의 편입. 관련 보도: CNN.
- CodeRabbit, A semantic history of vibe coding . 용어의 의미 변천 정리.
- Karpathy, 컨텍스트 엔지니어링 지지 트윗 (2025-06). Tobi Lütke의 명명(2025-06-19)을 정식화한 계기. 정리 글: Simon Willison, Phil Schmid.
- GitHub, Spec-driven development with AI: Spec Kit 공개 (2025-09). 바이브의 반작용으로서의 명세 우선 워크플로.
- Simon Willison, Vibe engineering (2025년 가을). 책임지는 가속이라는 재정의 시도.
- ADTmag, Loop Engineering Emerges as Developers Put AI Coding Agents on Repeat (2026-07-01). 루프 엔지니어링 개념의 부상 보도.
- Faros AI, Harness Engineering (2026). “Agent = Model + Harness”, 프롬프트→컨텍스트→하네스 3단계론.
- O’Reilly Radar 팟캐스트, Vibe Coding with Steve Yegge. CHOP(chat-oriented programming)의 명명자 인터뷰. Gene Kim·Steve Yegge의 단행본 “Vibe Coding”도 함께 볼 것(출간 정보 확인 후 추가).

16.3 실무

- Anthropic, Building effective agents (2024-12). 에이전트 설계 원칙의 기준 문서.
- Anthropic, Prompt engineering 공식 가이드. 전사(前史) 쪽 기준 문서.

16.4 교육과 학습

- David Deming, Lucy Swedberg, The Value of Human Skills in an AI Economy (Harvard Business Impact Education 웨비나, 2026-07-28, 2026-07-29 확인). 인간의 비교우위를 빠른 사회적 학습과 전략적 상호작용에서 찾고, 수업에서는 먼저 스스로 씨름한 뒤 AI를 비판과 피드백에 쓰는 순서를 제안한다.

> **i** 수집 대기
>
> 루프 엔지니어링의 원 포스트(Addy Osmani, 2026-06)와 그 원류가 된 실무자 글들, 국내 논의 자료, 각 챕터별 핵심 문헌. 담당자가 챕터 조사와 함께 채울 것. 링크는 반드시 접속 확인 후 추가한다.

17 기여자

이 책은 서울대학교 데이터사이언스 특강 〈바이브 코딩〉에서 함께 쓴다. 판마다 그 학기의 기여자를 기록한다.

17.1 2026년판 (2026년 2학기)

책임

- 박현우 (서울대학교 데이터사이언스대학원) · 기획, 편집 책임

티칭 팀

- (확정 후 기재)

저자 (수강생)

- (챕터 배정 후 기재. 형식: 이름 · 담당 챕터)

> **i** 노트
>
> 수강생은 첫 PR이 머지될 때 자신의 이름을 이 목록에 추가한다. 그것이 이 수업의 통과의례다.