소프트웨어 아키텍처의 역사

경계를 그은 여섯 개의 손

빅히스토리, AI의 혼돈, 그리고 다음 형태

Zhuge Hyuk

소프트웨어 아키텍처의 역사

경계를 그은 여섯 개의 손
빅히스토리, AI의 혼돈, 그리고 다음 형태

Zhuge Hyuk

1차본 · 2026

이 책에 대하여

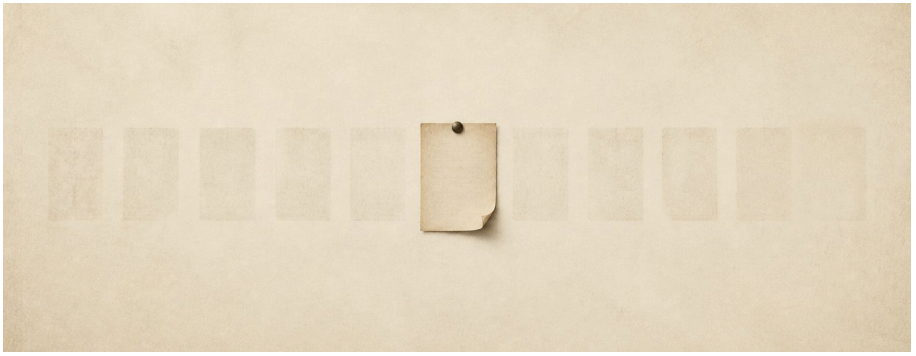
이 책은 아키텍처의 통사가 아니다. 하나의 질문만 따라간다 — 경계를 누가 그었고, 무엇이 그것을 강제했는가.

1부는 역사다. 1964년의 메모 한 장에서 2014년의 쿠버네티스까지, 경계를 긋는 손이 여섯 번 바뀌는 과정을 따라간다. **2부는 혼돈이다.** 지금 조직들이 쪼갠 것을 다시 합치고 있고, 그 되감기의 근거로 가장 많이 인용되는 사례는 잘못 읽힌 것이며, 한 회사는 경계를 지키는 도구를 만들었다가 버렸다. **3부는 예측이다.** 사람의 인지 상한에서 시작해 — 그리고 사람이 그 상한 앞에서 무엇을 했는지를 먼저 보고 — 기계의 상한이 어디서 오는지를 판다. 산수와 메모리와 청구서가 창을 막고, 그 위에서 측정이 시작된다. 결론은 이렇다. 기계가 한 번에 감당할 수 있는 인지부하가 유한하기 때문에, 지금으로서는 최대한 작은 닫힌 계를 만들고 그것을 블랙박스로 조립하는 것 외에 방법이 없다는 것. 다만 나누기는 해법이 아니라 **거래**이고, 통로에 들어가는 것만 나눌 수 있다.

그리고 그 예측에는 날짜가 붙어 있다. 3년쯤 뒤에 이 책의 후반부는 틀린 이야기가 되어 있을지도 모른다. 만료 조건은 부록 A에 적어두었다.

이 책은 실존 인물의 전기적 사실이나 인용을 창작하지 않았다. 장면이 필요한 자리에서는 그들이 남긴 것 — 논문의 첫 문장, 메모의 한 장, 블로그 글의 실토, 벤치마크의 숫자 — 에서 채굴했다. 사실 주장의 검증 등급은 부록 C에 있다.

차례



프롤로그

벽에 붙은 종이 한 장

벨 연구소(Bell Labs)의 어느 사무실 벽에, 누렇게 바랜 종이 한 장이 자석으로 붙어 있었다.

한 장뿐이었다. 원래는 더 길었던 문서의 열 번째 페이지. 데니스 리치(Dennis Ritchie)는 그것을 오래 붙여두었다. 종이 아래쪽에 서명이 있었다. M. D. 맥일로이(Malcolm Douglas McIlroy), 1964년 10월 11일.[1]

벨 연구소 (Bell Labs)

AT&T의 연구 부문으로, 1925년 뉴욕 웨스트가에서 출범해 나중에 뉴저지 머리힐로 본거지를 옮겼다. 트랜지스터, 레이저, 정보이론, 태양전지가 여기서 나왔고 노벨 물리학상이 아홉 번 나갔다. 이 책에 등장하는 이유는 다른 산물 때문이다 — 1969년에 켄 톰슨(Ken Thompson)과 데니스 리치가 여기서 유닉스(Unix)를 만들었고, 그 설계 원칙들이 이후 반세기의 소프트웨어 구조를 규정했다.

데니스 리치 (Dennis Ritchie, 1941-2011)

C 언어를 만들었고 켄 톰슨과 함께 유닉스를 만들었다. 1983년 튜링상. 지금 돌아가는 거의 모든 운영체제가 그의 두 산물 위에 있거나 그것을 베낀 것이다. 이 프롤로그에 나오는 것은 그의 발명이 아니라 그가 **벽에 붙여둔 남의 문서**다.

더그 맥일로이 (Doug McIlroy, 1932-)

벨 연구소 컴퓨팅 기술 연구 부서장. 유닉스 파이프(pipe)의 제안자이자 그것을 관철시킨 사람. 유닉스 철학 — "한 가지를 잘 하는 프로그램을 만들라, 함께 동작하는 프로그램을 만들라, 텍스트 스트림을 다루는 프로그램을 만들라" — 을 정식화한 것도 그다.[2]

그 페이지는 "무엇이 가장 중요한가"를 정리한 요약이었고, 네 항목이 적혀 있었다. 링크-로딩(link-loading)이 가능한 로더. 라이브러리를 정리하는 일반적인 방식. 시스템 구성 요소를 개인적으로 확보할 수 있는 능력. 그리고 첫 번째 항목.

첫 번째 항목은 프로그램을 서로 연결하는 방법에 관한 것이었다. 정원 호스(garden hose)처럼 연결하자는 것이었다 — 다른 방식으로 데이터를 주무를 필요가 생기면, 세그먼트를 하나 더 돌려 끼우는 식으로.

1964년의 벨 연구소에서 컴퓨팅은 대체로 배치 처리(batch processing)였다. IBM 7090과 7094가 카드 뭉치를 삼키고 종이를 뱉었다.[3] 프로그램들이 서로 대화한다는 발상은 그 방에서 당장 만들 수 있는 것이 아니었다. 리치는 나중에, 네 항목 중 첫 번째의 그 비유가 결국 가장 큰 영향을 남겼다고 적었다.

파이프가 실제로 운영체제에 들어간 것은 그로부터 8년쯤 지나서였다. 리치는 1972년이라 적었고, 문서로는 1972년 6월의 유닉스 2판에 없고 1973년 1월 15일 시점에는 있다. 리치는 그 사이의 일에 대해 한 가지를 분명히 했다. 파이프는 맥일로이의 제안 덕만이 아니라, 그의 집요함 덕이었다.

◆

62년 뒤, 다른 종류의 종이 위에 다른 종류의 숫자가 적힌다.

저장소 현대화 벤치마크(RepoMod-Bench)의 결과표다.[4] 실제 오픈소스 프로젝트를 코딩 에이전트(coding agent)에게 주고, 낡은 코드를 현대적인 형태로 바꾸게 한 뒤 테스트를 돌린다. 10만 줄이 넘는 프로젝트들에서 평균 통과율이 20% 아래로 떨어진다. 16만 2천 줄짜리 프로젝트 하나에서는 네 개의 에이전트가 전부 0을 기록한다.

코딩 에이전트 (coding agent)

사람이 자연어로 지시하면 스스로 파일을 읽고 고치고 테스트를 돌리는 프로그램. 2024년 이후 대형 언어 모델(LLM) 위에 도구 사용 능력을 얹어 만들어진다. 이 책의 벤치마크에 나오는 것들은 클로드 코드(Claude Code), 코덱스 CLI(Codex CLI), 오픈코드(OpenCode) 같은 것들이다. 중요한 성질이 하나 있다 — 이들은 코드베이스 전체를 한 번에 읽지 않는다. 필요할 때 파일을 골라 읽고, 맥락이 차면 스스로 압축한다.

숫자보다 눈에 걸리는 것은 그 옆에 붙은 문장이다. 논문의 저자들은 이것이 컨텍스트 윈도우(context window) 문제가 아니라고 적는다. 그 에이전트들은 파일을 필요할 때 골라 읽을 수 있고, 컨텍스트가 차면 자동으로 압축한다. 임의로 큰 코드베이스를 다룰 수 있다. 그런데도 0이다.

저자들이 지목한 어려움은 다른 데 있었다. 수천 개의 서로 얽힌 파일에 걸쳐 **일관된 아키텍처 이해를 유지하는 것.**[5]

> **컨텍스트 윈도우 (context window)**
>
> 언어 모델이 한 번에 입력으로 받을 수 있는 토큰(token)의 최대 개수. 2026년 현재 20만에서 1000만까지 광고된다. 벤더가 파는 숫자이고, 매년 커진다. 이 책이 이 말을 되도록 쓰지 않는 이유는 16장과 18장에 있다 — **담을 수 있는 양**과 **한 번에 관계를 유지하며 판단할 수 있는 양**은 같지 않고, 후자가 훨씬 작다.

◆

이 두 장면 사이에 62년이 있다.

그 사이에 소프트웨어 아키텍처는 여러 번 형태를 바꿨다. 계층으로 나뉘었고, 클라이언트와 서버로 갈렸고, 객체가 되었다가, 네트워크 너머로 흩어졌다가, 다시 하나로 뭉쳤다가, 또 흩어졌다. 진자가 왕복했다고 흔히들 말한다.

그런데 왕복한 것은 형태이고, 매번 바뀐 것은 **이유**다.

1980년대에 경계를 그은 것은 설계자가 아니라 하드웨어였다. 기계가 물리적으로 떨어져 있었기 때문에 코드도 떨어졌다. 1990년대에는 그 경계를 숨기려는 시도가 있었고, 실패했다. 2000년대에 사람들은 경계의 본질이 코드가 아니라 계약이라는 것을 발견했다. 2010년대에는 조직이 경계를 그었다. 한 팀이 통째로 소유할 수 있는 크기가 서비스의 크기가 되었다. 2020년대에는 그 청구서가 돌아왔고, 사람들은 되감기 시작했다.

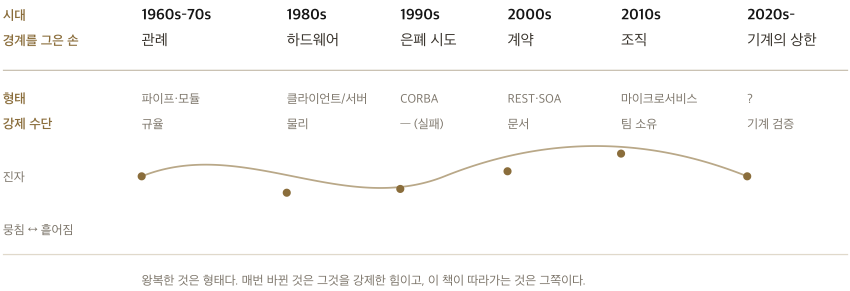


그림 0 — 경계를 그은 손이 여섯 번 바뀐다. 형태는 왕복했지만 이유는 매번 새것이었다.

그리고 지금, 여섯 번째 손이 올라와 있다.

이번에 경계를 긋는 것은 하드웨어도 조직도 아니다. **기계가 한 번에 감당할 수 있는 양**이다.

◆

이 책은 아키텍처의 통사가 아니다. 하나의 질문만 따라간다.

경계를 누가 그었고, 무엇이 그것을 강제했는가.

그 질문을 따라가다 보면 이상한 것들이 눈에 걸린다. 소프트웨어 공학에서 가장 널리 가르쳐지는 체크리스트 중 하나에는 논문이 없다. 업계 전체가 인용하는 여섯 줄짜리 칙령에는 1차 문서가 없다. 조직 구조와 시스템 구조의 관계를 말한 논문은 "증명하지 못했다"는 이유로 반려당했고, 그 증명은 44년 뒤에 왔다.

경계를 명시하라고 가르쳐온 분야가, 정작 자기 규범의 출처는 명시하지 않았다.

그리고 이 책의 마지막 부분은 예측이다. 지금으로서는 다시 잘게 쪼개고 블랙박스로 조립하는 것 외에 방법이 없다는 것. 그 이유가 기계의 인지부하

(cognitive load)라는 것. 그리고 그 진단에 **날짜가 붙어 있다는 것.**

3년쯤 뒤면 이 책의 후반부는 틀린 이야기가 되어 있을지도 모른다. 그것이 이 책이 할 수 있는 가장 정직한 약속이다.

◆

리치의 벽에 붙어 있던 종이는 아이디어와 구현 사이에 8년이 걸린 문서였다. 리치 본인의 회고에 따르면 구현 자체는 간단한 일이었다. 8년을 잡아먹은 것은 그것이 할 만한 일이라는 확신이었고, 퍼지는 데는 그 다음에 또 하나가 필요했다 — 양쪽 프로그램이 그 경계에 맞춰 다시 만들어지는 일.

지금 우리가 다시 그리고 있는 경계에는 무엇이 그렇게 맞춰져야 하는가.

주

1 맥일로이의 1964-10-11 메모. 리치가 소장하던 사본이 노키아 벨 연구소의 유닉스 아카이브에 남아 있다. 원문 이미지: 〈https://www.nokia.com/bell-labs/about/dennis-m-ritchie/mdmpipe.html〉 · 리치의 회고는 "Advice from Doug McIlroy"에 있다. ↩

2 맥일로이가 정식화한 유닉스 철학의 표준 인용문은 M. D. McIlroy, E. N. Pinson, B. A. Tague, "UNIX Time-Sharing System: Foreword", *The Bell System Technical Journal* 57(6), 1978, pp. 1899–1904에 있다. ↩

3 IBM 7090(1959)과 7094(1962)는 벨 연구소가 이 시기에 쓰던 트랜지스터 기반 대형 과학계산기다. 입력은 천공카드, 출력은 라인프린터였고 대화형 사용이 아니라 작업을 모아 한꺼번에 돌리는 배치 처리 방식이었다. ↩

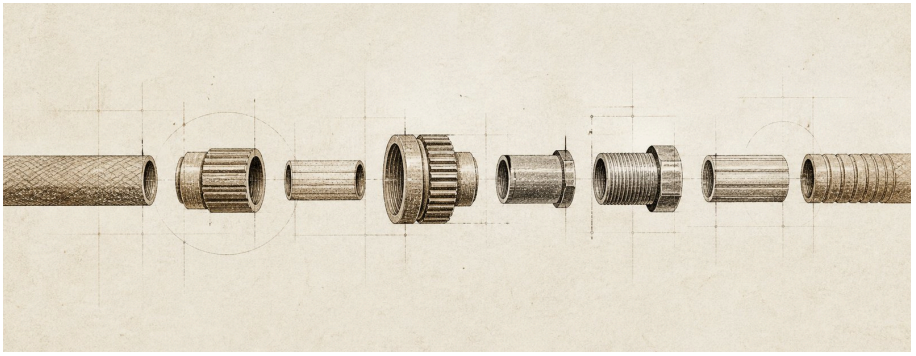
4 Xuefeng Li, Nir Ben-Israel, Yotam Raz, Belal Ahmed, Doron Serebro, Antoine Raux, "RepoMod-Bench: A Benchmark for Code Repository Modernization via Implementation-Agnostic Testing", arXiv:2602.22518 [cs.SE], 2026-02-26. 〈https://arxiv.org/abs/2602.22518〉 — 21개 저장소·8개 언어·160만 줄·11,616개 테스트. 규모별 평균 통과율은 91.3%(1만 줄 미만) → 66.9%(1만~5만) → 15.3%(5만 이상)로 무너진다. 2026년 8월 현재 arXiv 프리프린트이며 학회 게재 여부는 확인되지 않았다. ↩

5 같은 논문. 원문: "The challenge therefore lies not in fitting code into context, but in maintaining coherent architectural understanding across thousands of interdependent files — a qualitatively different problem from the context-window bottleneck of raw LLMs." ↩

1 부

경계를 그은 손들

경계를 누가 그었고, 무엇이 그것을 강제했는가.



1장

딱 한 장이 남았다

1964년 10월, 뉴저지

남은 것은 그 메모의 열 번째 페이지 한 장이다. 원래 몇 장이었는지는 기록이 없다.

맥일로이(Doug McIlroy)가 그 문서에서 정리한 것은 목록이었다. 무엇이 가장 중요한가. 네 가지가 적혔다. 프로그램을 정원 호스처럼 연결하는 것. 링크-로딩을 할 수 있는 로더. 라이브러리를 정리하는 일반적인 체계. 시스템 구성 요소를 개인적으로 확보할 수 있는 것.[1]

넷 중에 하나만 살아남았다. 정확히 말하면, 넷 중 하나가 나머지 셋을 압도했다.

그 항목이 말한 것은 단순했다. 프로그램들을 세그먼트처럼 취급하자. 데이터를 다른 방식으로 주물러야 할 일이 생기면, 새 세그먼트를 하나 더 돌려 끼우면 된다. 호스를 잇듯이.

이 비유가 왜 강한지는 비유가 **생략한 것**을 보면 알 수 있다. 정원 호스를 이을 때 우리는 앞 세그먼트 안에서 물이 어떤 경로로 흘렀는지 묻지 않는다. 나사산의 규격만 맞으면 된다. 안을 몰라도 꽂을 수 있다는 것 — 그것이 이 비유의 전부이고, 이 책이 60년 뒤에 다시 다루게 될 것도 정확히 그것이다.

> **파이프 (pipe)**
>
> 한 프로그램의 표준 출력(stdout)을 다른 프로그램의 표준 입력(stdin)에 직접 연결하는 유닉스의 장치. 셸에서 수직선 하나로 쓴다 — `ls | grep txt | wc -l`. 중간에 파일을 만들지 않고, 앞 프로그램이 내보내는 즉시 뒤 프로그램이 받는다. 이 책이 파이프를 첫 장에 놓는 이유는 기능 때문이 아니라 **계약의 모양** 때문이다: 양쪽 프로그램은 서로의 존재조차 모르고, 합의한 것은 "줄바꿈으로 나뉜 바이트가 흐른다"는 것 하나뿐이다.

8년

메모는 1964년에 쓰였다. 리치는 파이프가 1972년에 유닉스에 들어갔다고 적었다.

문서로 확인되는 범위는 그보다 넓다. 1972년 6월의 2판 매뉴얼에는 파이프가 없다. 1973년 1월 15일에 맥일로이가 돌린 강연 공지에는 있다. 회상은 1972년을 가리키고 문서는 그 두 시점 사이를 가리킨다 — 이 책은 둘을 다 적어 둔다.[2]

8년 동안 무슨 일이 있었는가에 대해, 리치(Dennis Ritchie)는 두 가지를 남겼다.

하나는 겸손이다. 그는 아이디어 자체가 새롭지 않았다고 적었다. 파이프라인은 코루틴(coroutine)의 한 형태이고, 다트머스 시분할 시스템

(Dartmouth Time-Sharing System)의 "커뮤니케이션 파일(communication files)"이 거의 같은 일을 하고 있었다.[3]

> **코루틴 (coroutine)**
>
> 서로를 부르고 부름당하는 관계가 아니라, 실행권을 주고받는 관계에 있는 두 루틴. 일반적인 함수 호출은 부른 쪽이 끝날 때까지 기다리지만, 코루틴은 한쪽이 값을 하나 내놓고 멈추면 다른 쪽이 이어 돌고 다시 돌려준다. 파이프로 이어진 두 프로그램이 정확히 이 관계다 — 앞쪽이 한 줄을 내보내면 뒤쪽이 그것을 받아 처리하고, 앞쪽이 다시 돈다. 멜빈 콘웨이(Melvin Conway)가 1958년에 만든 말이고, 활자로 처음 나온 것은 1963년 논문이다. 3장에서 다시 만나게 될 그 콘웨이다.[4]

다른 하나는 인정이다. 파이프가 운영체제에 들어간 것은 맥일로이의 제안 때문만이 아니라 그의 **집요함** 때문이었다.

역사가 마이클 마호니(Michael S. Mahoney)가 기록한 이 두 번째 문장은, 아이디어와 채택 사이에 무엇이 필요한지에 대한 이야기다.[5] 맥일로이는 그 8년 동안 같은 말을 했다.

무엇이 8년을 잡아먹었는가

파이프가 들어가고 난 뒤 리치가 남긴 평가는 짧다. 구현은 **비교적 간단한 일**이었다는 것.[6]

그러면 8년은 어디에 쓰였는가.

리치는 그 답도 적어 두었다. 초기에 맥일로이의 제안을 받았을 때 자기들이 무엇을 생각했는지에 대한 회고다. 표기가 너무 급진적으로 보였고, 명령의

매개변수와 입출력 파일을 어떻게 구분할지 알 수 없었고, 명령 하나에 입력 하나 출력 하나라는 모델이 너무 갑갑해 보였다. 그리고 그 문단은 이렇게 끝난다.

상상력의 실패였다.[7]

8년을 잡아먹은 것은 기술이 아니었다. 그것이 할 만한 일이라는 확신이었다.

나사산이 실제로 맞춰진 방식

파이프가 들어가면서 표기법도 하나 붙었다. 입출력 리다이렉션에 쓰던 문자를 재활용한 것이었고, 지금 우리가 쓰는 수직선이 아니라 이런 모양이었다.

```
sort input >pr>opr>
```

여기서 널리 퍼진 이야기 하나를 정정해야 한다. 수직선 기호가 붙어서 파이프가 퍼졌다는 서술이 흔한데, 리치의 회고는 그 반대 순서를 적는다.

새 기능은 열광적으로 받아들여졌고, filter라는 말이 곧 만들어졌다. 그리고 이 부분이 이 책에 중요하다.

많은 명령어가 파이프라인에서 쓸 수 있도록 고쳐졌다. 리치가 든 예가 구체적이다. `sort`나 `pr` 같은 유틸리티가 인자 없이 불렸을 때 표준 입력을 정렬하거나 출력하기를 누군가 원하리라고는 **아무도 상상하지 않았다**는 것이다.

이 모든 일이 `>pr>opr>` 표기 아래에서 일어났다.

수직선은 그 다음이다. 옛 표기에 문제가 드러났다. 공백으로 구분되는 탓에 인자를 주려면 따옴표를 씌워야 했고, `<`도 같은 뜻으로 받아들이는 바람에 같은 파이프라인을 여러 방식으로 쓸 수 있어 표기가 유일하지 않았다. 그래서 **두어 달 만에** 지금의 표기로 교체된다. 리치의 평가는 절제되어 있다. 옛 표기에도 나름의 매력과 내적 일관성이 있었지만 새 표기가 확실히 낫다는 것.[8]

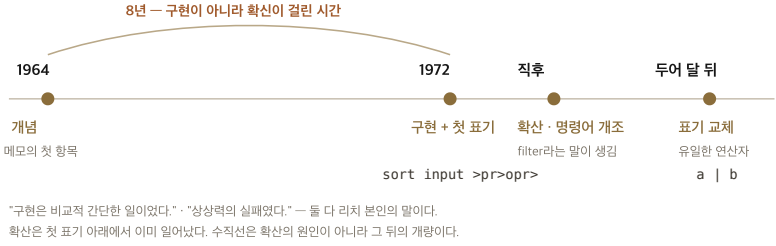


그림 1 — 8년은 구현이 아니라 확신에 쓰였고, 표기는 확산 뒤에 다듬어졌다.

그래서 무엇이 경계를 실재하게 했는가

순서를 다시 세우면 이렇게 된다.

개념은 1964년에 있었다. 구현은 간단했다. 8년을 잡아먹은 것은 그것이 할 만한 일이라는 확신이었다. 그리고 기능이 들어간 순간 퍼졌다.

그런데 퍼지는 데 필요했던 것이 하나 더 있었다. **양쪽 프로그램이 고쳐져야 했다.**

파이프는 두 프로그램 사이에 계약을 하나 세운다. 줄바꿈으로 나뉜 바이트가 표준 입력으로 들어오고 표준 출력으로 나간다는 것. 그런데 그 계약은 파이프가 생겼다고 존재하게 되는 것이 아니다. `sort`가 인자 없이 불렸을

때 표준 입력을 읽도록 고쳐져야, 그때 비로소 그 자리에 꽂을 수 있는 부품이 된다.

나사산은 발명된 것이 아니라 **양쪽이 그것에 맞춰 깎이면서** 실재하게 되었다.

이 순서는 이 책에서 계속 돌아온다. 경계를 긋는 장치가 생기는 것과, 그 경계 양쪽이 그 장치를 전제하고 다시 만들어지는 것은 다른 일이고, 후자가 훨씬 오래 걸린다.

대가

정원 호스 비유에는 지불해야 할 것이 있었다.

호스를 이으려면 모두가 같은 나사산을 써야 한다. 유닉스에서 그 나사산은 텍스트 스트림(text stream)이었다. 프로그램들 사이를 오가는 것은 구조체도 객체도 아니고, 줄바꿈으로 나뉜 바이트였다.

그 규격은 놀랄 만큼 오래 버텼고, 동시에 오래 불편했다. 구조가 있는 데이터를 텍스트로 납작하게 만들었다가 다시 파싱해서 복원하는 일이 수십 년간 반복됐다. 파싱은 자주 틀렸다. 공백이 하나 다르면 무너졌다.[9]

경계를 하나 그으면 그 경계를 넘는 비용이 생긴다. 그 비용을 감당할 수 있을 만큼 규격이 단순해야 하고, 단순한 규격은 표현할 수 있는 것이 적다.

이 교환은 이 책 전체에서 계속 돌아온다. 나사산을 단순하게 하면 조립이 쉬워지고 표현이 가난해진다. 나사산을 풍부하게 하면 표현이 늘고 아무도 그것을 지키지 않는다.

남은 질문

리치의 벽에 붙어 있던 한 장은, 나머지를 잃고 살아남은 페이지였다.

무엇이 그 한 장을 남겼는가. 나머지 세 항목 — 로더, 라이브러리 체계, 구성 요소 확보 — 도 그 시대에는 절실한 문제였다. 그것들은 해결되었고, 해결된 뒤에 잊혔다.

정원 호스는 해결되지 않았다. 매 세대마다 다시 물어졌다. 프로시저 호출로, 원격 객체로, 메시지 큐로, HTTP 요청으로, 그리고 지금은 에이전트 사이의 핸드오프 문서로.

같은 나사산 문제가 60년째 돌아오고 있다면, 그것은 아직 답이 없다는 뜻인가, 아니면 답이 매번 달라야 한다는 뜻인가.

주

1 맥일로이의 1964-10-11 벨 연구소 내부 메모. 리치 소장본 이미지: 〈https://www.nokia.com/bell-labs/about/dennis-m-ritchie/mdmpipe.html〉 ↩

2 리치 본인의 문장은 "Pipes appeared in Unix in 1972"다(주 3). 문서로는 유닉스 2판(1972-06) 매뉴얼에 파이프 시스템 콜이 없고 1973-01-15자 맥일로이의 강연 공지에는 있다. 회상과 문서가 완전히 겹치지 않으므로 이 책은 어느 한쪽으로 확정하지 않는다. ↩

3 Dennis M. Ritchie, "The Evolution of the Unix Time-sharing System", *AT&T Bell Laboratories Technical Journal* 63(6 Part 2), 1984, pp. 1577–1593. 원문은 벨연구소 사이트에서 내려갔다. 아카이브: 〈https://web.archive.org/web/2019/https://www.bell-labs.com/usr/dmr/www/hist.html〉 ↩

4 Melvin E. Conway, "Design of a Separable Transition-Diagram Compiler", *Communications of the ACM* 6(7), 1963, pp. 396–408. DOI 10.1145/366663.366704 — "coroutine"이라는 용어가 여기서 처음 쓰였다. ↩

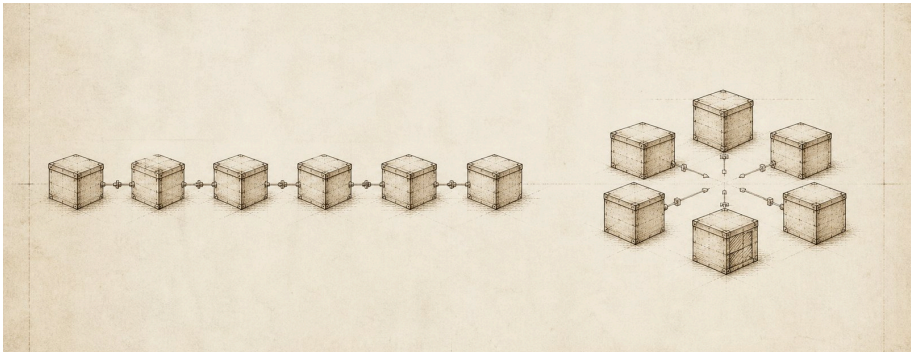
5 마이클 마호니(1939–2008)는 프린스턴의 과학사가로, 1989년에 벨 연구소의 유닉스 개발자들을 광범위하게 구술 인터뷰했다. 전사본: 〈https://www.princeton.edu/~hos/mike/transcripts/〉 ↩

6 주 3의 같은 글. 원문: "thanks to McIlroy's persistence, pipes were finally installed in the operating system (a relatively simple job), and a new notation was introduced." ↩

7 같은 글. 초기에 제안을 받아들이지 않은 이유들을 열거한 뒤 이어지는 문장이 "What a failure of imagination!"이다. ↩

8 주 3의 같은 글. 원문: "The new facility was enthusiastically received, and the term `filter' was soon coined. Many commands were changed to make them usable in pipelines." 표기 교체에 대해서는 "The pipe notation using `<' and `>' survived only a couple of months; it was replaced by the present one that uses a unique operator … the new one is certainly superior." 리치는 새 표기가 낫다고만 적었을 뿐 그것이 확산의 원인이었다고 말하지 않는다. "표기법이 붙고 나서야 퍼졌다"는 통설은 이 문단의 순서를 뒤집은 것이다. ↩

9 이 교환에 반대 방향으로 답한 사례가 2006년의 파워셸(PowerShell)이다. 파이프로 흐르는 것을 텍스트가 아니라 구조화된 .NET 객체로 바꿨다 — 파싱은 사라졌지만 나사산 규격이 특정 런타임에 묶였다. 〈https://learn.microsoft.com/powershell/scripting/overview〉 ↩



2 장

아무도 기준을 묻지 않았다

1972년 12월, CACM

파나스(David Lorge Parnas)는 논문을 인용으로 연다.[1]

데이비드 파나스 (David Lorge Parnas, 1941-)

소프트웨어 공학의 창시자 세대에 속하는 캐나다·미국의 컴퓨터과학자. 정보 은닉(information hiding)을 정식화했고, 미 해군 A-7E 코세어 II 공격기의 소프트웨어 요구사항 문서를 설계했으며, 1985년에는 레이건 행정부의 전략방위구상(SDI, "스타워즈") 자문위원을 소프트웨어가 신뢰성 있게 만들어질 수 없다는 이유로 공개 사임해 논쟁을 일으켰다. 이 장의 논문은 컴퓨터과학에서 가장 많이 인용된 논문 중 하나다.

CACM (Communications of the ACM)

미국 계산기학회(Association for Computing Machinery)의 기관지. 1958년 창간. 컴퓨터과학 분야에서 가장 오래되고 널리 읽히는 정기간행물이며, 이 책에 나오는 여러 논문의 게재지다.

그가 인용한 것은 1970년에 나온 시스템 프로그램 설계 교과서였다. 그 책은 시스템을 모듈로 나누라고 말한다. 좋은 조언이다. 그리고 파나스가 지적한 것은, 그 책이 **나누는 기준에 대해서는 아무 말도 하지 않았다**는 것이다.

이것이 그가 여섯 쪽짜리 논문에서 한 일이다. 답을 준 게 아니라, 아무도 묻지 않던 질문이 거기 비어 있다는 것을 보여줬다.

논문 제목은 「시스템을 모듈로 분해할 때 사용해야 할 기준에 관하여」(On the Criteria To Be Used in Decomposing Systems into Modules)였다. 제목이 곧 지적이다. 기준(criteria). 사람들이 모듈을 만들고는 있었지만, 왜 그 선에서 잘랐는지는 설명하지 못했다.

두 개의 분해

파나스는 같은 문제를 두 번 푼다.

문제 자체는 사소하다. 텍스트를 읽어서 어떤 처리를 하고 결과를 출력한다.[2] 첫 번째 분해는 자연스러운 것이다. 처리 단계를 따라 나눈다. 입력을 읽는 모듈, 순환 이동을 만드는 모듈, 알파벳순으로 정렬하는 모듈, 출력하는 모듈. 순서도를 그대로 상자로 바꾼 모양이다.

두 번째 분해는 이상하게 생겼다. 단계를 따르지 않는다. 대신 **바뀔 것 같은 결정**을 하나씩 골라서 각각을 모듈 안에 감춘다. 문자를 어떻게 저장할

것인가. 줄을 어떻게 표현할 것인가. 정렬을 언제 할 것인가.

첫 번째 분해에서는 문자 저장 방식을 바꾸면 네 모듈이 전부 바뀐다. 두 번째 분해에서는 한 모듈만 바뀐다.

그림 2 — 같은 문제의 두 분해. 색이 칠해진 상자가 "문자 저장 방식이 바뀌었을 때 손대야 하는 모듈"이다.

파나스는 이 결론을 선언하지 않는다. 두 설계를 나란히 놓고, 변경이 들어왔을 때 각각이 어떻게 되는지 보여준다. 독자가 스스로 계산하게 한다.

정보 은닉

이 논문에서 나온 말이 정보 은닉(information hiding)이다.

흔히 "변경 가능성을 기준으로 나눠라"라고 요약되지만, 원문의 동사는 감추는 것이다. 바뀔 것 같은 설계 결정을 모듈 안에 넣고 바깥에서 보이지 않게 한다. 바깥은 그 결정이 존재한다는 것조차 몰라야 한다.[3]

정보 은닉 (information hiding)

"바뀔 가능성이 있는 설계 결정을 모듈 안에 넣고, 그 모듈 바깥에서는 그 결정의 존재조차 알 수 없게 한다"는 원칙. 흔히 캡슐화(encapsulation)와 혼용되지만 층위가 다르다 — 캡슐화는 데이터와 그것을 다루는 코드를 한 단위로 묶는 **기법**이고, 정보 은닉은 무엇을 묶을지 고르는 **기준**이다. `private` 키워드를 쓰는 것은 캡슐화이고, 무엇을 `private`으로 할지 고르는 판단이 정보 은닉이다.

맥일로이의 나사산과 같은 방향이다. 안을 모르는 채로 쓸 수 있어야 한다. 다만 맥일로이는 **연결**을 말했고 파나스는 **분해**를 말했다. 어디서 자를 것인가와, 자른 것을 어떻게 이을 것인가. 두 사람이 각각 절반씩 말했다.

파나스가 지적한 효율 문제도 남았다. 그의 두 번째 분해는 당시의 통념 — 모듈은 서브루틴 하나 이상으로 구성된다는 가정 — 아래에서 구현하면 느려진다. 그는 그 대가를 숨기지 않았고, 그 페널티를 피할 다른 구현 방식을 스케치했다.

전제

이 기준에는 조용한 전제가 하나 들어 있다.

무엇이 바뀔지 안다는 것.

파나스의 분해가 좋은 분해가 되려면, 설계자가 미래의 변경을 어느 정도 맞혀야 한다. 문자 저장 방식이 바뀔 것 같다고 판단했기 때문에 그것을 감춘 것이다. 만약 실제로 바뀌는 것이 정렬 알고리즘이 아니라 입력 형식이었다면, 그 분해는 잘못된 곳을 감싼 것이 된다.

그래서 이 기준은 정확히 **사람이 도메인을 아는 만큼** 정확하다. 이 시스템이 무엇을 위한 것인지, 누가 쓰는지, 어느 쪽에서 요구가 들어올지를 아는 사람만이 어디를 감출지 고를 수 있다.

대가

정보 은닉에는 청구서가 있고, 그 청구서는 시간이 지나야 도착한다.

감춘다는 것은 바깥에서 안을 볼 수 없다는 뜻이다. 성능 문제가 생겼을 때, 잘못된 데이터가 흘렀을 때, 원인이 감춰진 쪽에 있으면 찾기가 어려워진다. 잘 감춘 모듈일수록 잘 디버깅되지 않는다.

그리고 감춤이 잘못된 곳에 그어졌을 때, 그것을 다시 여는 비용은 처음 그을 때보다 훨씬 크다. 경계는 굳는다.

54년 뒤

파나스의 질문은 지금도 유효하다. 무엇이 함께 변하는가.

다만 이 책이 다루려는 것은 그 질문의 답이 아니라, **질문의 주어**다. 파나스가 상정한 변경의 주체는 사람이었다. 요구사항을 받고, 코드를 읽고, 어디를 고칠지 판단하고, 고치는 사람.

그 자리에 다른 것이 들어오면 무엇이 달라지는가. 무엇이 바뀔지 예측해서 감추는 대신, **한 번에 볼 수 있는 양**을 기준으로 감춰야 한다면.

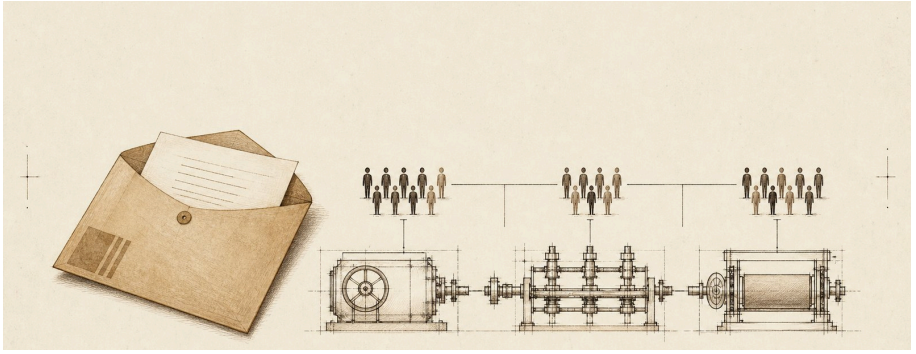
1972년의 논문은 그 질문에 답하지 않는다. 다만 그 질문을 물을 수 있는 어휘를 남겼다.

주

1 David L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM* 15(12), 1972-12, pp. 1053–1058. DOI 10.1145/361598.361623 〈https://dl.acm.org/doi/10.1145/361598.361623〉 — 1971년 카네기멜런 대학 테크리포트가 선행판이다. 전문 PDF: 〈https://cse.msu.edu/~cse870/Public/Lectures/SS2007/ParnasPapers/decomposition-macklem.pdf〉 ↩

2 파나스가 쓴 예제는 KWIC(Key Word in Context) 색인 시스템이다. 문장들을 입력받아 각 문장의 단어를 앞뒤로 순환 이동시킨 뒤 알파벳순으로 정렬해 출력한다. 1950년대 도서관 색인 기법에서 온 문제로, 당시 독자에게는 익숙한 사례였다. ↩

3 원문의 표현은 "its interface or definition was chosen to reveal as little as possible about its inner workings"이며, 각 모듈이 감추는 것은 "a design decision which it hides from all others"다. 한국어 요약에서 흔히 쓰이는 "변경 가능성 기준"은 이 문장의 파생이지 원문 표현이 아니다. ↩



3 장

반려당한 법칙

1967년, 거절

콘웨이(Melvin E. Conway)는 논문을 하버드 비즈니스 리뷰(Harvard Business Review, HBR)에 보냈다.

거절 사유가 남아 있다. 테제를 증명하지 못했다는 것이었다.[1]

멜빈 콘웨이 (Melvin Edward Conway, 1931-)

미국의 컴퓨터과학자. 1963년 논문에서 코루틴(coroutine)이라는 말을 처음 썼고(1장 참조), 1958년에 버로스 220용 어셈블러 SAVE를 만들었으며 — 코루틴을 실제로 적용한 최초의 어셈블러로 꼽힌다 —, 1980년대에는 개인용 컴퓨터를 위한 파스칼 컴파일러를 설계했다. 그러나 그의 이름이 남은 것은 이 장의 네 쪽짜리 잡지 기고 때문이고, 그 이름조차 그가 붙인 것이 아니다.

하버드 비즈니스 리뷰 (Harvard Business Review)

1922년 창간된 하버드 경영대학원의 경영 학술·실무 잡지. 학계와 실무계를 잇는 위치에 있어 경영 아이디어의 정전(canon) 역할을 해왔다. 이 장에서 중요한 것은 그 권위가 아니라, 그 권위가 요구한 것 — **데이터** — 이다.

논문이 주장한 것은 한 문장으로 요약된다. 시스템을 설계하는 조직은 그 조직의 커뮤니케이션 구조를 복제한 설계를 내놓게 되어 있다.[2] 넓은 의미의 시스템 어디에나 해당한다고 그는 적었다.

그리고 그 논문은 컴파일러 설계와 군용 무기 체계를 예로 들었다. 여덟 명을 다섯과 셋으로 나눠 각각 코볼과 알골 컴파일러를 맡겼더니 5패스짜리와 3패스짜리가 나왔다는 이야기다.

HBR이 요구한 것은 데이터였다. 예시가 아니라, 이 관계가 실제로 성립한다는 증거.

그가 가진 것은 관찰이었다.

1968년 4월, 게재

논문은 이듬해 데이터메이션(Datamation)에 실린다. 당시 IT 분야에서 가장 큰 잡지였다. 1968년 4월호. 제목은 「위원회는 어떻게 발명하는가」(How Do Committees Invent?).[3]

데이터메이션 (Datamation)

1957년 창간된 미국의 데이터 처리 산업 잡지. 1960~70년대에는 이 분야에서 가장 발행부수가 큰 매체였다. 학술지가 아니라 업계지였다는 점이 이 장의 아이러니에 포함된다 — 학술적 증명을 요구받아 거절당한 논문이, 증명을 요구하지 않는 매체에 실려 반세기를 살아남았다.

네 쪽 남짓이다.

그 네 쪽 안에 또 하나의 문장이 있다. 설계 팀을 조직하는 행위 자체가 이미 일부 설계 결정을 내린 것이라는 문장. 사람을 배치하는 순간 시스템의 이음매가 정해진다는 뜻이다.

이름

콘웨이의 법칙(Conway's Law)이라는 이름은 콘웨이가 붙이지 않았다.

가장 널리 받아들여지는 설명은 프레드 브룩스(Frederick P. Brooks Jr.)다. 1975년의 『맨먼스 미신』(*The Mythical Man-Month*)이 그 논문을 인용하면서 그렇게 불렀고, 그 이름이 남았다.[4]

프레드 브룩스 (Frederick P. Brooks Jr., 1931-2022)

IBM System/360 운영체제 개발을 이끈 뒤, 그 실패에서 배운 것을 『맨먼스 미신』(1975)으로 썼다. "늦어진 소프트웨어 프로젝트에 인력을 더 투입하면 더 늦어진다"는 브룩스의 법칙이 여기서 나왔다. 1999년 튜링상. 이 책에서 그가 하는 역할은 저자가 아니라 **명명자**다.

다른 설명도 있다. 논문이 나오고 몇 달 뒤인 1968년 7월, 모듈러 프로그래밍 심포지엄에서 조지 밀리(George H. Mealy)가 콘웨이 논문의 요지를 다시 정리하면서 그것을 콘웨이의 법칙이라 부르자고 적었다는 것이다.[5]

두 설명 중 어느 쪽이 맞는지는 이 책에서 판정하지 않는다. 확실한 것은 저자 본인이 아니라는 것뿐이다.

44년

증명은 2012년에 도착한다.

맥코맥(Alan MacCormack)과 볼드윈(Carliss Y. Baldwin)과 러스낙(John Rusnak)이 「제품 아키텍처와 조직 아키텍처의 이중성」이라는 논문에서 미러링 가설(mirroring hypothesis)을 검증한다. 같은 종류의 소프트웨어를, 하나는 긴밀하게 모인 조직이 만들고 하나는 흩어진 조직이 만들었을 때, 코드의 결합도가 어떻게 다른지를 측정했다.[6]

> **미러링 가설 (mirroring hypothesis)**
>
> "조직의 구조와 그 조직이 만든 제품의 구조가 서로를 반영한다"는 명제를 검증 가능한 형태로 다듬은 것. 콘웨이의 관찰과 다른 점은 **측정 가능성**이다. 조직의 결합도(같은 회사인가·같은 지역인가·같은 팀인가)와 코드의 결합도(설계 구조 행렬로 계량한 파일 간 의존)를 각각 수치로 만들고, 둘의 상관을 통계로 다룬다. 콘웨이가 못 했던 것이 정확히 이 계량화다.

논문은 *Research Policy* 41권 8호에 실린다. 1309쪽에서 1324쪽.

콘웨이가 원고를 보낸 지 45년, 게재된 지 44년 뒤다.

그림 3 — 관찰에서 증명까지. 이 책의 3부가 하려는 유비는 이 그림의 오른쪽 끝에서 검증 기간 0으로 출발한다.

HBR은 틀리지 않았다

되짚어 보면 편집자는 정확했다. 1967년의 그 원고에는 증명이 없었다. 관찰과 예시와 설득력 있는 논증이 있었지만, 그것은 증명이 아니다.

편집자가 몰랐던 것은 그 증명에 44년이 걸린다는 사실이다. 소프트웨어 조직의 커뮤니케이션 구조를 측정 가능한 형태로 만들고, 코드의 결합도를 계량하고, 둘 사이의 상관을 통계적으로 다룰 수 있게 되기까지 필요한 도구들이 그때는 없었다.

증명 요구는 옳았다. 다만 그 요구를 그 시점에 충족할 방법이 없었다.

그 사이에 벌어진 일

44년 동안 그 법칙은 증명되지 않은 채로 쓰였다.

인용되었고, 조직 개편의 근거가 되었고, 역으로 이용되었다. 원하는 아키텍처를 먼저 정하고 그 모양대로 팀을 나누는 방식에 이름이 붙었다. 역콘웨이 기동(Inverse Conway Maneuver)이라는 말이 쓰이기 시작한 것은 2010년대 중반이다.[7]

역콘웨이 기동 (Inverse Conway Maneuver)

콘웨이의 법칙을 도구로 뒤집어 쓰는 것. 조직이 아키텍처를 결정한다면, **원하는 아키텍처를 먼저 정하고 조직을 그 모양으로 재편하면** 그 아키텍처가 나온다는 발상이다. 소트웍스(Thoughtworks)의 기술 레이더가 이 표현을 퍼뜨렸다. 9장에서 이 개념의 시간 순서에 문제가 있다는 점을 다시 다룬다.

즉 업계는 검증되지 않은 관찰 위에 조직 설계를 얹었다. 그것이 잘못이었다고 말하기는 어렵다. 그 관찰은 결국 맞는 것으로 밝혀졌다.

다만 그것이 맞다는 것을 알기 전에 사람들이 그 위에 올라섰다는 사실은 남는다.

대가

콘웨이의 법칙에는 결정론으로 읽힐 위험이 있다.

조직 구조가 시스템 구조를 정한다는 문장을 강하게 읽으면, 설계자에게는 할 일이 없어진다. 조직도만 보면 아키텍처를 예측할 수 있다는 이야기가 되고, 실제로 그렇게 인용되는 경우가 있다.

원문은 그것보다 조심스럽다. 콘웨이가 말한 것은 조직이 커뮤니케이션 구조의 복사본을 만들도록 **제약된다**(constrained)는 것이다. 제약과 결정은 다르다.

이 책이 서 있는 자리

이 책의 뒷부분에서 비슷한 유비를 하게 된다. 사람 조직이 시스템 구조를 정했다면, 에이전트 편성도 그럴 것인가.

그 유비를 하기 전에 이 장을 먼저 놓는 이유가 있다. 콘웨이의 관찰이 증명되기까지 44년이 걸렸다. 사람으로 이루어진 조직에서, 반세기 가까이.

기계로 이루어진 편성에 대해 같은 종류의 주장을 할 때, 우리가 가진 검증 기간은 얼마인가.

주

1 반려 경위는 콘웨이 본인의 사이트에 기록되어 있다. 〈https://www.melconway.com/Home/Conways_Law.html〉 ↩

2 콘웨이 본인 사이트의 재진술: "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." ↩

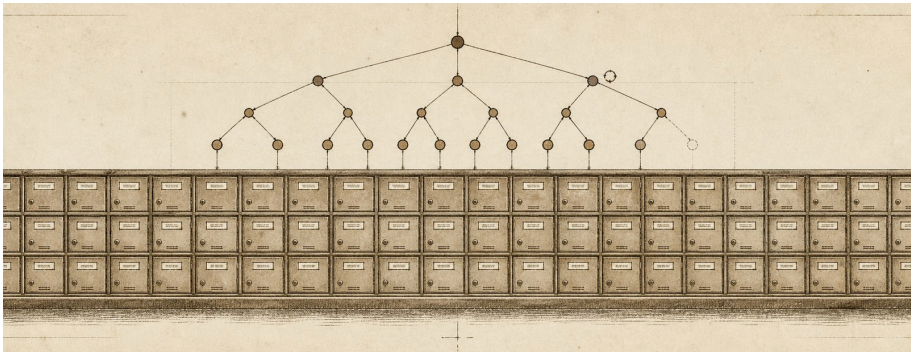
3 Melvin E. Conway, "How Do Committees Invent?", *Datamation*, 1968-04. 전문: 〈https://www.melconway.com/Home/Committees_Paper.html〉 — 흔히 붙는 서지 `14(5):28–31`은 2차 데이터베이스에만 있다. 콘웨이 본인 사이트의 재수록 저작권 표기는 매체명과 "April, 1968"까지만 적는다. 이 책은 권·호·쪽수를 확정 톤으로 쓰지 않는다. ↩

4 Frederick P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975. 콘웨이 법칙 언급은 "Why Did the Tower of Babel Fail?" 장 근처에 있다. 흔히 도는 "p.111"은 1995년 기념판 쪽수이고 1975년 초판과는 맞지 않으므로 이 책은 쪽수를 적지 않는다. ↩

5 출처로 지목되는 것은 1968년 7월 National Symposium on Modular Programming에서 조지 밀리가 발표한 "How to Design Modular (Software) Systems"다. 이 책은 그 자료를 1차 확인하지 못했다. 그리고 **콘웨이 본인 사이트는 브룩스만 명명자로 적는다** — 밀리 설은 그 사이트에 없다. 두 설 중 어느 쪽도 이 책은 확정하지 않는다. ↩

6 Alan MacCormack, Carliss Baldwin, John Rusnak, "Exploring the duality between product and organizational architectures: A test of the 'mirroring' hypothesis", *Research Policy* 41(8), 2012-10, pp. 1309–1324. DOI 10.1016/j.respol.2012.04.011 ↩

7 소트웍스 기술 레이더의 항목 설명: 〈https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver〉 ↩



4 장

메일함 하나와, 죽어도 되는 프로세스

1973년, 스탠퍼드

휴잇(Carl Hewitt)과 비숍(Peter Bishop)과 스타이거(Richard Steiger)가 IJCAI에 낸 논문의 제목에는 인공지능이 들어간다.[1]

칼 휴잇 (Carl Hewitt, 1944-2022)

MIT 인공지능 연구소의 컴퓨터과학자. 액터 모델(Actor Model)의 창안자. 플래너(Planner)라는 초기 인공지능 프로그래밍 언어를 만들었고, 그 경험이 액터 모델로 이어졌다. 그의 모델은 주류가 되지 못했지만, 지금 동시성을 다루는 거의 모든 언어와 프레임워크가 그 개념을 부분적으로 차용하고 있다.

IJCAI (International Joint Conference on Artificial Intelligence)

1969년부터 열린 인공지능 분야의 최상위 국제 학회. 1973년 스탠퍼드에서 열린 제3회 대회에서 액터 논문이 발표되었다.

그 시절의 인공지능은 지금과 다른 것을 가리켰다. 생성 모델이 아니라 계산 시스템 일반에 가까웠다. 논문은 컴퓨터 아키텍처와 프로그래밍 언어 설계와 인공지능 접근법을 아우르는 연구 비전을 펼치는 포지션 페이퍼(position paper)였다. 리스프(Lisp)와 시뮬라(Simula)와 스몰토크(Smalltalk)의 영향이 들어가 있었다.[2]

거기서 제안된 것이 액터(actor)다.

규칙 세 개

액터의 정의는 짧다.

모든 것이 액터다. 액터는 사적인 상태와 메일함(mailbox)을 가진 가벼운 프로세스다. 메시지를 **한 번에 하나씩** 순차적으로 처리한다. 상태는 절대 공유되지 않고, 경계를 넘어가는 것은 메시지뿐이다.

> **액터 모델 (Actor Model)**
>
> 동시성 계산의 수학적 모델. 액터는 메시지를 받으면 세 가지만 할 수 있다 — ①다른 액터에게 메시지를 보낸다 ②새 액터를 만든다 ③다음 메시지를 받을 때의 자기 행동을 바꾼다. 공유 메모리도 없고 락(lock)도 없다. 구울 아그하(Gul Agha)가 1985년 박사논문에서 이를 형식적으로 정식화했고, 이듬해 MIT 출판부에서 단행본으로 나왔다.[3]

이 세 줄에서 따라 나오는 결과가 하나 있다. 액터 하나 안에서는 데이터 경합(data race)이 원천적으로 불가능하다. 상태가 사적이고 한 번에 한 메시지만 처리하니까, 두 개의 흐름이 같은 값을 동시에 건드릴 방법이 없다.

동시성 문제를 푸는 방식이 아니라, 동시성 문제가 생길 수 없는 모양을 만드는 방식이다.

이 책이 쓰는 어휘로 옮기면 액터는 **닫힌 계**의 가장 순수한 형태다. 안에서 판단이 완결된다. 바깥을 알 필요가 없다. 알 방법도 없다.

1973년이었다.

스웨덴의 교환기

이론이 교환기 위에서 돌게 된 것은 다른 이야기다.

얼랭(Erlang)과 OTP는 통신 시스템을 위해 설계되었다. 수백만 개의 동시 연결을 처리하는 교환기가 목표였다.

> **얼랭과 OTP (Erlang / OTP)**
>
> 스웨덴의 통신장비 회사 에릭슨(Ericsson)에서 조 암스트롱(Joe Armstrong) 등이 만든 프로그래밍 언어와 그 표준 라이브러리(Open Telecom Platform). 전화 교환기의 요구 — 수백만 동시 연결, 무중단 운영, 부분 장애 감내 — 를 언어 수준에서 풀었다. 1998년 오픈소스로 공개됐다. 왓츠앱(WhatsApp)이 가장 유명한 사례인데, 흔히 도는 "서버 수십 대"는 부정확하다 — 유명한 숫자는 **엔지니어 50명 안팎**과 **서버 한 대당 200만 연결**이고, 서버 자체는 수백 대 규모였다.[4]

거기서 붙은 것이 위치 투명성(location transparency)이다. 액터를 가리키는 참조는 그 액터가 같은 프로세스 안에 있든, 같은 기계에 있든, 네트워크 건너에 있든 똑같이 동작한다. 얼랭의 프로세스 식별자(PID)는 로컬 프로세스를 가리킬 때와 다른 노드의 프로세스를 가리킬 때 생김새가 같다.

여기서 잠깐 멈출 필요가 있다. 다음 장에 나올 CORBA도 같은 것을 약속했다. 네트워크 너머의 객체를 로컬 객체처럼 부르게 해주겠다는 약속이었고, 그것은 실패로 기록된다.

차이는 방향에 있다. CORBA는 원격 호출을 로컬 호출처럼 **보이게** 만들었다. 얼랭은 로컬 호출도 원격 호출처럼 **다루게** 만들었다. 전자는 실패 가능성을 숨겼고, 후자는 모든 것에 실패 가능성을 부여했다.

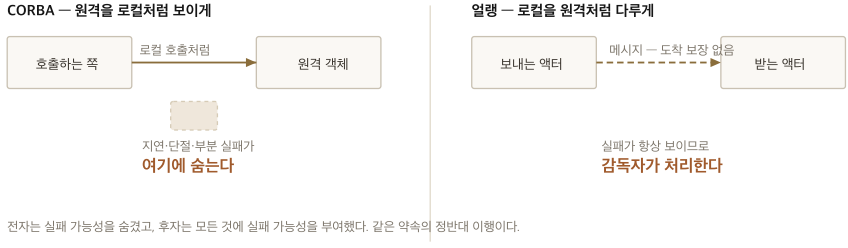


그림 4 — 같은 "위치 투명성", 반대 방향. 한쪽은 실패를 숨겼고 다른 쪽은 모든 호출에 실패를 부여했다.

죽게 둔다

얼랭에서 가장 자주 인용되는 문장은 오류 처리에 대한 것이다.

let it crash. 죽게 둬라.

구조는 이렇다. 오류가 날 만한 일은 말단 액터에 밀어 넣는다. 그 액터는 예상치 못한 실패가 오면 처리하려 들지 않고 그냥 죽는다. 감독하는 부모가 그것을 보고 정해진 지침 중 하나를 실행한다. 다시 시작하거나, 재개하거나, 멈춘다. 부모가 감당할 수 없는 종류의 문제면 위로 올린다.

액터들이 자식을 만들면서 계층이 생기고, 하나의 루트에서 내려오는 나무 모양이 된다. 감독 트리(supervision tree)다.[5]

> **감독 트리 (supervision tree) · let it crash**
>
> 얼랭/OTP의 장애 처리 구조. 작업하는 워커(worker) 프로세스 위에 감독자(supervisor) 프로세스를 두고, 감독자는 자식이 죽으면 미리 선언한 전략대로 반응한다 — `one_for_one`(죽은 것만 재시작), `one_for_all`(형제 전부 재시작), `rest_for_one`(그 뒤 순서의 형제들 재시작). 감독자 자신도 상위 감독자의 자식이라 트리가 된다. **핵심은 복구가 아니라 격리다** — 실패가 어디까지 번지는지를 트리의 모양으로 미리 정해 둔 것이다.

철학은 이렇게 정리된다. **모르는 오류를 처리하려 들지 않는다.** 대신 그것이 전체를 무너뜨리지 않게 한다.

이 문장이 왜 반직관적인지는 반대편을 생각하면 보인다. 대부분의 언어에서 좋은 코드란 오류를 잡아서 처리하는 코드다. 예외를 삼키지 말고, 로그를 남기고, 복구를 시도하는 것. 얼랭은 그 반대를 말한다. 무엇을 해야 할지 모를 때 뭔가를 하려는 시도가 상태를 더 망친다.

강제하는 것

이 장이 이 책에 있는 이유는 격리 자체가 아니라, **격리를 무엇이 강제했는가**다.

파이프에서 경계를 강제한 것은 관례였다. 프로그램이 표준 입출력을 쓰기로 약속했다. 어기려면 어길 수 있었다. 파나스의 정보 은닉도 마찬가지다. 모듈 안의 결정을 감춘다는 것은 규율이었고, 언어가 강제하는 경우는 드물었다.

얼랭에서는 런타임이 강제했다. 프로세스끼리 메모리를 공유할 수 없는 것은 예의가 아니라 물리였다. 우회할 방법이 없다.[6]

경계를 지키는 힘이 사람의 규율에서 시스템의 구조로 옮겨간 첫 대규모 사례다. 이 이동은 이 책의 후반부에서 다시 나온다. 다만 그때는 런타임이 아니라 타입 시스템과 CI가 그 자리를 맡는다.

대가

모든 것을 메시지로 바꾸면 값을 치른다.

함수 하나를 부르면 될 일이 메시지를 만들고 보내고 받고 응답을 기다리는 절차가 된다. 코드가 길어지고, 흐름이 한 곳에서 읽히지 않는다. 무엇이 무엇을 호출했는지 따라가려면 여러 파일을 오가야 한다.

그리고 디버깅이 어려워진다. 스택 트레이스(stack trace) 하나로 사고 경위를 재구성할 수 없다. 액터 A가 보낸 메시지가 B에서 처리되다 죽었고 C가 그것을 재시작했다면, 그 이야기는 세 곳의 로그에 흩어져 있다.

격리는 사고를 가둔다. 동시에 사고의 **이야기**도 가둔다.

남은 것

액터 모델은 주류가 되지 못했다. 얼랭은 통신과 메시징 인프라라는 특정 자리에서 오래 살아남았고, 그 아이디어는 엘릭서(Elixir)와 아카(Akka) 같은 후손으로 이어졌다.[7]

주류가 되지 못한 이유는 여러 가지로 설명되지만, 이 책이 관심을 갖는 것은 다른 부분이다. 사람이 코드를 쓰는 동안에는 모든 것을 액터로 만드는

대가가 이득보다 컸다. 사람은 함수 호출을 읽을 수 있고, 스택 트레이스를 따라갈 수 있고, 전역 상태를 머릿속에 담을 수 있다. 어느 정도까지는.

그 어느 정도가 달라지면 계산이 달라지는가.

주

1 Carl Hewitt, Peter Bishop, Richard Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence", *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, Stanford, 1973, pp. 235–245. 〈https://dl.acm.org/doi/10.5555/1624775.1624804〉 ↩

2 시뮬라(Simula 67)는 객체와 클래스를 처음 도입한 언어이고, 스몰토크(Smalltalk)는 "모든 것이 객체이며 객체는 메시지로만 소통한다"는 원칙을 밀어붙인 언어다. 액터 모델은 이 계보 위에서 메시지 전달을 동시성의 유일한 수단으로 격상시켰다. ↩

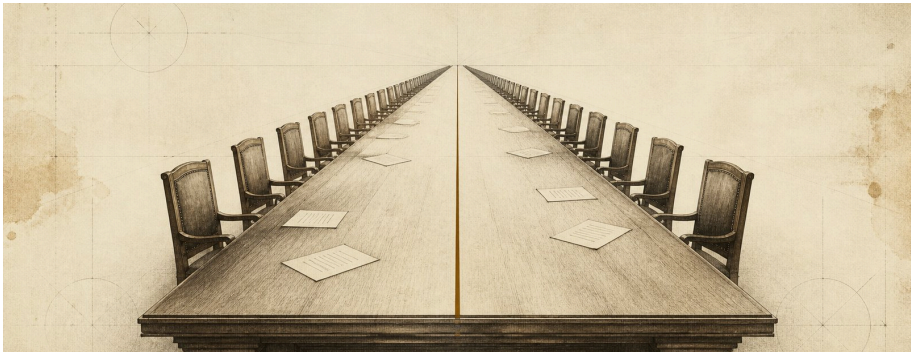
3 Gul Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986. 학위논문은 1985년 **미시간 대학**(지도교수 John H. Holland)이다 — 연구 자체는 MIT 휴잇 그룹에서 이루어졌으나 학위는 미시간에서 받았다. 이 둘을 흔히 혼동한다. ↩

4 얼랭의 설계 배경은 조 암스트롱의 박사논문에 정리되어 있다. Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, KTH, 2003. 〈https://erlang.org/download/armstrong_thesis_2003.pdf〉 — 개발 착수 시점을 1986년으로 적는 2차 자료가 흔하나, 이 책은 그 연도를 1차 확인하지 못했다. ↩

5 OTP 감독자 동작 명세: 〈https://www.erlang.org/doc/design_principles/sup_princ.html〉 ↩

6 얼랭 프로세스는 각자 독립된 힙(heap)을 갖고 가비지 컬렉션도 따로 돈다. 메시지는 값 복사로 전달된다(대용량 바이너리는 참조 카운팅되는 별도 영역을 쓰는 예외가 있다). 공유가 없다는 것이 관례가 아니라 런타임의 메모리 구조라는 뜻이다. ↩

7 엘릭서(Elixir)는 얼랭 가상머신(BEAM) 위에서 도는 언어로 2011년 조제 발림(José Valim)이 만들었다. 아카(Akka)는 JVM 위의 액터 프레임워크다. ↩



5 장

위원회가 그은 선

1989년, 열한 개 회사

객체 관리 그룹(Object Management Group, OMG)은 열한 개 회원사로 출범한다. HP·IBM·썬·애플·아메리칸 항공·데이터 제너럴·3Com·캐논·필립스·유니시스 같은 이름들이었다.[1]

> **OMG (Object Management Group)**
>
> 1989년에 설립된 비영리 기술 표준 컨소시엄. 특정 벤더에 묶이지 않는 상호운용 표준을 회원사 합의로 만드는 것이 목적이었다. CORBA 외에도 UML(통합 모델링 언어), BPMN(비즈니스 프로세스 모델 표기법) 같은 표준이 여기서 나왔다. 이 장에서 중요한 것은 이 조직이 만든 산출물이 아니라 **산출물을 만드는 방식** — 회원사 투표 — 이다.

헌장에 적힌 목표는 넓게 쓰였다. 널리 이용 가능한 인터페이스 명세에 기반해서, 객체지향 애플리케이션을 위한 공통 아키텍처 프레임워크를

제공한다는 것이었다. 서로 다른 벤더가 만든 객체 컴포넌트들이 네트워크와 운영체제를 넘어 상호운용되게 하는 것.

문제 설정 자체는 정확했다. 1989년의 기업 전산실에는 서로 말이 통하지 않는 시스템들이 쌓여 있었다.

1991년, 말이 통하지 않는다

CORBA 1.0은 1991년 10월에 나온다. 객체 모델과 인터페이스 정의 언어(IDL)가 들어갔고, 동적 요청 관리와 인터페이스 저장소를 위한 핵심 API가 있었다. 언어 매핑은 C 하나였다.[2]

> **CORBA (Common Object Request Broker Architecture)**
>
> 서로 다른 언어·운영체제·기계에 있는 객체들이 서로의 메서드를 부를 수 있게 하는 표준. 인터페이스 정의 언어(IDL)로 계약을 적으면 각 언어의 스텁(stub)과 스켈레톤(skeleton) 코드가 생성되고, 그 사이를 ORB가 중개한다. 1990년대 기업 미들웨어의 표준이 되려 했고, 실패했다. 이 책에서 CORBA는 **경계를 숨기려는 시도의 대표 사례**다.

> **ORB (Object Request Broker)**
>
> CORBA에서 원격 객체 호출을 중개하는 부품. 호출하는 쪽의 요청을 받아 대상 객체를 찾고, 인자를 직렬화해 보내고, 결과를 되돌려준다. 프로그래머 눈에는 그냥 지역 메서드 호출로 보인다 — 그것이 설계 의도였고, 문제의 시작이기도 했다.

그리고 빠진 것이 있었다.

1.x는 ORB끼리 통신할 표준 프로토콜을 정하지 않았다. 표준 프로토콜이 없다는 것은, 한 벤더의 ORB가 다른 벤더의 ORB와 말을 못 한다는 뜻이었다.

서로 다른 벤더의 컴포넌트가 상호운용되게 하겠다는 헌장을 가진 명세가, 자기 자신의 구현체들끼리 상호운용되지 않는 상태로 발행되었다.

그 구멍을 메운 것이 IIOP(Internet Inter-ORB Protocol)이고, 그것이 들어간 것은 1996년 8월의 2.0이다. 5년.[3]

절차

미치 헤닝(Michi Henning)은 1995년부터 2002년까지 CORBA를 다뤘다. OMG 아키텍처 보드의 일원이었고, ORB를 직접 구현했고, 컨설팅과 교육을 했고, C++ 프로그래밍 책을 썼다.

그가 쓴 회고문의 제목은 「CORBA의 흥망」(The Rise and Fall of CORBA)이다.[4]

그 글이 기술적 결함을 나열하기는 한다. API의 불필요한 복잡성과 비일관성. 버그가 많고 쓰기 어려운 C++ 매핑. 보안과 버전 관리 같은 핵심 기능의 부재. OMG는 보안과 방화벽 통과를 여러 번 명세하려 했지만 기술적 미비와 방화벽 벤더들의 무관심 때문에 포기했다. 버전 관리는 상속에 의한 버전 구분 말고는 아무 메커니즘이 없었다.

그런데 헤닝이 그 목록에 붙인 문장은, 이것들이 원인이 아니라 증상이라는 것이다.

세계 최대의 소프트웨어 컨소시엄이 어떻게 이런 결함을 만들어냈는가. 그가 지목한 답은 절차였다.

OMG는 합의로 기술을 발행한다. 회원사가 RFP(제안 요청서) 발행을 투표하고, 회원사들이 초안 명세를 제출하고, 회원사들이 무엇을 채택할지 투표한다. 이론상 민주적이다.

실제로는 두 가지가 겹쳤다. 참가에 자격 요건이 없어서, 투표 대상 기술을 거의 이해하지 못한 참가자가 많았다. 그리고 RFP가 검증되지 않은 기술을 요구하는 일이 잦았다. 아직 아무도 만들어보지 않은 것을 명세부터 만들었다.

그 결과가 헤닝이 적은 문장으로 요약된다. 기능들을 합치다 보니 명세가 누군가 한 번쯤 생각해낸 모든 기능이 들어간 부엌 싱크대(kitchen sink)가 되었다.

그림 5 — 경계를 그은 손이 "합의 절차"일 때 생기는 것. 각 회원사가 자기 조각을 밀어 넣고, 결과물에는 주인이 없다.

반박

이 회고에는 반박이 붙어 있다.

TAO ORB로 알려진 더글러스 슈미트(Douglas C. Schmidt)가 응답을 게시했다. 일부 기술적 한계 지적은 맞지만, 다른 것들은 맥락에서 떼어냈거나 오래된 명세에 근거한 것이며, CORBA는 1990년대 전성기보다 오히려 더 많은 산업 영역에서 쓰이고 있다는 것이었다.[5]

이 책은 어느 쪽이 옳은지 판정하지 않는다. 다만 한 가지는 양쪽 다 부정하지 않는다. CORBA는 사람들이 기대한 자리에 있지 못했다.

숨기려던 것

CORBA의 야심을 한 문장으로 줄이면, 네트워크를 안 보이게 하겠다는 것이었다.

원격 객체의 메서드를 로컬 객체의 메서드처럼 부를 수 있다면, 프로그래머는 분산을 신경 쓰지 않아도 된다. 코드는 깨끗해지고, 나중에 배치를 바꿔도 코드는 그대로다.

그런데 네트워크는 로컬 호출과 다르다. 지연이 있고, 끊기고, 순서가 바뀌고, 부분적으로 실패한다. 로컬 호출처럼 보이게 만들면 이 차이들이 사라지는 게 아니라 **보이지 않는 곳으로 밀려난다**.

이 교훈은 이후 여러 번 다시 발견된다. 경계는 숨겨서 없앨 수 있는 것이 아니다. 숨기면 그것을 넘는 비용이 예측 불가능한 순간에 청구된다. 다음 장의 여덟 줄짜리 목록이 정확히 이 교훈을 압축한 것이고, 그 목록이 나온 자리도 같은 시기의 분산 객체 시스템이었다.

위원회가 그은 선

이 장의 제목이 가리키는 것은 헤닝의 진단이다.

경계를 그은 것이 사람도 하드웨어도 조직도 아니라 **합의 절차**였을 때 무슨 일이 벌어지는가.

투표로 만들어진 경계에는 주인이 없다. 어느 회원사도 그 경계 전체를 책임지지 않는다. 각자가 자기 부분을 밀어 넣었고, 결과물은 아무도 원하지 않은 모양이 된다.

그리고 아무도 그것을 강제하지 않는다. 명세는 문서일 뿐이고, 문서는 스스로를 검사하지 않는다. 구현체들이 각자 다르게 해석해도 그 사실을 알려주는 것이 없다.

남은 질문

CORBA는 위원회가 만들었기 때문에 실패했는가, 아니면 그 시점에 아무도 풀 수 없는 문제를 명세부터 시작했기 때문인가.

이 질문이 지금 다시 물어질 수 있는 자리가 있다. 에이전트들이 서로 통신하는 방식에 대한 명세들이 지금 여러 곳에서 만들어지고 있다.[6] 그중 얼마가 구현보다 먼저 쓰이고 있는가.

주

1 OMG의 연혁과 명세 버전 이력: 〈https://www.omg.org/about/〉 · 〈https://www.omg.org/spec/CORBA/〉 ↩

2 CORBA 1.0(1991-10), 1.1(1992-02), 1.2(1993-12), 2.0(1996-08). 헤닝의 회고는 2.0을 1997년으로 적는데, 이는 명세 발행 시점과 정식 채택 시점의 차이로 보인다. 이 책은 OMG의 버전 이력 표기를 따른다. ↩

3 IIOP는 GIOP(General Inter-ORB Protocol)를 TCP/IP 위에 얹은 것이다. 2.0에서 처음 필수 사항이 되었고, 이후 자바 RMI-IIOP 등으로 이어졌다. ↩

4 Michi Henning, "The Rise and Fall of CORBA", *ACM Queue* 4(5), 2006-06. 〈https://queue.acm.org/detail.cfm?id=1142044〉 — *Communications of the ACM* 51(8), 2008에 재수록. ↩

5 반론은 더글러스 슈미트의 밴더빌트 대학 페이지에 게시되어 있다. 〈http://www.dre.vanderbilt.edu/~schmidt/corba-response.html〉 — 그가 TAO(The ACE ORB)라는 실시간 CORBA 구현의 주개발자인 것은 확인되지만, **그 페이지에는 서명이 없다.** 그래서 이 책은 "그가 게시했다"까지만 적고 집필자를 단정하지 않는다. ↩

6 2024년 이후 등장한 것들로 MCP(Model Context Protocol, Anthropic, 2024-11), A2A(Agent2Agent, Google, 2025-04) 등이 있다. 이 책은 이들의 성패를 예측하지 않는다 — 다만 CORBA가 남긴 질문(구현이 명세를 앞섰는가)을 적용해 볼 수는 있다. ↩

6 장

논문이 없는 목록

여덟 줄

분산 시스템을 가르치는 거의 모든 입문 과정에 이 목록이 나온다.

네트워크는 신뢰할 수 있다. 지연은 0이다. 대역폭은 무한하다. 네트워크는 안전하다. 토폴로지는 바뀌지 않는다. 관리자는 한 명이다. 전송 비용은 0이다. 네트워크는 균질하다.[1]

분산 컴퓨팅의 오류 (Fallacies of Distributed Computing)

분산 시스템을 처음 만드는 사람이 반드시 하게 되는 여덟 가지 잘못된 가정의 목록. 각 항목은 **참인 명제가 아니라 사람들이 무심코 참이라 가정하는 명제**다. 여덟 개 전부 거짓이고, 그것을 참이라 가정하고 세운 시스템은 나중에 반드시 기워야 한다는 것이 요점이다.

여덟 개 전부 틀렸다는 것이 요점이다. 분산 애플리케이션을 처음 만드는 사람이 반드시 하게 되는 잘못된 가정들이고, 그 위에 세운 시스템은 필요 이상으로 복잡해져서 나중에 기워야 한다.

이 목록은 인용된다. 강의 슬라이드에 들어가고, 설계 문서의 근거가 되고, 코드 리뷰에서 무기가 된다.

그리고 이 목록에는 논문이 없다.

화이트보드

가장 널리 받아들여지는 경위는 이렇다.

1990년대 초, 썬 마이크로시스템즈(Sun Microsystems)에서 빌 조이(Bill Joy)와 라이언(Lyon)이라는 성을 가진 동료가 처음 네 개를 정리했다. 제임스 고슬링(James Gosling)이 그것을 네트워크 컴퓨팅의 오류라고 이름 붙였다. 1994년에 피터 도이치(L. Peter Deutsch)가 다섯 번째부터 일곱 번째를 추가해 일곱 개짜리 목록을 만들었다. 1997년경 고슬링이 여덟 번째를 붙였다. 네트워크는 균질하다는 항목이다.

썬 마이크로시스템즈와 세 사람

썬 마이크로시스템즈(1982–2010)는 워크스테이션과 서버를 만들던 회사로, 자바(Java)·NFS·ZFS·솔라리스가 여기서 나왔다. 2010년 오라클에 인수됐다. **빌 조이**는 공동창업자이자 BSD 유닉스와 vi 편집기의 저자. **제임스 고슬링**은 자바를 만든 사람. **피터 도이치**는 고스트스크립트(Ghostscript)의 저자이자 스몰토크 구현으로 유명한 컴파일러 연구자. 즉 이 목록은 **당대 최고 실력자들의 입에서 나왔고, 그래서 더더욱 아무도 출처를 묻지 않았다.**

어느 시점에도 논문은 없었다.

한 설명에 따르면 도이치는 분산 객체 시스템의 문제를 논의하던 중에 화이트보드에 일곱 개를 적었다. 그것이 시작이었다. 정식으로 출판된 적이 없이 구전으로 퍼졌고, 처음 제대로 문서화된 것은 2006년에 아르논 로템갈오즈(Arnon Rotem-Gal-Oz)가 쓴 해설이다.[2]

화이트보드에서 문서까지 12년이다.

누가 썼는지도 다르다

출처가 없는 목록에는 다른 종류의 문제가 따라온다. 귀속이 흔들린다.

첫 네 개를 정리한 동료의 이름이 소스마다 데이브 라이언(Dave Lyon)이기도 하고 톰 라이언(Tom Lyon)이기도 하다. 한 소스는 빌 조이와 톰 라이언이 처음 네 개를 만들고 고슬링이 다섯에서 일곱을 보탰다고 적는다. 도이치와 고슬링의 역할이 뒤바뀐 판본이다. 여덟 번째가 고슬링의 것인지 도이치의 것인지도 확정되지 않은 채 두 이름이 같이 인용된다.[3]

한 해설자는 이 상황을 이렇게 요약했다. 오류들의 정확한 기원은 다소 혼란스럽다.

아이러니

이 목록이 가르치는 것은 하나로 요약된다.

네트워크 경계를 숨기지 마라. 명시하라.

지연이 있다는 것을 코드에 드러내라. 실패할 수 있다는 것을 타입에 넣어라. 대역폭에 한계가 있다는 것을 설계에 반영하라. 보이지 않게 만든 것은 사라지지 않고 나중에 청구된다.

그 규범 자체가 12년 동안 명시되지 않은 채 전해졌다.

출처가 어디인지, 누가 무엇을 보탰는지, 원문이 어떻게 쓰였는지가 명시되지 않은 채로. 사람에게서 사람에게로. 슬라이드에서 슬라이드로.

규범	1차 문서	강제 수단	실제로 지켜졌나
파이프 · 텍스트 스트림	있음	관례	대체로
정보 은닉 (파나스)	있음	규율	사람에 따라
분산 컴퓨팅의 오류	없음	구전 — 없음	아니오
베조스 API 칙령	없음	해고 (전언)	확인 불가
semver	있음	패키지 매니저	예
타입 검사기 (소르베)	있음	컴파일 실패	예

경계를 명시하라고 가르쳐온 분야가, 정작 자기 규범의 출처는 명시하지 않았다.

그림 6 — 이 책이 따라가는 다섯 규범의 "강제 수단". 구전 규범은 30년 동안 아무도 강제하지 않았고, 기계가 읽는 규칙만 실제로 강제된다.

왜 살아남았는가

출처가 없는데도 이 목록이 살아남은 이유를 생각해볼 만하다.

한 가지 설명은 그것이 짧다는 것이다. 여덟 줄이고, 각 줄이 한 문장이며, 전부 부정문이다. 외우기 쉽고 옮기기 쉽다.

다른 설명은 그것이 매번 다시 확인된다는 것이다. 분산 시스템을 만드는 사람은 조만간 이 여덟 개 중 하나에 걸린다. 걸린 다음에는 목록이 옳았다는 것을 안다. 출처가 필요 없다. 증거가 자기 시스템 안에서 나온다.

이 두 번째 설명이 CORBA와 대비된다. CORBA의 명세는 길고, 정식으로 출판되었고, 위원회의 승인을 거쳤다. 그리고 아무도 그것을 다 읽지 않았다.

문서화의 형식과 규범의 생존은 다른 축에 있다.

대가

출처 없이 살아남은 규범에는 값이 붙는다.

수정할 수 없다. 이 여덟 개 중 일부는 2026년의 데이터센터 안에서 다르게 읽힌다. 같은 랙 안의 두 서버 사이에서 지연은 0이 아니지만 0에 가깝고, 대역폭은 무한하지 않지만 대부분의 워크로드에게는 충분하다. 원본이 없는 목록은 개정판을 낼 방법이 없다.

그리고 맥락이 사라진다. 이 목록이 나온 자리는 1990년대의 분산 객체 시스템이었다. RPC와 CORBA와 자바 RMI가 원격 호출을 로컬 호출처럼 보이게 만들려던 시절이다. 그 맥락을 모른 채 목록만 받으면, 그것이 무엇에 대한 경고였는지가 흐려진다.

> **RPC와 자바 RMI**
>
> RPC(Remote Procedure Call, 원격 프로시저 호출)는 네트워크 너머의 함수를 지역 함수처럼 부르게 하는 방식의 총칭으로, 1980년대 초 제록스 파크(Xerox PARC)에서 정식화됐다. **자바 RMI**(Remote Method Invocation)는 그 자바판으로 1997년 JDK 1.1에 들어갔다. CORBA와 함께, 여덟 오류가 겨냥한 바로 그 "숨기려는 시도"의 계보다.

이 책이 이 장을 놓는 이유

이 책은 소프트웨어 아키텍처의 규범들이 어디서 왔는지를 따라간다. 그 과정에서 반복해서 만나게 되는 것이 있다.

경계를 명시하라고 가르쳐온 분야가, 자기 규범의 출처는 명시하지 않았다.

이것은 조롱이 아니라 관찰이다. 그리고 이 관찰이 8장에서 한 번 더 반복된다. 그때는 목록이 아니라 칙령이다.

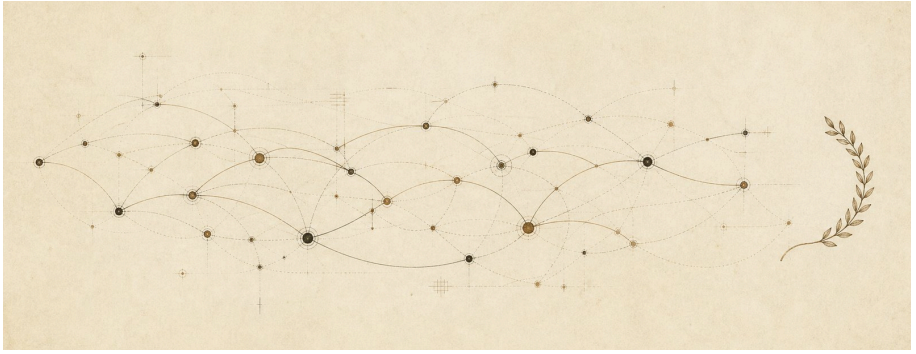
지금 우리가 에이전트에게 주는 지침들 — 어떤 파일을 읽어야 하는지, 어디까지 고쳐도 되는지, 무엇을 먼저 확인해야 하는지 — 은 어디에 명시되어 있는가. 그리고 5년 뒤에 그 출처를 되짚을 수 있는가.

주

1 원문 순서: The network is reliable / Latency is zero / Bandwidth is infinite / The network is secure / Topology doesn't change / There is one administrator / Transport cost is zero / The network is homogeneous. ↩

2 Arnon Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained", 2006. 위키백과의 정리: 〈https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing〉 · 해설: 〈https://sookocheff.com/post/distributed-systems/unpacking-the-eight-fallacies-of-distributed-computing/〉 ↩

3 이 책은 귀속의 어느 판본도 확정하지 않는다. 확정할 1차 문서가 없다는 것이 이 장의 논점이기 때문이다. "Deutsch 1994 논문"이라고 쓰는 것은 명백한 오류다 — 그런 논문은 존재하지 않는다. ↩



7 장

승리하지 않은 승리

2000년, 어바인

필딩(Roy Thomas Fielding)의 박사논문 5장에 이름이 붙는다. 표현 상태 전이(Representational State Transfer). 줄여서 REST.[1]

> **로이 필딩 (Roy T. Fielding, 1965-)**
>
> 캘리포니아 대학 어바인 캠퍼스에서 박사학위를 받은 컴퓨터과학자. HTTP/1.0과 1.1 명세의 주저자 중 하나이고, 아파치 HTTP 서버 프로젝트의 공동창립자이며, 아파치 소프트웨어 재단의 초대 의장이었다. 즉 그는 REST를 사후에 설명한 관찰자가 아니라 **웹의 프로토콜을 직접 쓴 사람**이다. 그 점이 2008년의 항의를 무겁게 만든다.

논문이 하는 일은 웹이 왜 작동하는지를 설명하는 것이다. 하이퍼텍스트 시스템으로서의 웹이 가진 제약들 — 클라이언트-서버, 무상태(stateless),

캐시 가능성, 균일 인터페이스, 계층 구조 — 을 뽑아내고, 그 제약들이 왜 그런 확장성을 만들어냈는지를 논증한다.

> **REST의 여섯 제약**
>
> ① **클라이언트-서버** 분리 ② **무상태** — 요청 하나가 필요한 모든 정보를 담고, 서버는 요청 사이에 클라이언트 상태를 갖지 않는다 ③ **캐시 가능성** ④ **균일 인터페이스**(uniform interface) ⑤ **계층 시스템** — 중간에 프록시·게이트웨이가 끼어도 클라이언트는 모른다 ⑥**주문형 코드**(선택). 이 중 ④가 이 장의 주제이고, ④ 를 이루는 네 하위 제약 중 마지막이 HATEOAS — 하이퍼미디어가 애플리케이션 상태의 엔진이라는 것 — 이다.

핵심은 균일 인터페이스다. 자원(resource)마다 다른 조작 방법을 만들지 않는다. 모든 자원이 같은 몇 개의 동사로 다뤄지고, 자원이 무엇인지는 그것이 반환하는 표현(representation)이 알려준다. 그리고 다음에 무엇을 할 수 있는지는 그 표현 안의 링크가 알려준다.

마지막 부분이 이 장의 요점이 된다.

6년에서 10년

논문은 2000년이다. 업계가 REST라는 말을 쓰기 시작한 것은 그보다 한참 뒤다.

2000년대 초중반의 기업 시스템은 SOAP과 WS-* 스택 위에 있었다. XML 스키마와 WSDL과 UDDI. 명세가 두껍고 도구가 무거웠다. 그 스택을 만든 힘은 CORBA를 만든 힘과 비슷했다 — 벤더들의 합의와, 아직 아무도 만들어보지 않은 것에 대한 명세.

SOAP · WS-* · WSDL

SOAP(Simple Object Access Protocol)은 XML로 감싼 메시지를 주고받는 프로토콜, WSDL(Web Services Description Language)은 그 서비스의 인터페이스를 XML로 적는 언어, WS-*는 보안·트랜잭션·신뢰전송 등을 얹은 수십 개 확장 명세의 총칭이다. 2000년대 초 기업 통합의 표준이었고, 5장의 CORBA와 같은 병 — 위원회가 구현보다 명세를 먼저 만드는 — 을 앓았다. 지금은 레거시 통합 외에는 거의 쓰이지 않는다.

REST가 실제로 채택된 시기는 대략 2006년에서 2010년 사이로 잡힌다. 그것도 논문의 논증 때문이라기보다는 다른 것들 때문이었다. HTTP는 이미 어디에나 있었다. JSON이 XML보다 다루기 편했다. 웹 브라우저에서 바로 호출할 수 있었다. 문서를 읽지 않고도 URL을 보면 대충 무슨 뜻인지 짐작이 갔다.

2008년, 항의

그리고 필딩이 자기 블로그에 글을 하나 쓴다.

제목은 이랬다. REST API는 하이퍼텍스트 주도여야 한다(REST APIs must be hypertext-driven).[2]

그가 항의한 것은 업계가 REST라고 부르는 것들이 REST가 아니라는 점이었다. URL을 예쁘게 만들고 HTTP 동사를 쓴다고 REST가 되는 것이 아니다. 응답 안에 다음에 무엇을 할 수 있는지가 들어 있어야 한다. 클라이언트가 URL 구조를 미리 알고 조립하는 방식은, 서버와 클라이언트를 단단히 묶는 것이고 균일 인터페이스의 요점을 버리는 것이다.

HATEOAS

Hypermedia As The Engine Of Application State. 클라이언트가 다음에 할 수 있는 행동을 **응답 안의 링크로부터** 알아내야 한다는 제약. 예를 들어 주문 조회 응답에 "이 주문은 취소 가능하며 그 주소는 여기"라는 링크가 들어 있어야 하고, 클라이언트가 `/orders/{id}/cancel` 이라는 규칙을 미리 알고 조립해서는 안 된다. 이 제약을 지키면 서버는 URL 구조를 자유롭게 바꿀 수 있다. 지키지 않으면 클라이언트 코드가 서버의 URL 규칙에 못 박힌다.

이 글은 널리 읽혔고, 널리 무시되었다.

2026년에도 대부분의 REST API는 필딩의 정의상 REST가 아니다. HTTP 위에 얹은 원격 프로시저 호출이다. 클라이언트는 문서를 보고 URL 규칙을 배우고, 그 규칙을 코드에 하드코딩한다.

무엇이 남았는가

그렇다고 REST가 실패한 것은 아니다. 이 부분이 미묘하다.

REST라는 이름으로 퍼진 것들 — HTTP 동사를 의미대로 쓰기, 자원 중심으로 URL 짜기, 상태를 서버에 두지 않기, JSON으로 주고받기 — 은 SOAP 스택보다 훨씬 나은 결과를 만들어냈다. 진입 장벽이 낮았고, 도구가 가벼웠고, 디버깅이 쉬웠다.

논문의 절반이 채택되었다. 나머지 절반은 채택되지 않았다.

채택된 절반은 **당장 이득이 보이는** 것들이었다. 채택되지 않은 절반은 **나중에 이득이 보이는** 것이었다. 하이퍼텍스트 주도 방식의 이득은

클라이언트와 서버를 따로 진화시킬 수 있다는 것인데, 그 이득은 몇 년 뒤에야 청구된다.

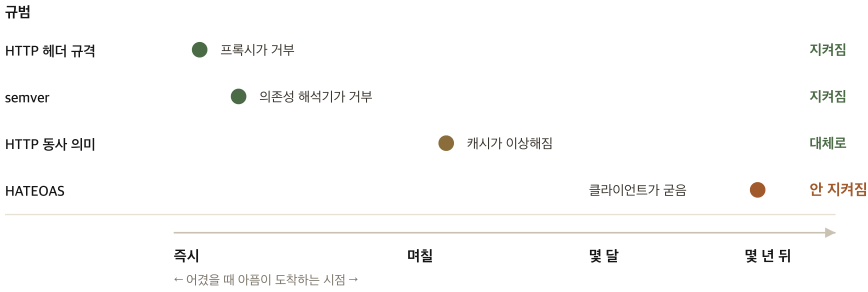


그림 7 — 계약이 지켜지려면 무엇이 필요한가. 어겼을 때 즉시 아픈 것만 지켜졌다.

계약과 강제

이 장이 이 책의 논지에서 차지하는 자리가 여기다.

REST는 계약이었다. 명세가 있고, 논증이 있고, 저자가 직접 오해를 정정했다. 그런데 그 계약을 **강제하는 것이 없었다**.

서버가 하이퍼텍스트 주도가 아닌 응답을 보내도 아무 일도 일어나지 않는다. 컴파일러가 막지 않는다. 테스트가 실패하지 않는다. 클라이언트가 URL을 하드코딩해도 그날은 잘 돌아간다. 청구서는 2년 뒤 API 버전을 올릴 때 온다.

같은 시기에 강제된 계약도 있었다. HTTP 자체의 규격은 강제된다. 잘못된 헤더를 보내면 프록시가 거부한다. 상태 코드를 엉뚱하게 쓰면 캐시가 이상하게 동작한다. 그 부분은 대체로 잘 지켜졌다.

계약은 선언된다고 계약이 되지 않는다. 강제될 때만 계약이다.

이 문장은 이 책에서 여러 번 돌아온다. 12장에서 한 회사가 이 문장을 자기 코드베이스에서 실험으로 확인하게 되고, 3부에서는 이 문장이 예측의 근거가 된다.

대가

강제되는 계약에도 값이 있다.

강제는 유연성을 죽인다. 컴파일러가 막는 것은 실수도 막지만 실험도 막는다. 타입 시스템이 엄격할수록 잘못된 코드가 줄고, 동시에 빠르게 시도해보는 일이 어려워진다.

REST가 느슨했기 때문에 퍼졌다는 해석도 가능하다. 아무나 시작할 수 있었고, 반쯤 맞게 해도 돌아갔다. 필딩이 원한 형태로 강제되었다면 진입 장벽이 SOAP만큼 높아졌을지도 모른다.

느슨한 계약은 퍼지고 굳지 않는다. 단단한 계약은 굳고 퍼지지 않는다.

남은 질문

2000년의 논문과 2008년의 항의 사이에 8년이 있다.

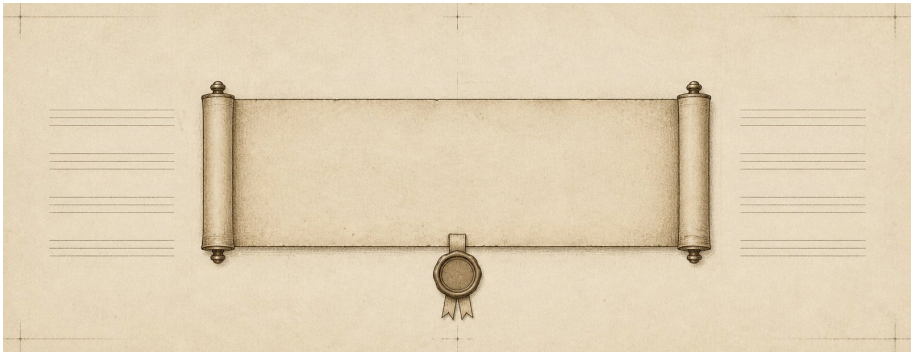
무엇이 그 8년 동안 REST를 다른 것으로 만들었는가. 사람들이 논문을 안 읽어서인가, 아니면 논문이 요구한 것이 그 시점의 도구로는 비용이 너무 컸기 때문인가.

그리고 그 질문은 이렇게 이어진다. 어떤 계약이 지켜지려면 무엇이 필요한가. 좋은 논증인가, 아니면 어겼을 때 즉시 아프게 하는 장치인가.

주

1 Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, 박사논문, University of California, Irvine, 2000. 5장이 REST다. 〈https://ics.uci.edu/~fielding/pubs/dissertation/top.htm〉 ↩

2 Roy T. Fielding, "REST APIs must be hypertext-driven", 2008-10-20. 〈https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven〉 — 원문의 첫 문장: "I am getting frustrated by the number of people calling any HTTP-based interface a REST API." ↩



8 장

문서 없는 칙령

여섯 줄

업계에서 가장 자주 인용되는 내부 지시가 있다.

모든 팀은 데이터와 기능을 서비스 인터페이스로 노출한다. 팀끼리는 그 인터페이스를 통해서만 소통한다. 다른 형태의 프로세스 간 통신은 허용되지 않는다. 직접 링크도, 다른 팀 데이터 저장소의 직접 읽기도, 공유 메모리도, 뒷문도 안 된다. 유일하게 허용되는 통신은 네트워크를 통한 서비스 인터페이스 호출이다. 어떤 기술을 쓰든 상관없다.

모든 서비스 인터페이스는 예외 없이 처음부터 외부에 노출 가능하도록 설계한다. 바깥세상의 개발자에게 그대로 열 수 있어야 한다.

그리고 마지막 줄. 이것을 하지 않는 사람은 해고된다.

베조스 API 칙령 (Bezos API Mandate)

아마존(Amazon) 창업자 제프 베조스(Jeff Bezos)가 2002년경 사내에 내렸다고 전해지는 지시. 모든 팀 간 통신을 서비스 인터페이스로만 하도록 강제해 아마존의 서비스 지향 아키텍처 전환을 이끌었고, 그 결과 내부 인프라를 외부에 팔 수 있게 되어 AWS가 나왔다는 것이 표준 서사다. **이 장이 다루는 것은 그 서사의 내용이 아니라 그 서사의 출처다.**

이 여섯 줄은 마이크로서비스를 설명하는 거의 모든 글에 나온다. 특히 다섯 번째 — 외부에 노출 가능하게 설계하라는 조항 — 이 뒤에 나올 AWS 이야기와 직접 이어진다. 플랫폼 전략을 논하는 글에도 나오고, API 우선 설계를 주장하는 글에도 나온다. 아마존이 어떻게 AWS를 만들 수 있었는지에 대한 표준적인 설명이 되었다.

출처

이 지시의 1차 문서는 존재하지 않는다.

유일한 출처는 2011년에 구글 내부에 작성된 메모다. 쓴 사람은 스티브 예게(Steve Yegge)로, 아마존에서 6년쯤 일하다가 구글로 옮긴 엔지니어다. 그 메모는 구글이 플랫폼을 이해하지 못하고 있다는 이야기를 하기 위해 쓰였고, 그 대비 사례로 아마존을 들면서 저 여섯 줄을 회상했다.[1]

스티브 예게의 "플랫폼 랜트" (Steve Yegge's Platforms Rant, 2011)

예게가 구글 사내 게시판에 올리려던 장문의 글이 실수로 공개 설정으로 올라가 인터넷에 퍼진 사건. 구글이 제품은 잘 만들지만 플랫폼은 이해하지 못한다는 비판이 본론이고, 아마존의 API 칙령은 대조군으로 등장한다. 예게 본인이 곧바로 내렸지만 이미 복제된 뒤였다. **업계 전체가 인용하는 그 조항들은 이 우발적 유출본이 유일한 출처다.**

그리고 그 메모는 실수로 공개되었다. 내부 게시물로 올린다는 것이 외부로 나갔다.

예게 본인이 시점에 대해 헤지를 달았다. 2002년쯤이었던 것 같다고, 플러스마이너스 1년이라고 적었다.

회상과 문서

이 차이가 중요한 이유는 무엇이 인용되고 있는지에 있다.

인용되는 것은 번호가 매겨진 조항들이다. 조항처럼 생겼고, 번호가 붙어 있고, 마지막에 처벌 조항이 있다. 법령의 형식이다.

실제로 존재하는 것은 한 사람이 9년 뒤에 다른 회사에서 기억을 되살려 쓴 문단이다. 그마저도 인용될 때 자주 다섯 개로 줄어든다 — 외부 노출 조항이 흔히 빠진다.

이 둘의 차이는 형식에 있다. 회상은 요점을 정리하면서 자연스럽게 조항의 모양을 갖추게 된다. 사람은 기억을 정돈해서 말하고, 정돈된 기억은 원래보다 깔끔하다.

"안 하는 사람은 해고" 라는 문장이 특히 그렇다. 그 문장은 이 이야기가 널리 퍼진 이유이기도 하다. 극적이기 때문이다. 그리고 그 문장 역시 같은 회상에서 나왔다.

그림 8 — 인용되는 것과 실제로 존재하는 것. 9년의 간격이 문단을 조항으로 바꿨다.

확인되는 것

예게의 회상에는 확인 가능한 요소도 있다. 그는 이 작업을 감독할 사람들이 지정되었고 릭 달젤(Rick Dalzell)이 그것을 이끌었다고 적었다. 예게가 그를 부른 호칭은 "월마트의 전 최고 고문관 겸 CIO"였다. 릭 달젤은 실존 인물이고, 월마트를 거쳐 아마존의 CIO를 지낸 것도 독립적으로 확인된다.[2]

그리고 아마존이 이후 몇 년에 걸쳐 서비스 지향 아키텍처(SOA)로 내부를 전환했다는 것은 별개로 잘 실증된다. AWS의 출현 자체가 그 증거에 가깝다. 내부 인프라를 인터페이스 뒤로 숨기지 않았다면 그것을 외부에 팔 수 없었을 것이다.

즉 아키텍처 전환은 실제로 일어났다. 조항의 텍스트가 재구성이다.

부인되지 않았다

한 분석은 이 상황을 이렇게 정리한다. 이 메모가 실제로 존재했는지 우리는 확신할 수 없다.

다만 베조스도 그의 최고 엔지니어들도 이것을 부인한 적이 없다. 그리고 사건의 정확성과 무관하게, 이 이야기는 기업 소프트웨어 개발의 방향을 바꿨다.

이 두 문장이 나란히 있는 상태가 이 장의 관심사다. 부인되지 않았다는 것은 확인되었다는 것과 다르다. 그리고 영향을 미쳤다는 것은 사실이라는 것과 다르다.

왜 이 이야기가 필요했는가

2000년대 중후반의 엔지니어들에게 이 이야기는 쓸모가 있었다.

서비스 경계를 긋자고 주장할 때, 그것이 왜 필요한지를 처음부터 논증하는 것은 어렵다. 분산의 대가는 즉시 보이고 이득은 나중에 온다. 회의실에서 그 논증을 이기기는 쉽지 않다.

그런데 아마존이 그렇게 해서 AWS가 되었다는 이야기는 논증을 대체한다. 결과가 이미 나와 있는 것처럼 보이고, 그 결과가 크다.

업계 전체가 인용하는 계약 규범이, 검증되지 않은 구전 위에 섰다.

6장과 같은 과

이 장은 6장과 같은 종류의 이야기다.

여덟 개의 오류에는 논문이 없었다. 여섯 줄의 칙령에는 문서가 없다. 둘 다 경계에 대한 규범이고, 둘 다 널리 가르쳐지고, 둘 다 출처를 되짚으면 사라진다.

그리고 이 책이 이것을 지적하는 방식에는 조건이 있다. 이 책 자체가 그 규범들에 기대고 있다. 여덟 개의 오류는 여전히 유용하고, 서비스 인터페이스로만 소통한다는 원칙도 여전히 유효하다.

지적하는 것은 그것들이 틀렸다는 게 아니라, **우리가 그것들을 어떻게 알게 되었는지 모른다**는 것이다.

대가

이 칙령이 실제로 있었다고 가정해도, 청구서는 붙는다.

모든 통신을 네트워크 호출로 강제하면 조직 안에 수백 개의 실패 지점이 생긴다. 함수 호출이었다면 실패하지 않았을 것들이 타임아웃을 내기 시작한다. 팀 사이의 협업이 API 협상이 되고, 협상에는 시간이 든다.

아마존이 그 값을 지불할 수 있었던 것은 규모 때문이었다는 해석이 있다. 팀이 수백 개일 때 조정 비용이 네트워크 비용을 넘어선다. 팀이 다섯 개일 때는 그렇지 않다.

같은 규범이 규모에 따라 이득이 되기도 하고 세금이 되기도 한다.

남은 질문

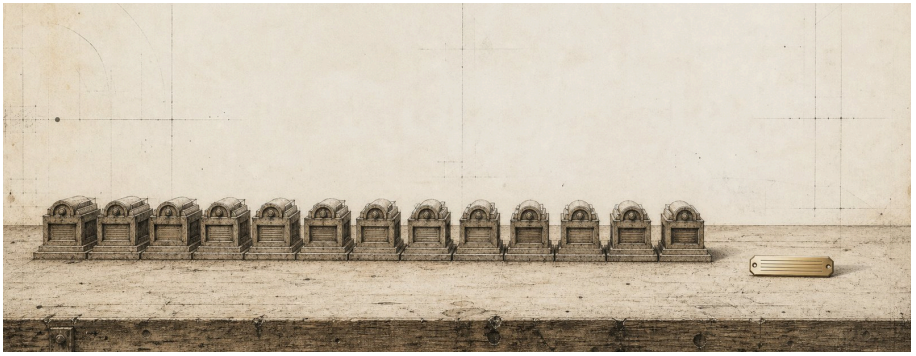
이 이야기가 사실이 아니어도 유용했다면, 유용함은 어디서 온 것인가.

그리고 지금 우리가 에이전트에게 주는 규범들 — 무엇을 읽어라, 어디까지 고쳐라, 무엇을 먼저 확인해라 — 중 얼마가 5년 뒤에 출처를 되짚을 수 있는 형태로 남을 것인가.

주

1 예게 메모의 널리 유통되는 사본: 〈https://gist.github.com/chitchcock/1281611〉 · 〈https://gist.github.com/kislayverma/d48b84db1ac5d737715e8319bd4dd368〉 — 원 게시물은 삭제되었고, 이 사본들의 상호 일치가 유일한 무결성 근거다. ↩

2 릭 달젤은 월마트를 거쳐 1997년 아마존에 합류해 2007년까지 CIO 겸 수석 부사장을 지냈다. 그의 재직 사실과 직책은 독립적으로 확인되지만, 그가 이 지시를 이끌었다는 진술의 출처는 여전히 예게의 회상 하나뿐이다. 예게의 원문이 그를 아마존이 아니라 **월마트**의 CIO로 소개한다는 점도 그대로 옮겨 둔다 — 회상이 인물을 어느 시점의 직함으로 부르는지까지가 그 회상의 성질이다. ↩



9 장

이름이 10년 늦게 왔다

2011년 5월, 베네치아 북쪽

소프트웨어 아키텍처 워크숍이라는 이름의 행사가 열린다. 베네치아에서 북쪽으로 조금 떨어진 곳이었다. 유럽과 미국과 아프리카에서 서른 명쯤이 초대되었다.

형식은 언컨퍼런스(unconference)였다. 사흘 동안, 참석자들이 직접 의제를 만든다.

언컨퍼런스 (unconference)

발표자와 프로그램을 미리 정하지 않고, 참석자들이 현장에서 의제를 제안하고 투표해 세션을 구성하는 회의 형식. 2000년대 초 기술 커뮤니티에서 퍼졌다. 이 장에서 중요한 이유는 **마이크로서비스라는 이름이 위원회의 산물이 아니라 이런 자리에서 나왔다는 것** — 5장의 CORBA와 정확히 반대 지점 — 이다.

거기서 반복해서 올라온 주제가 하나 있었다. 왜 시스템은 바꾸기가 이렇게 어려운가. 커진 제품을 어떻게 쪼개고 유지보수를 어떻게 개선할 것인가.

참석자 중에 마틴 파울러(Martin Fowler)가 있었다. 제임스 루이스(James Lewis)와 프레드 조지(Fred George)도 있었다. 셋 다 소트웍스(Thoughtworks)와 연결되어 있었다.

> **마틴 파울러 (Martin Fowler, 1963-) · 소트웍스**
>
> 영국 태생의 소프트웨어 작가·컨설턴트. 『리팩터링』(1999), 『엔터프라이즈 애플리케이션 아키텍처 패턴』(2002)의 저자이고, 애자일 선언(2001)의 서명자 17인 중 하나다. **소트웍스**(Thoughtworks)는 그가 수석 과학자로 있던 글로벌 소프트웨어 컨설팅 회사로, 지속적 배포(CD)·기술 레이더 등 여러 업계 관행의 발신지였다. 파울러의 웹사이트 `martinfowler.com`은 20년 넘게 소프트웨어 용어의 사실상 표준 정의처 역할을 해왔다.

그 워크숍에서 마이크로서비스(microservices)라는 말이 논의된다. 참석자 여럿이 최근에 각자 탐색하고 있던 아키텍처 스타일에 이름을 붙이기 위해서였다.

이듬해 5월, 같은 그룹이 그 이름을 확정한다.[1]

이미 있던 것

이름이 붙기 전에 관행이 있었다.

아마존은 2002년쯤부터 그 방향으로 가고 있었다. 넷플릭스(Netflix)는 2009년쯤부터 웹 규모에서 그것을 하고 있었고, 애드리안 콕크로프트(Adrian Cockcroft)는 그것을 세밀한 SOA(fine-grained SOA)라고 불렀다.

SOA (Service-Oriented Architecture, 서비스 지향 아키텍처)

시스템을 재사용 가능한 서비스 단위로 구성하고 네트워크 인터페이스로 조립하는 접근. 2000년대 중반의 기업 IT 유행어였고, 흔히 무거운 ESB(엔터프라이즈 서비스 버스)와 SOAP/WS-* 스택을 동반했다. 마이크로서비스는 SOA의 부정이 아니라 **더 잘게 나누고 중앙 버스를 없앤 판본**에 가깝다 — 그래서 콕크로프트가 자기가 하던 것을 "세밀한 SOA"라 부른 것이다. 파울러·루이스도 아티클에서 이 계보를 인정한다.

즉 사람들은 그것을 하고 있었고, 각자 다르게 부르고 있었다. 세밀한 SOA. 작은 서비스. 마이크로 웹 서비스(Micro-Web-Services). 피터 로저스(Peter Rodgers)가 2005년 웹 서비스 엣지 컨퍼런스에서 쓴 표현이 마지막 것이다. 프레드 조지는 2004년부터 소트웍스에서 관련된 실험을 하고 있었다.

관행이 2002년에 시작했고 이름이 2012년에 붙었다면 10년이다.

제목

루이스가 이 이야기를 처음 공개적으로 발표한 것은 2012년 3월, 크라쿠프의 33rd Degree 컨퍼런스에서였다.

발표 제목은 이랬다.

Microservices — Java, the Unix Way.[2]

프레드 조지도 비슷한 시기에 다른 스타일로 같은 이야기를 하고 있었다.

파울러와 루이스가 이 스타일의 특징을 정리한 아티클을 낸 것은 2014년 3월이다. 그 글에서 두 사람은 이것이 새로운 것이 아니라고 적었다. 유닉스 철학에 뿌리를 두고 있다는 것이었다.[3]

1장이 돌아온다

이 책은 1964년의 메모에서 시작했다.

프로그램을 정원 호스처럼 연결하자. 다른 방식으로 데이터를 주무를 필요가 생기면 세그먼트를 하나 더 돌려 끼운다. 안을 몰라도 꽂을 수 있다.

마이크로서비스가 자기 이름을 얻던 그 발표의 제목이 유닉스 방식이었다.

이것은 우연한 유사성이 아니라 선언이다. 그 스타일을 정리하던 사람들이 스스로 자기 계보를 지목했다. 우리는 새로운 것을 만든 게 아니라 파이프가 하던 일을 서비스 크기로 다시 하고 있다는 것.

무엇이 달라졌는가

같은 아이디어가 왜 2012년에 다시 필요해졌는가.

파이프의 세그먼트는 한 기계 안의 프로세스였다. 마이크로서비스의 세그먼트는 다른 기계에 있는 서비스다. 그 차이가 만들어내는 것은 두 가지다.

하나는 독립 배포다. 파이프로 연결된 프로그램들은 같은 기계에 같이 설치된다. 서비스로 연결된 것들은 각자 다른 시각에 배포될 수 있다. 이것이 2010년대에 가장 크게 팔린 이득이었다. 한 팀이 다른 팀을 기다리지 않고 배포한다.

다른 하나는 팀이다. 파이프의 세그먼트에는 주인이 없다. 서비스에는 주인이 있다. 이 스타일이 확산되면서 자주 인용된 기준이 하나의 서비스는 한 팀이 통째로 소유할 수 있는 크기여야 한다는 것이었다.

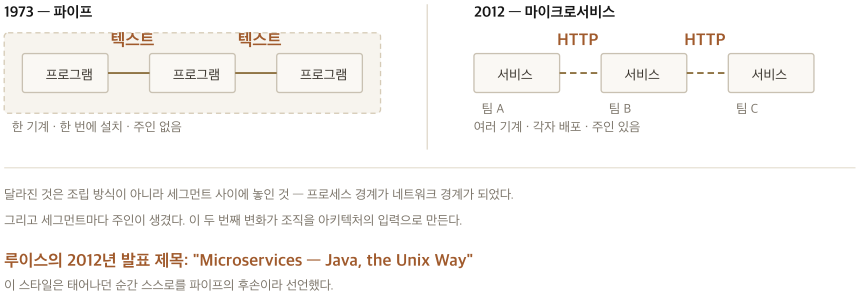


그림 9 — 파이프와 마이크로서비스. 같은 조립 방식, 다른 경계와 다른 주인.

3장의 콘웨이가 여기서 다시 나온다. 조직 구조가 시스템 구조를 정한다면, 원하는 시스템 구조를 먼저 정하고 조직을 그 모양으로 만들 수 있다. 역콘웨이 기동이라는 이름이 2010년대 중반에 붙는다.

조심할 것

여기서 흔히 쓰이는 인과 서술 하나를 이 책은 쓰지 않는다.

마이크로서비스가 콘웨이의 법칙을 역이용한 결과라는 서술이다. 그것은 사후에 정리된 이야기에 가깝다. 실제로 그 시기에 이 스타일을 밀어 올린 힘은 여러 개였다. 지속적 배포 관행의 성숙, 클라우드의 탄력적 자원, 그리고 조직 규모의 확대.

역콘웨이라는 개념이 이름을 얻은 것은 이 스타일이 퍼지고 난 뒤다. 원인으로 쓰기에는 순서가 맞지 않는다.

대가

마이크로서비스가 지불한 값은 이 스타일을 정리한 사람들 자신이 먼저 적었다.

분산 트랜잭션이 사라진다. 여러 서비스에 걸친 작업의 원자성을 데이터베이스가 보장해주지 않으므로, 보상 트랜잭션(compensating transaction)이나 최종 일관성(eventual consistency) 같은 것을 직접 설계해야 한다.

> **분산 트랜잭션과 그 대안**
>
> 한 데이터베이스 안에서는 여러 변경을 묶어 전부 성공하거나 전부 취소되게(원자성) 할 수 있다. 서비스가 나뉘면 그 보장이 사라진다. 대안은 두 가지다. **보상 트랜잭션** — 뒤 단계가 실패하면 앞 단계를 되돌리는 반대 작업을 명시적으로 실행한다(사가 패턴). **최종 일관성** — 잠깐 어긋난 상태를 허용하고, 시간이 지나면 맞춰지게 한다. 둘 다 데이터베이스가 공짜로 주던 것을 애플리케이션 코드가 직접 짜는 일이고, 둘 다 틀리기 쉽다.

운영 복잡도가 곱해진다. 서비스가 서른 개면 배포 파이프라인 서른 개, 모니터링 대시보드 서른 개, 온콜 대상 서른 개다.

디버깅이 흩어진다. 요청 하나가 다섯 서비스를 지나갔다면 그 이야기는 다섯 곳의 로그에 나뉘어 있다. 4장의 액터 모델이 이미 겪은 문제와 같은 종류다.

남은 질문

관행이 10년 앞서고 이름이 뒤따라왔다.

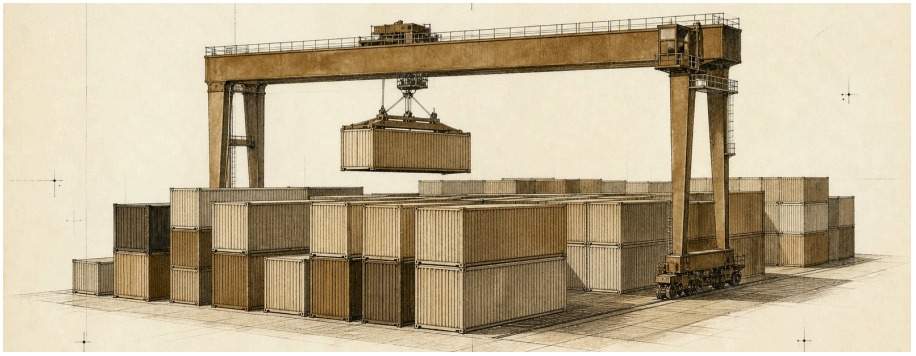
이 순서는 흔한가, 아니면 예외인가. 그리고 지금 우리가 에이전트를 여러 개 편성해서 하고 있는 일에는 아직 이름이 없다. 그 관행이 지금 몇 년째인가.

주

1 파울러·루이스 아티클의 각주에 명명 경위가 정리되어 있다. 2011-05 베네치아 근교 워크숍에서 논의 → 2012-05 같은 그룹이 이름 확정. ↩

2 James Lewis, "Microservices — Java, the Unix Way", 33rd Degree, 크라쿠프, 2012-03. 슬라이드: 〈https://www.slideshare.net/boicy/microservices-java-the-unix-way〉 ↩

3 James Lewis, Martin Fowler, "Microservices: a definition of this new architectural term", 2014-03-25. 〈https://martinfowler.com/articles/microservices.html〉 — "2014년에 파울러가 만든 용어"라는 통설은 부정확하다. 이 아티클은 명명이 아니라 **정식 문서화**다. ↩



10 장

세금을 낼 수 있게 되다

2013년 3월, 산타클라라

파이콘(PyCon)의 라이트닝 토크에서 솔로몬 하이크스(Solomon Hykes)가 도커(Docker)를 시연한다. 3월 15일이었다.[1]

그가 대표로 있던 회사는 도트클라우드(dotCloud)라는 이름의 PaaS 업체였다. 2008년 파리에서 시작해 2010년에 미국 법인이 되었고, 도커는 그 회사 내부에서 쓰던 기술이었다. 그달에 아파치 2.0 라이선스로 공개된다.

컨테이너 (container)

애플리케이션과 그것이 필요로 하는 라이브러리·설정을 하나의 이미지로 묶어, 어느 기계에서든 같은 방식으로 실행되게 하는 격리 단위. 가상 머신(VM)과 달리 운영체제 커널을 호스트와 공유하기 때문에 가볍고 빠르게 뜬다. 리눅스에서는 커널의 네임스페이스(namespace)와 컨트롤 그룹(cgroups)이 그 격리를 제공한다.

PaaS (Platform as a Service)

개발자가 서버·운영체제·런타임을 직접 관리하지 않고 코드만 올리면 실행되게 해주는 클라우드 서비스 층. 2007년의 헤로쿠(Heroku)가 대표적이다. 도트클라우드도 그런 회사였고, 도커는 **그 서비스를 굴리기 위한 내부 도구**였다 — 팔던 것이 아니라 만들다 나온 것이었다.

컨테이너 자체는 새롭지 않았다. 계보는 2000년의 FreeBSD Jails와 2005년의 솔라리스 Zones(Solaris Zones)까지 올라간다. 리눅스에는 LXC가 이미 있었다.

바뀐 것은 도구였다.

첫 테스트 릴리스가 나가고 한 달 만에 개발자 만 명이 쓰기 시작한다. 2014년의 1.0 시점에 다운로드 275만 회, 그로부터 1년 뒤 1억 회를 넘는다. 도트클라우드는 결국 회사 전체를 도커로 돌리고 이름도 바꾼다.[2]

1장에서 본 것과 같은 종류의 일이다. 컨테이너라는 개념은 13년 전부터 있었고, 8년 전부터 리눅스에서 돌고 있었다. 8년을 잡아먹은 것이 확신이었듯, 여기서도 없던 것은 개념이 아니라 **손에 잡히는 형태**였다.

2014년 6월, 마운틴뷰와 샌프란시스코

같은 해 여름, 구글이 쿠버네티스(Kubernetes)를 공개한다. 6월 6일에 발표 글이 올라가고 최초 커밋이 들어갔으며, 나흘 뒤 도커콘 무대에서 소개된다.[3]

계보는 보그(Borg)다. 2000년대 초부터 구글 내부에서 수천 대의 기계에 걸쳐 워크로드를 돌리던 컨테이너 관리 시스템이다. 그 경험이 오메가(Omega)를 거쳐 쿠버네티스로 나온다.

쿠버네티스 (Kubernetes, K8s)

여러 대의 기계를 하나의 자원 풀로 묶고, 그 위에 컨테이너를 어디에 몇 개 띄울지 자동으로 결정·유지하는 오케스트레이터. 사용자는 "이 서비스를 세 벌 돌려라"라고 **원하는 상태**만 선언하고, 시스템이 실제 상태를 거기에 맞춘다(reconciliation loop). 컨테이너 하나가 죽으면 다시 띄우고, 기계가 죽으면 다른 기계로 옮긴다. 이름은 그리스어로 조타수(κυβερνήτης)에서 왔고, K8s는 K와 s 사이 글자 8개를 줄인 표기다.

발표 시점의 상황이 이 장의 요점이다. 도커 덕분에 컨테이너화는 막 뜨고 있었는데, **그 컨테이너들을 여러 기계에 걸쳐 관리하는 표준적인 방법이 없었다.** 컨테이너 하나를 돌리는 것은 시작일 뿐이고, 진짜 문제는 프로덕션에서 여러 기계에 걸쳐 오케스트레이션하는 것이었다. 구글이 몇 년째 내부에서 하고 있던 바로 그 일이다.

마이크로소프트와 레드햇과 IBM과 도커가 7월 10일까지 커뮤니티에 합류한다. 이듬해 7월 21일에 1.0이 나오고, 같은 해에 구글과 리눅스 재단이 CNCF(Cloud Native Computing Foundation)를 만든다.

순서

이 두 사건의 순서를 9장 옆에 놓아 보면 무엇인가가 보인다.

마이크로서비스라는 이름이 확정된 것이 2012년 5월이다. 파울러와 루이스의 아티클이 2014년 3월이다. 도커가 2013년 3월, 쿠버네티스가 2014년 6월이다.

이름이 붙은 직후에 도구가 왔다.

그리고 그 도구가 온 다음에 이 스타일이 퍼졌다. 2014년 이후 몇 년 동안 마이크로서비스는 기업 아키텍처의 기본값에 가까워진다.

세금 계산서

마이크로서비스의 청구서는 9장 끝에서 이미 열거했다. 배포 파이프라인이 곱해지고, 모니터링이 곱해지고, 온콜이 곱해진다.

곱해진다는 것이 문제였다. 서비스가 서른 개면 그 서른 개를 각각 어디에 어떻게 올릴 것인지, 어떻게 재시작할 것인지, 하나가 죽으면 어떻게 알 것인지를 서른 번 풀어야 했다.

2012년의 도구로 그 값을 치르려면 조직에 전담 인력이 필요했다. 넷플릭스와 아마존은 그 인력이 있었다. 대부분의 회사에는 없었다.

컨테이너와 오케스트레이터가 한 일은 그 곱셈을 나눗셈으로 바꾼 것이다. 서비스 하나를 어떻게 배포할지 정의하면 서른 개가 같은 방식으로 배포된다. 죽으면 다시 뜬다. 어디에 뜰지는 스케줄러가 정한다.

그림 10 — 곱셈이 나눗셈이 된다. 도구가 세금을 낮추기 전까지 이 스타일은 넷플릭스 규모의 회사에만 가능했다.

마이크로서비스는 좋은 아이디어여서 퍼진 게 아니라, 그 세금을 낼 수 있게 된 다음에 퍼졌다.

이 문장이 하는 일

이 문장은 이 책의 후반부에서 방향을 바꿔 다시 쓰인다.

경계를 긋는 것과 그 경계를 유지하는 것은 다른 문제다. 좋은 경계를 아는 것으로는 부족하고, 그 경계를 유지하는 비용이 감당 가능해야 그 경계가 실제로 존재하게 된다.

그리고 이 관계는 반대로도 작동한다. 비용이 다시 비싸지면 경계는 사라진다. 23장에서 이 이야기를 하게 된다.

기계가 읽는 계약

이 시기에 조용히 자리 잡은 것이 하나 더 있다.

시맨틱 버저닝(Semantic Versioning, semver)이다. 톰 프레스턴워너(Tom Preston-Werner)가 2010년에 제안하고, 1.0.0이 2011년 9월, 2.0.0이 2013년 6월에 나온다. 규칙은 세 자리다. 메이저는 호환을 깨는 변경, 마이너는 호환되는 기능 추가, 패치는 호환되는 버그 수정.[4]

시맨틱 버저닝 (semver)

`MAJOR.MINOR.PATCH` 세 자리로 버전을 매기고, **각 자리가 무엇을 약속하는지**를 규격화한 것. `2.4.1`에서 `2.5.0`으로 올라가면 기존 코드는 그대로 돌아야 하고, `3.0.0`이면 깨질 수 있다. 이 규칙의 특이한 점은 강제자가 사람이 아니라는 것이다 — npm·Cargo·RubyGems의 의존성 해석기가 이 숫자를 읽고 자동으로 올릴지 말지를 정한다. 톰 프레스턴워너는 깃허브(GitHub)의 공동창업자다.

이것이 이 책의 관심사인 이유는 강제 방식에 있다. semver를 지키는지 검사하는 사람은 없다. 대신 npm과 RubyGems와 Cargo의 의존성 해석기가 그 숫자를 읽고 자동으로 판단한다. 어떤 버전을 올려도 되는지, 어떤 것은 사람이 확인해야 하는지를 도구가 정한다.

사람의 눈이 아니라 기계가 계약을 읽는다. 지금까지 실행된 것 중 가장 큰 규모의 그런 실험이다.

7장의 문장이 여기서 반증된다기보다 보완된다. 계약은 강제될 때만 계약인데, semver를 강제하는 것은 컴파일러도 법도 아니고 **패키지 매니저가 그 규칙대로 행동한다는 사실**이다.

대가

컨테이너가 값을 싸게 만든 대신 다른 값이 생겼다.

추상화가 한 층 늘었다. 프로세스가 컨테이너 안에 있고, 컨테이너가 파드(pod) 안에 있고, 파드가 노드 위에 있고, 노드가 클러스터 안에 있다. 문제가 생겼을 때 어느 층인지 찾는 일이 새 기술이 되었다.

그리고 오케스트레이터를 운영하는 것 자체가 전문 영역이 되었다. 세금을 낼 수 있게 만든 도구가 자기 몫의 세금을 걷기 시작했다.

남은 질문

2013년과 2014년의 두 사건이 없었다면 마이크로서비스는 어떻게 되었을까.

이름이 붙은 채로 소수의 큰 회사에만 남았을 것인가, 아니면 다른 형태의 도구가 나왔을 것인가.

그리고 지금, 기계가 코드를 쓰는 시대에 경계를 유지하는 비용을 낮춰줄 도구는 무엇인가. 그것은 이미 와 있는가.

주

1 Solomon Hykes, "The future of Linux Containers", PyCon US 2013 라이트닝 토크, 2013-03-15, 산타클라라. (컨퍼런스 자체는 3월 13일부터 21일까지였고 13일은 튜토리얼 날이다 — 2차 자료가 흔히 3월 13일로 적는 원인이다.) 영상: 〈https://www.youtube.com/watch?v=wW9CAH9nSLs〉 ↩

2 도커의 초기 채택 수치는 도커 사의 공식 발표와 2014~2015년 언론 보도에 기반한다. 이 책은 이 수치를 논증의 하중에 쓰지 않는다 — 방향(급격한 채택)만 취한다. ↩

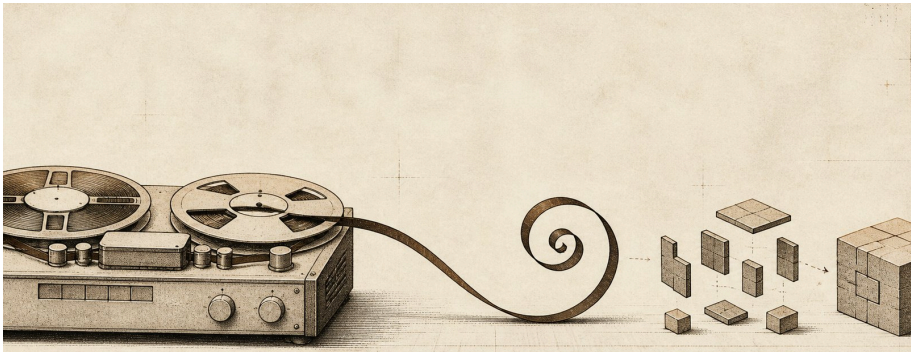
3 쿠버네티스의 보그 계보는 다음 논문에 정리되어 있다. Abhishek Verma et al., "Large-scale cluster management at Google with Borg", *EuroSys '15*, 2015. 〈https://research.google/pubs/large-scale-cluster-management-at-google-with-borg/〉 — 2014-06-06은 구글의 발표 글과 저장소 최초 커밋 날짜이고, 도커콘 2014(6월 9~10일) 무대 공개는 6월 10일 에릭 브루어의 키노트다. ↩

4 〈https://semver.org/〉 — 2.0.0 명세. 일부 2차 자료가 2013년을 semver의 기원으로 적으나, 2013-06은 2.0.0 발행 시점이고 최초 제안은 2010년이다. ↩

2 부

혼돈

되감는 사람들 — AI가 오기 전에 이미 시작된 혼돈.



11장

되감는 사람들

2026년의 층

지금 무엇을 쓰고 있느냐는 질문에는 하나의 답이 없다. 층으로 나뉘어 있다.

가장 아래층에서 논쟁은 끝났다. 컨테이너와 쿠버네티스는 기본값이다. 클라우드 네이티브 개발자 조사에서 백엔드 개발자의 77%가 클라우드 네이티브 기술을 최소 하나 쓴다고 답한다. 이 층은 더 이상 선택의 문제가 아니다.

그 위층은 다르다. 마이크로서비스를 쓰는 백엔드 개발자가 46%, API 게이트웨이가 50%다. 절반쯤이다. 여전히 크지만 기본값은 아니다.[1]

API 게이트웨이 (API gateway)

클라이언트와 여러 뒷단 서비스 사이에 놓여, 요청을 받아 알맞은 서비스로 넘기고 인증·속도 제한·로깅 같은 공통 관심사를 한 곳에서 처리하는 부품. 마이크로서비스를 쓰면 클라이언트가 상대해야 할 주소가 여러 개가 되는데, 게이트웨이가 그것을 하나로 되돌려 준다. 즉 **나눠서 생긴 문제를 부분적으로 되감는 장치**이고, 그래서 채택률이 마이크로서비스 자체보다 높다.

그리고 그 위에서 되감기가 진행 중이다.

42%

가장 널리 인용되는 숫자가 하나 있다. 마이크로서비스를 도입했던 조직의 약 42%가 일부 서비스를 다시 합치고 있다는 것이다. 이유로 꼽히는 것은 디버깅 복잡도, 운영 오버헤드, 사용자 경험에 영향을 준 네트워크 지연이다.

이 숫자를 이 책은 조심스럽게 쓴다.

원문을 따라가 보면 이 42%는 대부분 2차 인용이다. 재단 설문이나 컨설팅사 보고서를 블로그가 인용하고, 그 블로그를 다른 블로그가 인용한다. 그리고 소스끼리 어긋난다. 서비스 메시 채택 감소폭이 어떤 곳에서는 18%에서 8%로 적히고 다른 곳에서는 50%에서 42%로 적힌다. 후회율은 40%로도 나오고 60%로도 나온다.[2]

방향은 아마 맞을 것이다. 크기는 확인되지 않았다.

이 책은 이 숫자를 논증의 하중에 쓰지 않는다. 6장과 8장에서 지적한 것과 같은 종류의 문제를 이 책이 반복하지 않기 위해서다.

되감김의 정확한 이름

되감기라는 표현도 조심할 필요가 있다.

일어나고 있는 일은 마이크로서비스의 폐기가 아니다. 어떤 조사에서도 채택률이 급락하지 않았다. 일어나는 일은 **원래 나뉠 필요가 없었던 것을 다시 합치는 것**에 가깝다.

2010년대 중후반에 많은 조직이 서비스를 너무 잘게 나눴다. 서비스 하나가 한 팀이 소유할 수 있는 크기여야 한다는 기준이 있었는데, 팀이 다섯 개인 회사가 서비스를 마흔 개 만들었다. 그러면 한 팀이 여덟 개를 소유하게 되고, 소유의 이득은 사라지고 세금만 남는다.

지금의 조정은 그 계산을 다시 하는 일이다.

잘못 읽힌 사례

이 되감기 이야기에서 가장 많이 인용되는 사례가 하나 있고, 그것은 잘못 읽혔다.

2023년 3월 22일, 프라임 비디오(Prime Video)의 엔지니어 한 사람이 회사 기술 블로그에 글을 올린다. 모놀리식 아키텍처로 옮겨서 비용을 90% 줄였다는 내용이었다.[3]

그 글이 며칠 만에 클라우드 네이티브 커뮤니티를 흔들었다. 마이크로서비스의 우월함을 배우며 자란 세대에게 그 주장은 충격이었다. 어떤 매체는 아마존이 AWS 서버리스를 버스 밑으로 던졌다는 제목을 달았다.

실제 내용은 이랬다.

프라임 비디오의 비디오 품질 분석(Video Quality Analysis, VQA) 팀이 만든 도구가 있다. 수천 개의 라이브 스트림을 실시간으로 분석해서 블록 손상, 화면 멈춤, 싱크 문제 같은 품질 결함을 잡아낸다. 미디어 컨버터, 결함 검출기, 실시간 알림의 세 단계로 되어 있었다.

원래 구조는 Step Functions와 Lambda와 S3의 조합이었다. 스트림 수천 개로 확장하자 비용이 치솟았다. Step Functions의 상태 전이 비용과 S3에 대한 다수의 Tier-1 호출이 원인이었다. 그리고 예상 부하의 약 5% 지점에서 확장 한계에 부딪혔다. 병목은 Step Functions의 오케스트레이션 관리였다. 스트림 1초마다 여러 번의 상태 전이가 일어나면서 계정 한도에 걸렸다.

서버리스와 AWS Step Functions · Lambda

서버리스(serverless)는 서버가 없다는 뜻이 아니라, 개발자가 서버를 프로비저닝·관리하지 않고 실행한 만큼만 과금되는 모델이다. Lambda는 함수 하나를 올리면 요청이 올 때마다 실행해 주는 서비스이고, Step Functions는 그런 함수들을 상태 기계로 엮어 순서대로 실행시켜 주는 오케스트레이터다. **과금이 상태 전이 횟수 단위**라는 점이 이 사례의 핵심이다 — 초당 수십 번씩 전이가 일어나는 워크로드에는 구조적으로 맞지 않는다.

팀이 한 일은 모든 구성 요소를 하나의 ECS 태스크 안에서 돌리는 것이었다. S3를 거치던 데이터 전송이 메모리 안에서 일어나게 되었다. 그리고 그 태스크를 여러 벌 복제해서 단일 인스턴스의 용량 한계를 넘겼다.

세 가지 정정

이 사례에서 세 가지가 잘못 옮겨졌다.

첫째, 회사 차원의 정책 전환이 아니라 **한 팀의 한 워크로드**다.

둘째, 이동한 방향이 마이크로서비스에서 모놀리스가 아니다. **서버리스 오케스트레이션에서 단일 프로세스**로다. 커뮤니티에서 가장 크게 나온 반박이 이것이었다. 서버리스와 마이크로서비스는 다른 축인데 둘이 섞여서 논의되고 있다는 것.

셋째, 아이러니가 하나 붙어 있다. 루비 온 레일즈(Ruby on Rails)를 만든 데이비드 하이네마이어 핸슨(David Heinemeier Hansson)이 지적했다. 아마존은 원래 서비스 지향 아키텍처의 포스터 차일드였다고.[4]

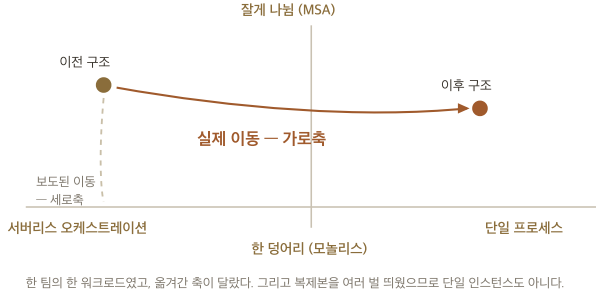


그림 11 — 프라임 비디오 사례가 실제로 이동한 축. "MSA → 모놀리스"는 두 축을 겹쳐 읽은 오독이다.

이 책이 이 사례를 쓰는 방식

이 책은 프라임 비디오를 모놀리스 회귀의 증거로 쓰지 않는다.

밈이 사실을 이긴 사례로 쓴다.

되감기의 증거가 필요하다면 더 깨끗한 카드들이 있다. 세그먼트(Segment)가 2018년에 쓴 「마이크로서비스여 안녕」, 우버(Uber)의 도메인 지향 마이크로서비스 아키텍처, 그리고 다음 장에서 다룰 쇼피파이(Shopify).[5]

대가

되감기에도 값이 붙는다.

합치면 배포가 다시 묶인다. 한 부분을 고치려면 전체를 다시 배포해야 하고, 한 부분의 버그가 전체를 세울 수 있다. 마이크로서비스가 해결하려던 문제가 그대로 돌아온다.

그래서 지금 자리 잡고 있는 형태는 중간항이다. 논리적 경계는 유지하고 물리적 분리는 하지 않는 것. 모듈러 모놀리스(modular monolith)라고 불린다. 한 프로세스 안에 있지만 모듈끼리 아무렇게나 부를 수는 없는 구조다.

> **모듈러 모놀리스 (modular monolith)**
>
> 하나의 배포 단위(한 프로세스, 한 저장소) 안에 있으면서도 내부가 명확한 모듈로 나뉘고, 모듈 사이의 호출이 정해진 인터페이스로만 일어나도록 제한된 구조. 마이크로서비스의 경계 이득은 취하고 네트워크 세금은 내지 않으려는 절충이다. **문제는 강제 수단**이다 — 프로세스가 나뉘어 있으면 물리가 막아 주지만, 한 프로세스 안에서는 무언가가 대신 막아야 한다. 다음 장이 그 무언가에 대한 기록이다.

그런데 한 프로세스 안에서 경계를 지키는 것은, 프로세스가 나뉘어 있을 때보다 어렵다. 네트워크가 강제해주지 않기 때문이다.

무엇이 그것을 강제하는가.

다음 장이 그 질문에 대한 한 회사의 답이다.

주

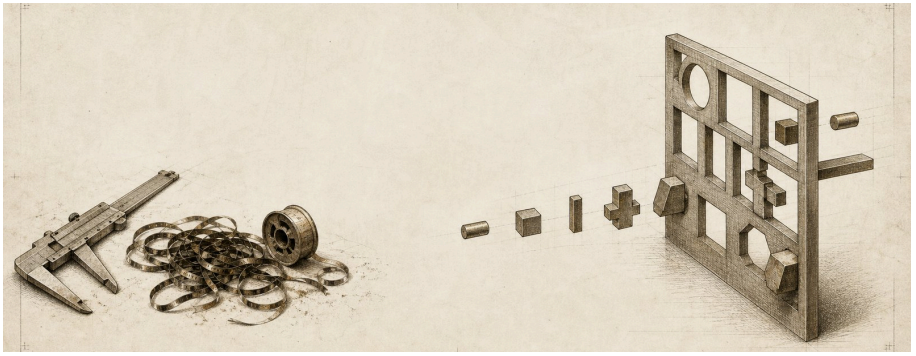
1 SlashData for CNCF, *State of Cloud Native Development, Q3 2025* (리포트 표기 2025-10, 발표 2025-11-11, KubeCon NA). 〈https://www.cncf.io/wp-content/uploads/2025/11/cncf_report_stateofcloud_111025a.pdf〉 — 세 수치 모두 **백엔드 개발자**(n=2,790) 기준이며 전체 응답자(n=12,021) 기준이 아니다. 77%는 "클라우드 네이티브와 강하게 연관된 기술을 **최소 하나** 쓴다"는 뜻이고, 리포트 자신의 "클라우드 네이티브에 해당한다"는 분류는 약 56%로 따로 있다. 조사 시기는 2025년 6~7월이고 2026-03에 후속판이 나왔으므로 이 수치는 한 사이클 지난 것이다. ↩

2 이 수치들의 원 출처를 추적한 결과는 `research/claims-ledger.md`의 L-11 항목에 정리되어 있다. 소스 간 충돌이 해소되지 않아 이 책은 등급을 [2차 인용, 검증 필요]로 둔다. ↩

3 Marcin Kolny, "Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%", Prime Video Tech, 2023-03-22. 원문은 이후 접근 경로가 바뀌었고, 널리 인용된 사본과 분석: 〈https://thenewstack.io/return-of-the-monolith-amazon-dumps-microservices-for-video-monitoring/〉 ↩

4 David Heinemeier Hansson, "Even Amazon can't make sense of serverless or microservices", 2023-05-04. 〈https://world.hey.com/dhh/even-amazon-can-t-make-sense-of-serverless-or-microservices-59625580〉 ↩

5 Alexandra Noonan, "Goodbye Microservices: From 100s of problem children to 1 superstar", Segment, 2018-07-10. 〈https://segment.com/blog/goodbye-microservices/〉 · Uber Engineering, "Introducing Domain-Oriented Microservice Architecture", 2020-07. 〈https://www.uber.com/blog/microservice-architecture/〉 ↩



12장

돌려보고 아는 것에서, 돌리기 전에 막는 것으로

2019년 2월, 오타와

쇼피파이(Shopify)의 기술 블로그에 커스틴 웨스테인데(Kirsten Westeinde)가 글을 올린다. 제목은 「모놀리스 해체하기」(Deconstructing the Monolith)였다.[1]

쇼피파이 (Shopify)

2006년 캐나다 오타와에서 시작한 전자상거래 플랫폼 회사. 상인이 온라인 상점을 열고 결제·배송·재고를 관리할 수 있게 해준다. 창업자들이 스노보드를 팔려다 쓸 만한 도구가 없어서 직접 만든 것이 시작이었고, 루비 온 레일즈로 지어졌다. 이 책에 등장하는 이유는 규모 때문이 아니라 **그들이 실패를 공개했기 때문**이다.

루비 온 레일즈 (Ruby on Rails)

2004년 데이비드 하이네마이어 핸슨이 공개한 웹 애플리케이션 프레임워크. "설정보다 관례"(convention over configuration)를 내세워 웹 개발 생산성을 크게 끌어올렸고 2000년대 후반 스타트업의 표준 도구가 됐다. 이 장에서 문제가 되는 성질이 하나 있다 — 웨스테인데의 표현으로 레일즈에서는 **애플리케이션 수준에서 모든 코드가 전역적으로 접근 가능하다**. 무엇이 무엇을 부르는지 언어가 제한하지 않는다.

글이 묘사하는 코드베이스는 이랬다. 현존하는 가장 큰 루비 온 레일즈 코드베이스 중 하나. 10년 넘게, 천 명 이상의 개발자가 손댔다. 상인에게 요금을 청구하고, 서드파티 앱을 관리하고, 상품을 갱신하고, 배송을 처리하는 기능들이 전부 들어 있다.

그리고 그 기능들 사이에 경계가 없었다.

경계가 없다는 것의 의미

배송료를 계산하는 코드가 결제를 처리하는 코드 옆에 살았다. 서로를 부르는 것을 막는 것이 거의 없었다.

시간이 지나면서 서로 다른 비즈니스 프로세스를 다루는 코드들 사이의 결합이 극단적으로 높아졌다. 무해해 보이는 변경 하나가 아무 관계 없는 테스트들을 연쇄적으로 깨뜨렸다. 단순한 수정에도 많은 맥락이 필요했다. 객체들이 얽혀 있어서 테스트를 쓰기가 고통스러웠고 실행이 느렸다.

웨스테인데가 남긴 문장이 하나 있다. 모놀리식 아키텍처는 구현하기 가장 쉽고, **아키텍처가 강제되지 않으면 결과는 십중팔구 모놀리스가 된다**는 것.[2]

경계가 없는 것이 기본값이다. 경계는 만들어야 생긴다.

이름

2017년 초에 작은 팀이 이 문제를 맡는다.

프로젝트의 처음 이름은 코어를 여러 조각으로 쪼개기(Break-Core-Up-Into-Multiple-Pieces)였다. 나중에 컴포넌트화(componentization)로 바뀐다.

그들이 먼저 손댄 것은 디렉터리 구조였다. 파일이 어디에 있는지가 그 파일이 무엇에 속하는지를 말하게 만드는 일이다.

마이크로서비스가 아니다

이 시점에서 명확한 선택이 하나 있었다.

마이크로서비스는 모놀리스 문제의 만능 해법처럼 마케팅되고 있었다. 쇼피파이가 적은 이유는 구체적이다. 배포 단위 수를 늘리지 않으면서 모듈성을 얻고 싶었다는 것. 마이크로서비스로 가면 서비스마다 테스트·배포 파이프라인과 인프라가 따로 생기고, 호출이 네트워크를 건너면서 지연이 붙고 신뢰성이 떨어지고, 서비스를 가로지르는 리팩터링과 배포 조율이 고통스러워진다.

그래서 그들이 고른 것은 모듈러 모놀리스다. 모든 코드가 하나의 코드베이스에 남되, 컴포넌트 사이의 경계는 정의되고 존중된다.

11장 끝의 질문이 여기서 다시 온다. 한 프로세스 안에서, 네트워크가 강제해주지 않는 상태에서, 무엇이 그 경계를 지키는가.

웨지

그들은 도구를 만들었다.

이름은 웨지(Wedge)였다. CI에서 테스트 스위트가 도는 동안 루비의 트레이스포인트(tracepoint)에 걸어 **전체 콜그래프를 얻는다.** 거기서 컴포넌트 경계를 넘는 호출만 걸러내고, ActiveRecord 연관관계와 상속 정보 같은 코드 분석 결과를 함께 넘긴다. 웨지가 어떤 넘나듦이 허용되는지 판정하고, 컴포넌트별 위반 목록과 전체 점수를 낸다.

> **콜그래프 (call graph) · 정적 분석 vs 동적 분석**
>
> **콜그래프**는 "어떤 함수가 어떤 함수를 부르는가"를 그린 그래프다. 이것을 얻는 방법은 두 가지다. **정적 분석**은 코드를 실행하지 않고 읽어서 추론한다 — 빠르지만, 루비처럼 실행 시점에 메서드가 정해지는 동적 언어에서는 상당 부분을 놓친다. **동적 분석**은 실제로 돌려 보고 일어난 호출을 기록한다 — 실제로 일어난 것만 보므로 정확하지만, 실행되지 않은 경로는 못 보고 데이터가 폭발한다. 웨지는 후자를 골랐고, 후자의 대가에 걸렸다.

발상은 명료하다. 경계를 넘는 일이 실제로 일어나는지 알려면, 실제로 일어나는 것을 보면 된다.

그리고 이것은 관측이지 강제가 아니다. 웨지는 경계 위반을 막지 않는다. 위반이 있었다고 알려준다.

2020년 9월, 실토

18개월 뒤에 후속 글이 올라온다. 제목은 「해체 중: 쇼피파이 모놀리스의 현재 상태」(Under Deconstruction: The State of Shopify's Monolith)

였다.[3]

글은 원래의 아이디어는 대체로 유지된다고 적는다. 그리고 거의 모든 세부는 바뀌었다고 적는다.

웨지에 대한 부분이 이 책이 이 장을 쓰는 이유다. 폐기 사유가 셋이다.

첫째, 노이즈. 콜그래프 로깅은 데이터를 엄청나게 많이 만들어내고, 그 안에서 신호와 노이즈를 가려내기가 어렵다.

둘째, 귀속이 모호하다. 어떤 호출이 어느 컴포넌트에서 어느 컴포넌트로 간 것인지조차 불분명한 경우가 있다. 메서드가 A에 정의되어 있고, B의 클래스가 그것을 상속받았고, 그 상속받은 것이 C를 부른다면 — 이 호출의 출발지는 어디인가.

셋째, 한 시간이 넘게 걸렸다. 분석 하나가 한 시간을 넘기면 피드백 사이클로 쓸 수 없다.

결과에 대한 평가는 한 문장으로 남았다. 산출된 결과가 자주 쓸모없었다는 것.

경계를 지키려고 만든 도구가, 자기가 만든 데이터의 양과 모호함과 소요 시간에 묻혔다.

대체 — 그리고 그것은 둘이었다

여기서 정확히 읽어야 한다. 웨지가 남긴 자리를 메운 것은 하나가 아니라 둘이고, 서로 다른 일을 한다.

경계 검사는 Packwerk가 받았다. 새로 만든 정적 분석 도구로, 코드가 실행되지 않은 상태에서 **정적 상수 참조**를 분석한다. 가장 큰 코드베이스 전체를 몇 분 안에 훑고, 그래서 풀 리퀘스트 워크플로에 붙일 수 있다. 의존성 그래프나 컴포넌트 캡슐화를 깨는 변경을 **메인 브랜치에 들어오기 전에 거부한다.**

진입점의 입출력 계약은 소르베(Sorbet)가 받았다. 원래는 dry-schema 위에 자체 제작한 `Component::Schema`라는 해시 스키마 검증 라이브러리를 썼는데, 깨지는 변경을 따라가기 어려웠고 복잡한 계약을 검사할 때 런타임 성능 문제가 있었다. 소르베로 옮긴 뒤 그것이 컴포넌트 경계에서 입출력 계약을 표현하는 기본 도구가 된다.

Packwerk와 소르베 (Sorbet) — 다른 일을 하는 두 도구

Packwerk는 쇼피파이가 만들어 2020년 9월에 오픈소스로 공개한 루비용 정적 분석기다. "이 컴포넌트가 저 컴포넌트의 내부를 참조하는가"를 검사한다 — 즉 **경계 자체**를 지킨다. **소르베**(Sorbet)는 스트라이프(Stripe)가 만들어 2019년에 공개한 루비용 점진적 정적 타입 검사기로, 기존 코드에 타입을 원하는 곳부터 조금씩 붙일 수 있다. 쇼피파이에서 소르베가 맡은 것은 경계 위에서 오가는 **값의 모양** — 무엇을 주면 무엇이 나오는가 — 이다. 둘 다 루비에는 컴파일 단계가 없으므로 별도의 정적 검사로 돌고, CI와 에디터에서 걸린다. 〈https://github.com/Shopify/packwerk〉 · 〈https://sorbet.org/〉

그림 12 — 같은 목적, 다른 시점. 관측은 사람의 판독을 요구하고, 정적 검사는 요구하지 않는다.

차이는 시점과 성질에 있다.

웨지는 사후에 관측했다. 테스트가 다 돌고 난 뒤에 로그를 뒤져서 위반을 찾는다. 추론이 필요하고, 노이즈가 섞이고, 사람이 읽어야 한다.

Packwerk는 사전에 거부한다. 그리고 그 근거가 다르다. 정적 상수 참조는 **개발자가 명시적으로 써 넣은 것**이므로, 그것을 지적하면 무엇을 고쳐야 할지가 분명하다. 동적으로 관측한 호출은 그렇지 않다.

경계를 지키는 힘이 관측에서 기계 검증으로 옮겨갔다.

남긴 문장

이 이동에서 끌어낼 것이 두 가지 있다.

하나는 강제에 대한 것이다. 웨스테인데의 문장이 이미 그것을 말했다 — 아키텍처가 강제되지 않으면 결과는 모놀리스가 된다. 규칙이 문서에 있으면 지켜지지 않는다. 7장의 REST가 겪은 것과 같은 이야기다.

다른 하나는 후속 글에 있다. 중앙 팀이 좋은 설계를 한 번 만들어낸다 해도, **그 팀이 관심을 다른 데로 돌리는 순간 설계는 퇴화한다.**[4]

이 문장이 왜 무거운지는 그 안에 든 시간 개념 때문이다. 설계는 한 번 옳게 만들어 두는 것이 아니라 계속 유지되어야 하는 것이고, 유지의 주체는 바뀐다. 그렇다면 유지를 사람의 주의력에 맡기면 안 된다.

이 장이 이 책의 경첩인 이유

2020년 9월이다.

대규모 코딩 에이전트가 나오기 전이다. 이 결정은 AI 때문에 내려진 것이 아니다. 사람이 사람의 코드베이스에서, 사람의 주의력이 옮겨간다는 이유로 내린 결정이다.

그런데 그 결정의 논리 구조가 3부에서 다시 나온다.

경계가 왜 거기 있는지를 아는 상태가 유지되지 않는다. 그렇다면 경계는 그것을 아는 사람이 아니라 **그것을 모르는 존재도 어길 수 없는 형태**로 있어야 한다.

기계는 맥락 없이 들어오는 새 참여자의 극단적인 경우다. 매번 새로 들어오고, 매번 떠난다.

AI는 이 이동을 시작하지 않았다. 이미 시작된 이동의 이유를 극단으로 밀어붙일 뿐이다.

대가

기계 검증으로 경계를 강제하면 값을 치른다.

도입 자체가 노동이다. 2020년 9월 시점에 Packwerk는 메인 모놀리스의 컴포넌트 37개 중 **약 3분의 1**에만 적용되어 있었다. 쇼피파이 규모의 루비 코드베이스에 점진적 타입을 얹는 것도 수년짜리 작업이다.

그리고 검사기가 표현할 수 있는 것에는 한계가 있다. 이 함수는 결제 컴포넌트에서만 불려야 한다는 규칙은 정적 참조로 잡힌다. 이 함수는 트랜잭션 안에서만 불려야 한다는 규칙은 어렵다. 강제할 수 있는 것만 강제되고, 나머지는 여전히 문서와 규율에 남는다.

강제의 범위가 곧 경계의 범위가 된다.

남은 질문

쇼피파이가 웨지를 버린 이유 중 하나는 노이즈였다.

그 노이즈는 도구가 나빠서 생긴 것인가, 아니면 코드베이스가 실제로 그만큼 얽혀 있어서 생긴 것인가.

관측이 실패한 자리에서 강제가 성공했다면, 관측이 알려주려던 것은 애초에 무엇이었는가.

주

1 Kirsten Westeinde, "Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity", Shopify Engineering, 2019-02-21. 〈https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity〉 ↩

2 원문: "Monolithic architecture is the easiest to implement. If no architecture is enforced, the result will likely be a monolith." 레일즈에 대한 진단은 같은 글의 "due to the global availability of all code at an application level". ↩

3 Philip Müller, "Under Deconstruction: The State of Shopify's Monolith", Shopify Engineering, 2020-09-16. 〈https://shopify.engineering/shopify-monolith〉 — 웨지 폐기 사유 원문: "Call graph logging produces a lot of data, so it's hard to separate the signal from the noise." / "Sometimes it's not even clear which component a call is from or to." / "took over an hour to run, which doesn't make for a useful feedback cycle." Packwerk 도입: "we developed a new tool called Packwerk to analyze static constant references." 소르베가 대체한 것: "Initially, we built a hash schema validation library called Component::Schema based on dry-schema" → "we ran into problems keeping up with breaking changes and runtime performance for checking more complex contracts." ↩

4 같은 글. 원문: "Even if a centralized team could make it happen, the design would degrade once the team switches its focus to something else." ↩

3 부

작은 닫힌 계

기계의 상한이 요구하는 형태.

13장

일곱 개, 혹은 네 개

1956년

밀러(George A. Miller)의 논문 제목에는 숫자가 들어간다. 마법의 숫자 7 플러스마이너스 2. 부제는 정보 처리 용량의 몇 가지 한계에 관하여였다.[1]

조지 밀러 (George Armitage Miller, 1920-2012)

프린스턴과 하버드에서 가르친 미국 심리학자. 인지혁명(cognitive revolution)의 창시자 중 하나로 꼽힌다 — 당시 주류였던 행동주의가 "머릿속에서 무슨 일이 일어나는지는 과학의 대상이 아니다"라고 본 데 반해, 밀러는 인간을 **정보를 처리하는 시스템**으로 다뤘다. 이 장의 논문은 심리학에서 가장 많이 인용된 논문 중 하나이며, 그 위치가 이 논문이 오용된 규모의 원인이기도 하다. 워드넷(WordNet)의 창시자이기도 하다.

논문이 정리한 것은 인지 시스템이 처리할 수 있는 항목의 수에 한계가 있다는 것이었다. 일곱 개 안팎을 넘어가면 유지율이 떨어진다.

이 숫자는 널리 인용되었고, 널리 오용되었다. 메뉴 항목을 일곱 개로 제한하라거나 슬라이드 불릿을 일곱 개로 맞추라는 식의 조언이 여기서 나왔다.[2]

2001년

코완(Nelson Cowan)이 그 숫자를 다시 계산한다.

논문 제목이 대칭적이다. 단기기억의 마법의 숫자 4. 부제는 정신적 저장 용량의 재고찰.[3]

> **넬슨 코완 (Nelson Cowan, 1951-)**
>
> 미주리 대학의 인지심리학자. 작업 기억(working memory) 연구의 주요 인물이다. 그의 기여는 밀러를 반박한 것이 아니라 **밀러의 숫자가 무엇을 측정한 것인지 분리해 낸 것**이다 — 되뇌기와 덩어리 짓기를 차단하면 순수 용량이 훨씬 작게 나온다는 것을 여러 실험 계열을 가로질러 보였다.

핵심은 실험 설계에 있었다. 밀러의 7±2는 사람이 속으로 되뇌거나(rehearsal) 항목을 묶어서(chunking) 기억하는 것을 막지 않은 상태의 숫자다. 그 두 가지를 차단하면 진짜 한계는 4±1에 가깝다.

> **되뇌기와 덩어리 짓기 (rehearsal / chunking)**
>
> **되뇌기**는 속으로 반복해 기억을 유지하는 것 — 전화번호를 외울 때까지 중얼거리는 것이 이것이다. **덩어리 짓기**는 여러 항목을 의미 있는 한 단위로 묶는 것 — `010 1234 5678`은 11개 숫자가 아니라 3덩어리다. 이 둘이 있으면 겉보기 용량이 커진다. 코완이 한 것은 **두 전략을 실험적으로 봉쇄하고 남는 순수 용량을 재는 것**이었다.

그리고 여기서 이 책에 필요한 구분이 나온다. **저장할 수 있는 양과 조작할 수 있는 양이 다르다.**

작업 기억은 대략 일곱 개의 덩어리를 붙들 수 있고, 지속 시간은 20초쯤이다. 그런데 그 덩어리들에 대해 뭔가를 **동시에 처리해야 한다면** 한계는 두세 개, 많아야 네 개로 떨어진다.

가만히 들고 있는 것과 관계를 유지하며 따지는 것은 같은 일이 아니다.

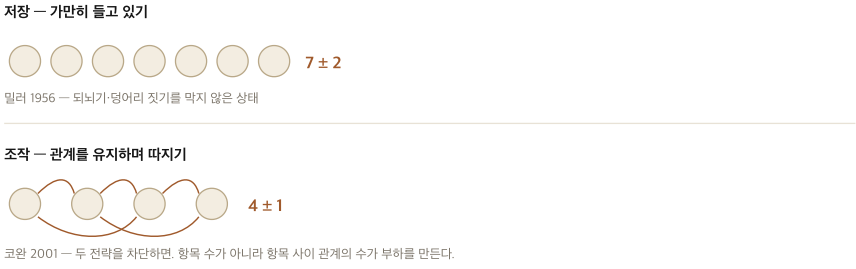


그림 13 — 저장과 조작은 다른 용량이다. 이 구분이 3부 전체의 뼈대가 된다.

1988년

스웰러(John Sweller)는 교육심리학자였다. 그가 풀려던 문제는 수학과 과학을, 특히 복잡한 문제를 어떻게 더 잘 가르칠 것인가였다.

1988년의 논문 제목은 「문제 해결 중의 인지부하: 학습에 미치는 효과」(Cognitive Load During Problem Solving: Effects on Learning)다.[4]

존 스웰러 (John Sweller, 1946-) · 인지부하 이론

호주 뉴사우스웨일스 대학의 교육심리학자. 인지부하 이론(Cognitive Load Theory)을 세웠다. 부하를 **내재적**(intrinsic, 문제 자체가 본래 가진 어려움)과 **외재적**(extraneous, 나쁜 설명·나쁜 구조가 덧붙인 불필요한 어려움)으로 나눈다.[5]

이 책에 결정적인 것은 그가 내재적 부하를 정의한 **방식**이다. 항목의 개수가 아니라 **요소 상호작용성**(element interactivity) — 동시에 고려해야만 이해가 성립하는 요소 쌍의 수 — 으로 정의한다. 각각 따로 배워도 되는 열 개는 부하가 낮고, 서로를 알아야만 뜻이 통하는 셋은 부하가 높다.[6]

두 부하를 가르는 기준도 여기서 나온다. **그것을 제거해도 해야 할 일이 그대로면 외재적이고, 아니면 내재적이다.** 코드베이스의 얽힘 중 어떤 것은 지울 수 있는 사고이고 어떤 것은 도메인이 실제로 결합돼 있어 설계로 지워지지 않는다 — 이 둘은 다른 것이다.

거기 나오는 문장이 이 책이 인용할 것이다.

인간의 단기기억은 심하게 제한되어 있고, 많은 수의 항목을 단기기억에 저장해야 하는 문제는 과도한 인지부하를 유발할 수 있다.

스웰러가 인지부하라고 부른 것은 작업 기억에 걸리는 요구다. 저장이든 처리든.

왜 이것이 아키텍처의 이야기인가

이 세 논문은 소프트웨어에 대한 것이 아니다. 기억과 학습에 대한 것이다.

그런데 우리가 모듈을 만든 이유의 절반이 여기 있다.

2부의 질문이 여기서 돌아온다. 관측이 실패한 자리에서 강제가 성공했다면, 관측이 알려주려던 것은 애초에 무엇이었는가. 답은 소프트웨어에 있지 않았다. 코드를 읽는 머리에 있었다.

파나스가 감추라고 한 이유를 다시 읽어 보면 그렇다. 바뀔 것 같은 결정을 모듈 안에 넣으면, 그 모듈 바깥에서 일하는 사람은 그 결정을 몰라도 된다. 몰라도 된다는 것은 머릿속에 담지 않아도 된다는 뜻이다.

한 팀이 통째로 소유할 수 있는 크기라는 마이크로서비스의 기준도 마찬가지다. 소유란 그 코드를 이해하고 있다는 뜻이고, 이해는 유한하다.

콘웨이가 말한 것도 결국 조정 비용의 한계다. 사람이 여섯 명이면 연결이 열다섯 개고, 열두 명이면 예순여섯 개다. 조합적으로 늘어난다.[7]

우리는 시스템이 복잡해서 나눈 게 아니다. **우리가 그 복잡성을 한 번에 담을 수 없어서 나눴다.**

대가

인지부하를 기준으로 나누면 값을 치른다.

첫째, 전체가 보이지 않는다. 조각을 이해할 수 있게 만든 대가로 전체를 이해하는 사람이 사라진다. 큰 회사에서 시스템 전체를 아는 사람이 없다는 상황은 이 교환의 결과다.

둘째, 경계를 넘는 문제에 대해 아무도 책임지지 않는다. 각 조각의 주인은 있는데, 조각 사이에서 생기는 문제의 주인은 없다.

셋째, 잘못 그은 경계를 고치기가 어렵다. 경계를 다시 그으려면 두 조각을 동시에 이해해야 하는데, 애초에 그것이 안 되어서 나눈 것이다.

상한이 옮겨간다면

이 장은 사람의 한계에 대한 것이다.

그리고 이 책이 하려는 질문은 그 다음이다.

코드를 쓰는 주체가 바뀌면 이 계산은 어떻게 되는가.

새 주체의 상한이 사람보다 훨씬 크다면 우리는 다시 합쳐도 된다. 상한이 사라진다면 나눌 이유 자체가 사라진다.

그런데 만약 새 주체에게도 상한이 있다면, 그리고 그 상한이 우리가 광고에서 보는 숫자와 다르다면.

그 질문에 답하기 전에 먼저 볼 것이 있다. **사람은 이 상한 앞에서 실제로 무엇을 했는가.** 우리에게는 그 답을 몇백 년 치러 본 사례가 하나 있다.

주

1 George A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *Psychological Review* 63(2), 1956, pp. 81–97. 〈https://doi.org/10.1037/h0043158〉 ↩

2 밀러 본인은 논문 첫 문단에서 이 숫자에 시달려 왔다고 농담조로 적었고, "7"을 설계 지침으로 쓰라고 말한 적이 없다. UI 항목 수를 7개로 제한하라는 조언은 이 논문에서 따라 나오지 않는다 — 화면에 보이는 목록은 기억할 필요가 없기 때문이다. ↩

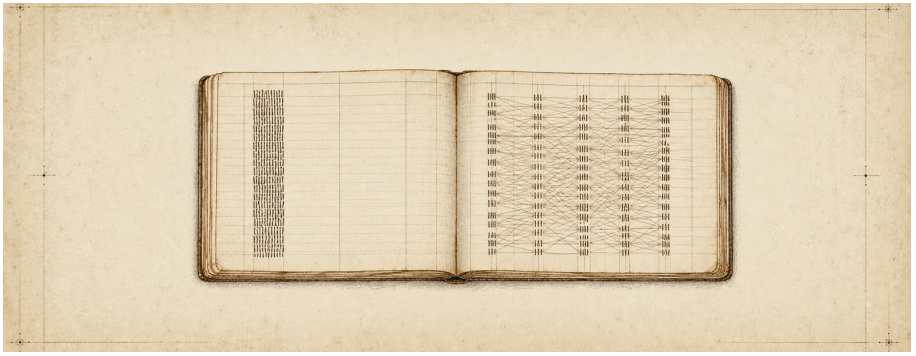
3 Nelson Cowan, "The magical number 4 in short-term memory: A reconsideration of mental storage capacity", *Behavioral and Brain Sciences* 24(1), 2001, pp. 87–114 (동료 논평 포함 pp. 87–185). 〈https://doi.org/10.1017/S0140525X01003922〉 ↩

4 John Sweller, "Cognitive Load During Problem Solving: Effects on Learning", *Cognitive Science* 12(2), 1988, pp. 257–285. 〈https://doi.org/10.1207/s15516709cog1202_4〉 ↩

5 1998년판 이론은 여기에 **본유적 부하**(germane load, 학습 자체에 쓰이는 부하)를 더한 3분류였다. 이 범주는 2010년대 재정식화에서 강등됐다 — 별도의 부하원이 아니라 내재적 부하를 처리하는 데 배분된 작업기억 자원으로 재해석됐다. 3분류로 인용하는 2차 자료가 여전히 많다. ↩

6 요소 상호작용성이 이 책의 "크기가 아니라 얽힘"과 같은 말이라는 점이 중요하다. 그리고 이 개념에는 이 책에 유리한 성질이 하나 더 있다 — 상호작용성은 자료의 절대 속성이 아니라 **자료 × 읽는 이의 사전지식**이다. 전문가는 여러 요소를 하나의 스키마로 접어 상호작용성 자체를 낮춘다. 19장의 "닫힘"이 읽는 주체에 상대적인 이유가 여기 있다. 다만 넘지 말아야 할 선이 있다 — 인지부하 이론의 측정은 초심자의 스키마 획득을 학습 성과로 재는 것이고, 코드 유지보수는 스키마 획득이 아니다. **구조의 유비이지 측정치의 이전이 아니다.** ↩

7 n명 사이의 양방향 연결 수는 n(n−1)/2다. 6명이면 15개, 12명이면 66개. 브룩스가 『맨먼스 미신』에서 인력 추가가 일정을 늦춘다고 한 근거가 이 조합적 증가다. ↩



14장

회사는 그 거래의 가장 오래된 구현체다

1937년, 런던

스물여섯 살의 경제학자가 아무도 묻지 않던 질문을 논문으로 낸다.

왜 회사가 존재하는가.

이상한 질문처럼 들리지 않는다면, 당시 경제학이 세상을 어떻게 그리고 있었는지를 알아야 한다. 시장이 있고, 가격이 있고, 가격이 자원을 최적으로 배분한다. 그것이 전부인 그림이었다. 그렇다면 회사라는 것 — 안에서는 가격이 아니라 **명령**으로 자원이 움직이는 섬 — 은 왜 있는가. 모든 사람이 모든 일을 그때그때 계약해서 사고팔면 되지 않는가.

로널드 코스(Ronald Coase)의 답은 짧다. **시장을 쓰는 데 비용이 들기 때문이다.**[1]

로널드 코스 (Ronald Coase, 1910-2013)

영국 태생의 경제학자. 「기업의 본질」(The Nature of the Firm, 1937)을 스물일곱에 발표했고, 그 논문과 「사회적 비용의 문제」(1960)로 **1991년 노벨 경제학상**을 받았다. 두 논문 모두 수식이 거의 없고 산문으로 되어 있다. 그가 만든 개념이 **거래비용**(transaction cost)이고, 이후 조직경제학·제도경제학이 그 위에 섰다.

가격을 알아내는 데 비용이 든다. 계약을 협상하고 문서로 만드는 데 비용이 든다. 필요할 때마다 새로 사람을 구해 조건을 정하는 것보다, **한 번 고용해서 "시키는 일을 한다"는 하나의 계약으로 묶는 것**이 싸다. 그 지점부터 회사가 시장을 이긴다.

회사는 시장의 반대가 아니라 **시장이 너무 비쌀 때 쓰는 대체 수단**이다.

그런데 코스는 이 책 편이 아니다 — 적어도 그대로는

여기서 이 책이 하고 싶은 유혹적인 말이 하나 있다. *사람은 인지 한계 때문에 회사를 발명했다.* 13장에서 본 것처럼 한 사람이 담을 수 있는 양이 유한하니, 그것을 넘는 일을 하려고 조직을 만들었다는 이야기.

코스 원문을 열어 보면 그 말이 반대 방향을 가리킨다.

코스에게 회사가 **생기는** 이유는 거래비용이다. 인지 한계가 등장하는 자리는 따로 있다 — 회사가 **무한히 커지지 못하는** 이유로 나온다. 조직이 커지면 관리의 수확이 체감하고, 기업가가 생산요소를 가장 가치 있는 용도에 배치하는 데 실패하기 시작한다. 그 지점에서 회사의 확장이 멈춘다.

즉 인지 한계는 회사의 **엔진이 아니라 브레이크**다. 장부의 반대편에 있다.

이 책은 그 사실을 그대로 적는다. 유혹적인 문장 하나를 얻자고 1차 출처가 말하지 않은 것을 말하게 하면, 6장과 8장에서 지적한 바로 그 일을 하는 것이 된다.

그러면 무엇을 가져올 수 있는가

주장을 낮추면 살아난다. 그리고 낮춘 형태가 오히려 이 책에 더 쓸모 있다.

발생론적 주장 — "사람은 인지 한계 때문에 조직을 발명했다" — 은 코스가 반박한다. 하지만 **사례 주장**은 반박당하지 않는다.

> 조직은 이 거래의 가장 오래된 구현체다.

무슨 거래인가. 이것이다.

나누기는 해법이 아니라 거래다

이 책이 지금까지 "쪼갠다"고 말해 온 것을 정직하게 회계로 쓰면 이렇게 된다.

나눈다고 총 부하가 줄지 않는다. **머리당 부하는 줄고, 대신 없던 부하가 새로 생긴다** — 조각들 사이를 맞추는 일이다.

그 새 부하가 얼마나 빨리 자라는지는 알려져 있다. 사람이 여섯이면 오갈 수 있는 짝이 열다섯이고, 열둘이면 예순여섯이다. 인원에 선형이 아니라 **제곱에 가깝게** 는다.[2] 브룩스가 늦어진 프로젝트에 사람을 더 넣으면 더 늦어진다고 한 근거가 이것이다.

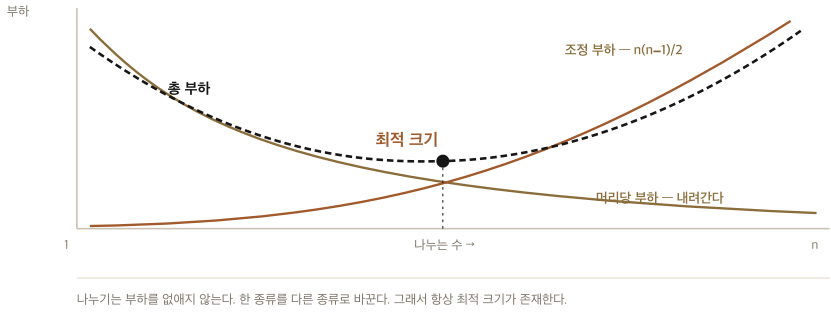


그림 14 — 나누기의 회계. 머리당 부하는 내려가고 조정 부하는 제곱으로 올라간다. 두 곡선이 만나는 자리가 최적 크기다.

두 곡선이 있으면 최저점이 있다. **모든 분산에는 최적 크기가 있고, 그것을 지나면 나눌수록 나빠진다.**

이 문장이 이 책의 앞뒤를 한꺼번에 설명한다. 13장의 되감기가 왜 일어나는지 — 팀 다섯 개인 회사가 서비스를 마흔 개 만들면 최저점을 한참 지나친 것이다. 그리고 20장에서 요약과 위임이 왜 기준선보다 나빴는지도 같은 그림이다.

150이라는 숫자를 이 책이 쓰지 않는 이유

조직 크기에 인지적 상한이 있다는 주장에는 유명한 숫자가 하나 붙어 있다. 던바의 수(Dunbar's number), 150이다.

로빈 던바(Robin Dunbar)는 영장류의 신피질 비율과 집단 크기의 상관을 재고, 그 회귀식에 사람의 값을 넣어 안정적으로 유지 가능한 관계의 수를 추정했다.[3]

이 책은 그 숫자를 근거로 쓰지 않는다. **점추정치는 150이지만 그 신뢰구간이 대략 4명에서 520명까지 벌어지기 때문이다.**[4] 구간이 두 자릿수 배로 열려 있는 추정치는 "사람의 조직에는 인지적 상한이 있다"는 문장을 지지하기에 너무 느슨하다.

이것을 밝히는 이유는 던바가 틀렸다고 말하기 위해서가 아니다. 이 책이 기계 쪽에서 "실효 범위는 광고의 몇 퍼센트"라는 단일 숫자를 거부한 것과 **같은 기준을 사람 쪽에도 적용하기 위해서**다. 한쪽에만 엄격한 기준은 기준이 아니다.

유비가 깨지는 곳 셋

이 장이 하려는 것은 사람 조직과 기계 편성을 나란히 놓는 일이다. 그러니 나란히 놓이지 **않는** 지점을 먼저 적어야 한다. 세 개가 있고, 마지막 것이 가장 무겁다.

첫째, 코스의 메커니즘이 기계에서는 반대를 가리킨다. 코스에서 회사가 이기는 이유는 지속적인 고용계약 하나가 수많은 개별 거래계약을 대체해 계약 비용이 분할상환되기 때문이다. 기계 편성에는 지속 계약이 없다. 매 호출이 일회성 거래이고 계약 비용은 0에 가깝다. **코스의 논리를 기계에 그대로 적용하면 "그러니 전부 시장으로 흩어라"가 나온다.** 회사가 아니라 회사의 해체를 가리킨다.

둘째, 사람의 상한은 고정이고 기계의 상한은 살 수 있다. 작업 기억의 네 덩어리는 돈으로도 버전업으로도 움직이지 않는다. 1956년에 측정된 것이 2026년에도 같다. 기계의 실효 범위는 움직인다. 고정 제약과 움직이는 제약

사이의 유비는 **특정 시점에서만** 유효하고, 이 책의 3년 시효가 그 사실의 자백이다.

셋째, 사람이 상한을 실제로 넘는 방법은 나누기가 아니라 기억이다. 이게 제일 아프다.

전문가가 초심자보다 나은 것은 일을 조금만 맡아서가 아니다. 장기기억에 쌓인 스키마가 여러 요소를 **한 덩어리로 접어** 요소 상호작용성 자체를 낮추기 때문이다. 13장에서 본 것처럼 부하를 만드는 것은 항목 수가 아니라 동시에 유지해야 하는 관계의 수인데, 숙련은 그 관계 자체를 하나로 접는다.

무상태 에이전트에게는 접힐 장기기억이 없다. 매번 새로 들어오고 매번 떠난다. 사람 조직이 상한을 넘는 핵심 기제가 기계 편성에는 통째로 빠져 있다.

이 세 번째 균열을 이 책은 약점으로 적지 않고 **논거로 쓴다.** 장기기억이 없다는 것은 곧, 사람이라면 숙련으로 흡수했을 얽힘을 기계는 흡수하지 못한다는 뜻이다. **사람에게는 접히는 복잡성이 기계에게는 그대로 남는다.** 그래서 지금 경계가 더 필요하다.

남은 질문

회사는 나누기의 대가를 몇백 년 치러 본 유일한 사례다.

그 사례가 남긴 교훈은 세 줄로 정리된다. 나누면 조정 비용이 생긴다. 그 비용은 제곱으로 자란다. 그래서 최적 크기가 있다.

그리고 이 책이 다음에 물어야 할 것은 이것이다. 기계에게도 상한이 있다면, **그 상한은 어디서 오는가.** 사람의 것은 생물학이었다. 기계의 것은 무엇인가.

다음 장이 그 자리를 판다.

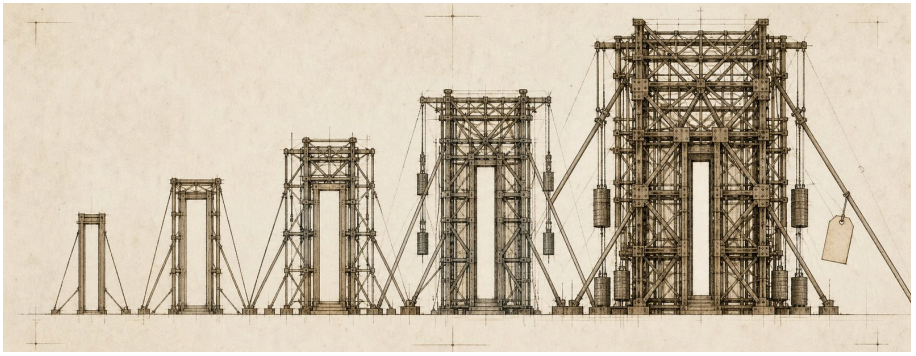
주

1 R. H. Coase, "The Nature of the Firm", *Economica* 4(16), 1937–11, pp. 386–405. DOI 10.1111/j.1468-0335.1937.tb00002.x — 회사의 존재 이유로 제시된 것은 "the cost of using the price mechanism"이다. 인지 한계에 해당하는 서술(관리의 수확체감, 기업가가 생산요소를 최적 용도에 배치하는 데 실패)은 회사가 존재하는 이유가 아니라 **회사의 크기가 멈추는 지점**을 설명하는 자리에 나온다. 이 방향을 뒤집어 인용하는 2차 자료가 흔하다. ↩

2 13장 주석에서 이미 본 n(n−1)/2다. Frederick P. Brooks Jr., *The Mythical Man-Month*, Addison-Wesley, 1975. ↩

3 R. I. M. Dunbar, "Coevolution of neocortical size, group size and language in humans", *Behavioral and Brain Sciences* 16(4), 1993, pp. 681–694. DOI 10.1017/S0140525X00032325 — 1992년 *Journal of Human Evolution* 논문이 선행판이다. ↩

4 Patrik Lindenfors, Andreas Wartel, Johan Lind, "'Dunbar's number' deconstructed", *Biology Letters* 17(5), 2021. DOI 10.1098/rsbl.2021.0158 — 원 회귀를 현대적 방법으로 재추정하면 점추정치가 데이터셋에 따라 크게 흔들리고 95% 신뢰구간이 한 자릿수에서 수백까지 벌어진다는 결과. 던바 본인은 이 재분석에 반론했고, 논쟁은 종결되지 않았다. ↩



15 장

창은 왜 그냥 커지지 않는가

회의실

기계의 상한이 어디서 오는지를 말하려면, 이 기계가 무엇을 하는 장치인지부터 한 문단으로 정리해야 한다. 비유 하나면 된다.

회의실이라고 하자. 발언이 하나 나올 때마다, 참석자 전원이 지금까지 나온 모든 발언을 각각 다시 읽는다. 그리고 각자 "지금 이 발언에 비추어 저 발언이 얼마나 중요한가"에 점수를 매긴다. 열 명이면 백 번 읽고, 백 명이면 만 번 읽는다.

이것이 트랜스포머(Transformer)의 셀프 어텐션(self-attention)이 하는 일이다. 비유가 아니라 거의 문자 그대로다 — 실제로 모든 토큰 쌍에 대해 점수를 계산해 **n × n 크기의 행렬**을 만든다.

트랜스포머와 셀프 어텐션 (Transformer / self-attention)

2017년 구글 논문 「Attention Is All You Need」가 제안한 신경망 구조. 지금 쓰이는 거의 모든 언어 모델이 이것의 변형이다.[1] 핵심은 **어텐션**이다 — 어떤 토큰을 처리할 때 입력 안의 다른 모든 토큰을 각각 얼마나 참조할지 가중치를 매기고, 그 가중치로 정보를 섞는다. 이전 구조(순환망)가 앞에서 뒤로 한 칸씩 훑어야 했던 것과 달리, 어텐션은 **거리에 무관하게 한 번에** 연결한다. 그것이 장문 처리를 가능하게 한 이유이고, 동시에 이 장이 다루는 대가의 출처다.

제곱

논문의 표 하나가 이 장의 출발점이다. 여러 층 구조를 나란히 놓고 계산 복잡도를 비교하는데, 셀프 어텐션 행에 적힌 것이 $O(n^2 \cdot d)$ 다.[2] n은 시퀀스 길이, d는 표현 차원이다.

입력이 10배 길어지면 어텐션 연산은 100배가 된다.

여기서 흔히 하는 실수를 미리 막아야 한다. 이 "100배"는 **어텐션 항에만** 맞고 모델 전체에는 틀리다. 트랜스포머에는 어텐션 말고도 층마다 붙는 계산이 있고, 그쪽은 길이에 선형이다. 그래서 실제로는 이렇게 된다.

1천 토큰에서 1만 토큰으로 갈 때 전체 연산은 약 11.7배다. 100배가 아니다. 그런데 10만에서 100만으로 갈 때는 약 69배가 된다.

제곱 항이 차지하는 비중이 길이에 따라 달라지기 때문이다. 1천 토큰 근처에서 어텐션은 전체의 2%도 안 된다. 100만 토큰에서는 95%가 된다.

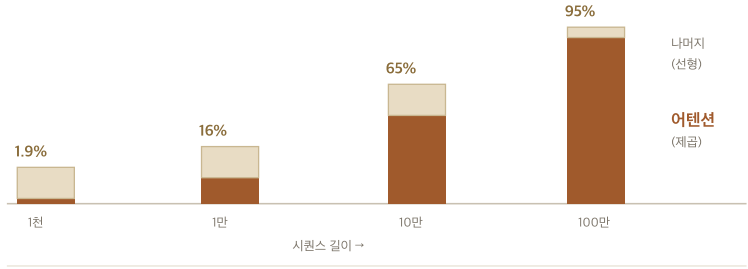


그림 15 — 제곱 항의 비중이 길이에 따라 뒤집힌다. 2017년에 아무도 이 문제를 걱정하지 않은 이유가 왼쪽 끝에 있다.

논문 자신이 그렇게 적었다. 셀프 어텐션 층은 시퀀스 길이 n이 표현 차원 d보다 작을 때 순환 층보다 빠르다고.[3] 2017년의 문장 길이에서는 그 조건이 늘 참이었다. **제곱은 그때 싸구려였다.**

그리고 진짜 벽은 메모리 쪽에 있다

연산량보다 먼저 부딪히는 것이 있다. 답을 한 글자씩 생성할 때, 모델은 앞서 계산한 중간값을 버리지 않고 들고 있어야 한다. 그것을 **KV 캐시**라고 부른다.

KV 캐시 (KV cache)

어텐션이 각 토큰에 대해 만들어 두는 키(Key)와 값(Value) 벡터를 저장해 두는 메모리. 답을 한 글자씩 만들 때 매번 전체를 다시 계산하지 않기 위한 장치다. 크기는 길이에 **선형**으로 늘지만, 계수가 크다 — 층 수 × 헤드 수 × 헤드 차원 × 2(K와 V) × 정밀도 바이트. **이것이 실무에서 가장 먼저 만나는 벽이다.** 연산은 기다리면 끝나지만 메모리는 없으면 아예 못 돈다.

계수가 얼마나 큰지는 숫자를 대입해 보면 안다. 널리 쓰이는 70B급 모델 하나를 잡으면, 100만 토큰짜리 대화 하나의 KV 캐시가 **수십 기가바이트** 규모가 된다.[4] 이것은 모델 가중치와 **별도**이고, **사용자 한 명당** 그렇다.

"GPU를 더 붙이면 되지 않나"가 순진한 답인 이유가 여기 있다. 사용자 수에 그대로 곱해지기 때문이다. 창을 열 배 키우면 같은 하드웨어로 받을 수 있는 동시 사용자가 십분의 일이 된다.

우회로는 있다 — 그리고 전부 뭔가를 판다

이 문제를 푸는 시도는 많았고 실제로 성공했다. 다만 무엇을 팔았는지를 봐야 한다.

플래시어텐션(FlashAttention)은 메모리를 오가는 횟수를 줄여 어텐션을 훨씬 빠르게 만든다.[5] 그런데 **점근 복잡도는 그대로** $O(n^2)$ 다. 상수를 줄인 것이지 지수를 바꾼 것이 아니다.

희소 어텐션은 모든 쌍을 보지 않고 일부만 본다. 연산은 선형에 가까워진다. 대가는 깊이로 청구된다 — 완전 어텐션이 한 층에서 하던 일을 하려면 층을 훨씬 많이 쌓아야 한다는 결과가 있다.[6]

고정 크기 상태를 쓰는 구조들은 길이와 무관하게 일정한 메모리로 돈다. 대가는 회상이다. 지나간 것을 정확히 되짚는 능력이 무너진다.[7]

그리고 가장 무거운 것.

광고된 창은 그 길이로 학습된 능력이 아니다

메타는 Llama 4 Scout를 발표하면서 "업계 최고 수준의 1000만 토큰 컨텍스트 윈도우"라고 적었다. 같은 발표문의 다른 문단에 이렇게 적혀 있다.

> 사전학습과 사후학습을 모두 25만 6천 컨텍스트 길이로 했다.[8]

1000만은 학습 길이의 39배 외삽이다. 그 길이로 배운 적이 없다.

이 구조가 이 분야의 표준이다. MiniMax는 100만으로 학습하고 추론에서 400만을 다룰 수 있다고 적었는데, 표준 벤치마크 수치는 100만까지만 보고하고 **400만은 건초더미 그림 한 장이 전부**다. 가능한 가장 약한 장문 평가다.[9]

1억 토큰을 주장한 회사도 있다. 벤치마크를 자기가 설계했고, 제품은 나오지 않았고, 발표 이후 2년째 침묵이다. 그 서사로 5억 달러 넘게 조달했다.[10]

그리고 그들이 되돌리고 있다

"무한히 커진다"에 대한 가장 강한 반증은 논문이 아니다. **벤더 자신의 후퇴**다.

MiniMax의 최신 모델은 100만이다. 400만이라는 문자열이 모델 카드에서 사라졌다. xAI의 최신 모델은 50만인데, **그보다 이전 모델이 100만이었다** — 뒤로 갔다. 메타는 1000만짜리 Scout 이후 15개월간 후속을 내지 않았다.

그리고 남은 것들이 한 자리에 모인다. 구글·앤트로픽·오픈AI·딥시크·Qwen·MiniMax가 **전부 100만 근처**다. 경쟁사들이 몇 퍼센트 안에서 같은

숫자에 수렴하는 것은, 우연이라기보다 **공유된 공학적·경제적 고원**으로 읽힌다.[11]

천장이 하나 더 있다 — 청구서

실무에서 가장 먼저 만나는 것은 연산도 품질도 아니다. 가격이다.

오픈AI의 현행 모델은 컨텍스트 105만을 광고하는데, 입력이 **27만 2천 토큰을 넘으면 그 요청 전체가** 입력 2배·출력 1.5배로 과금된다. xAI는 프롬프트가 20만에 닿으면 전 토큰이 상위 요율이다. 광고는 50만이다.

그리고 광고 숫자 자체가 쓸 수 있는 양이 아니다. 같은 오픈AI 모델의 최대 **입력**은 92만 2천이다 — 나머지는 출력과 추론용으로 예약된다.[12]

실사용 상한은 광고의 88%에서 시작해서, 가격 절벽에서 26%로 떨어진다.

여기서 멈춰야 한다 — 제곱은 "비싸다"이지 "틀린다"가 아니다

이 장이 지금까지 세운 것은 **비용** 주장이다. 컨텍스트를 키우면 연산과 메모리와 요금이 감당 안 되게 커진다는 것. 이건 강하고 1차 출처로 확인된다.

그런데 이 책의 3부가 필요로 하는 것은 다른 주장이다. **키운 만큼 잘 쓰지 못한다**는 것. 두 주장은 다르고, 앞의 것에서 뒤의 것이 따라 나오지 않는다.

반례가 명확하다. 플래시어텐션은 메모리 입출력을 열 배 가까이 줄이면서 **출력을 비트 단위로 동일하게** 유지한다. 정확한 어텐션이기 때문이다. 비용을 100배 아껴도 품질은 1점도 안 변한다. 역도 성립한다 — 예산이 무한하다면 $O(n^2)$는 품질을 깎지 않는다.

비용 축과 품질 축은 독립이다. 이 둘을 한 문장으로 뭉개면, 이 책이 부록 C에서 자백한 오류 — 정황을 증거로 격상하는 것 — 를 최상위 논증에서 저지르는 것이 된다.

그러면 다리는 어디 있는가. 두 개 있고, 둘 다 $O(n^2)$를 경유하지 않는다.

다리 ① — softmax는 언젠가 반드시 흩어진다

어텐션이 가중치를 매기는 방식에는 구조적 성질이 하나 있다. 가중치의 총합이 1로 고정된다.

여기서 흔히 쓰이는 논증이 하나 있는데, **그 형태로는 틀렸다.** *"합이 1이니 토큰이 늘면 각자 몫이 준다"* — 산술은 맞지만 어텐션은 평균을 내는 장치가 아니다. 한 헤드가 n이 얼마든 특정 키 하나에 0.9를 몰아줄 수 있다. 이대로 쓰면 *"그럼 점수를 크게 만들면 되잖아"* 한 줄에 무너진다.

빠진 단계는 **"점수가 위아래로 갇혀 있는가"** 다. 그리고 그 구멍을 메운 정리가 2024년에 나왔다.

로짓이 유계이면 n이 커질 때 모든 어텐션 계수가 1/n 규모로 흩어진다는 것(Lemma 2.1). 그리고 실제 트랜스포머에서 **입력 어휘가 유한하면 그 조건이 충족되므로**, 임의의 작은 값 ε에 대해 모든 계수가 ε 아래로 떨어지는 n이 반드시 존재한다(Theorem 2.2).[13]

증명된 것과 증명되지 않은 것을 나눠야 한다. 증명된 것은 *언젠가 반드시 흩어진다*이다. 증명되지 않은 것은 *지금 관측되는 저하가 이 때문이다*이다. 저자들 스스로 단서를 단다 — 실제 수치를 대입하면 경계가 유용하지 않을 만큼 느슨하고, 실측된 값의 퍼짐은 이론적 한계보다 훨씬 작다.

이 책은 그 선을 넘지 않는다. **구조적 상한이 존재한다는 것까지가 이 정리가 주는 것이다.**

다리 ② — 아무도 제곱을 정직하게 지불하지 않는다

두 번째 다리가 실제로는 더 무겁다.

$O(n^2)$가 품질을 직접 깎지는 않는다. 그런데 **아무도 $O(n^2)$를 그대로 내지 않는다.** 내면 서비스가 성립하지 않기 때문이다. 그래서 전부 앞 절의 우회로를 쓴다.

키와 값을 여러 헤드가 공유하게 만들고(제안자 본인이 "약간의 품질 저하"라고 적었다), 일부 쌍만 보게 하고, 고정 상태로 접고, 그리고 짧게 학습한 뒤 길게 늘려 붙인다.

비용이 타협을 강제하고, 타협이 품질을 깎는다. 사슬이 이렇게 이어진다. 이건 $O(n^2)$에서 품질로 가는 직행 논증이 아니라 **우회로**이고, 그래서 더 정직하다.

다만 이 우회로에도 한계가 있다. 어느 타협이 실효 범위 축소에 얼마나 기여했는지를 분해해 보인 연구를 이 책은 찾지 못했다. **그래서 이 사슬 전체는 가설로 표시한다.**

이 장이 남기는 것

이 장은 두 가지를 세웠다.

컨텍스트는 무한히 커질 수 없다. 이건 확정이다 — 연산이 제곱으로 늘고, 메모리가 사용자마다 곱해지고, 가격 절벽이 광고의 4분의 1 지점에 있고,

광고된 길이는 학습된 길이가 아니고, 벤더들이 이미 되돌리고 있다.

그리고 커지지 못하는 것과 잘 쓰지 못하는 것은 다른 문제다. 후자는 이 장이 증명하지 않았다. 다음 두 장이 그것을 **측정**으로 세운다.

사람의 상한은 생물학이었다. 기계의 상한은 이렇게 생겼다 — 산수와 메모리와 청구서와 공학적 타협이 겹쳐 만든 것. 성질이 완전히 다르다. 하나는 움직이지 않고 하나는 매년 움직인다.

같은 것은 상한이 있다는 사실뿐이다. 그리고 상한이 있으면 나눠야 한다는 것은, 14장에서 본 것처럼 거의 회계의 문제다.

주

1 Ashish Vaswani et al., "Attention Is All You Need", *NeurIPS 2017*. arXiv:1706.03762 〈https://arxiv.org/abs/1706.03762〉 ↩

2 같은 논문 Table 1. 층 유형별 층당 복잡도 비교에서 self-attention 행이 $O(n^2 \cdot d)$, recurrent 행이 $O(n \cdot d^2)$다. 언어 모델은 인과 마스크 때문에 뒤를 보지 못하므로 실제 쌍의 수는 $n^2/2$에 가깝다 — 절반이지만 여전히 제곱이다. ↩

3 같은 논문 §4의 문장: "self-attention layers are faster than recurrent layers when the sequence length n is smaller than the representation dimensionality d". 2017년의 전형적 설정에서 이 조건은 늘 참이었다. ↩

4 KV 캐시 크기 = 층 수 × KV 헤드 수 × 헤드 차원 × 2 × 정밀도 바이트 × 토큰 수. 70B급 모델의 전형적 구성(80층, GQA 8헤드, 헤드 차원 128, FP16)에 100만 토큰을 대입하면 대략 300GB대가 나온다. 그룹 질의 어텐션(GQA)이 이 계수를 크게 줄인 장치이고, 제안자 본인이 그 대가를 "minor quality degradation"이라 적었다. ↩

5 Tri Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", *NeurIPS 2022*. arXiv:2205.14135 — 이름의 **exact**가 핵심이다. 근사가 아니라 같은 값을 다른 순서로 계산한다. 그래서 출력이 동일하고, 그 동일성이 이 장 후반의 논증(비용≠품질)의 근거가 된다. ↩

6 Manzil Zaheer et al., "Big Bird: Transformers for Longer Sequences", *NeurIPS 2020*. arXiv:2007.14062 — 희소 어텐션으로 완전 어텐션의 표현력을 흉내 내려면 층 수가 크게 늘어야 한다는 결과가 논문 안에 있다. ↩

7 고정 크기 상태를 쓰는 구조에서 손실이 회상 과제에 집중된다는 분석이 여러 건 있다. 이 책은 그 계열의 개별 수치를 인용하지 않고 방향만 취한다. ↩

8 Meta AI, "The Llama 4 herd", 2025-04-05. 〈https://ai.meta.com/blog/llama-4-multimodal-intelligence/〉 — "industry-leading context window of 10M"과 "both pre-trained and post-trained with a 256K context length"가 같은 페이지에 있다. ↩

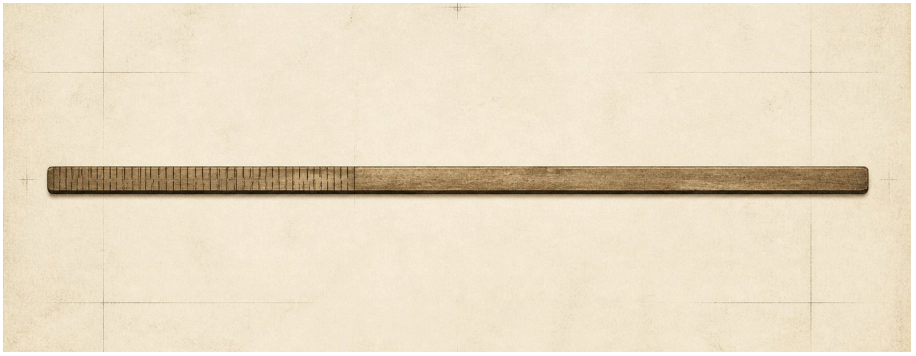
9 MiniMax-Text-01 모델 카드. "training context length is extended to 1 million tokens" / "can handle a context of up to 4 million tokens during the inference". RULER 결과는 100만까지 보고되고(1M에서 0.910), 400만은 건초더미 그림만 있다. 〈https://huggingface.co/MiniMaxAI/MiniMax-Text-01〉 ↩

10 Magic, "100M Token Context Windows", 2024-08-29. 〈https://magic.dev/blog/100m-token-context-windows〉 — 평가에 쓴 HashHop을 자사가 설계했고("we've designed HashHop"), 제품 출시·API·독립 평가가 없으며, 이 발표 이후 회사 블로그에 새 글이 없다. ↩

11 MiniMax-M3 모델 카드("1M context") · xAI 모델 문서(grok-4.6 500k, 이전 grok-4.3 1M) · 메타 공식 저장소에 Llama 4 이후 모델 없음. 전부 2026-08 시점 벤더 1차 문서. ↩

12 OpenAI 모델 문서(컨텍스트 1,050,000 / 최대 입력 922,000 / 272K 초과 시 요청 전체 할증) · xAI 모델 문서(200k 도달 시 전 토큰 상위 요율). 둘 다 2026-08 시점. ↩

13 Petar Veličković, Christos Perivolaropoulos, Federico Barbero, Razvan Pascanu, "softmax is not enough (for sharp size generalisation)", arXiv:2410.01104, ICML 2025. Lemma 2.1과 Theorem 2.2. 저자들이 직접 단 유보: 부동소수 범위를 대입하면 경계가 "별로 유익하지 않게 느슨"하고, 실측 로짓 스프레드는 이론 한계보다 훨씬 작다. ↩



16 장

광고된 것과 쓸 수 있는 것

두 개의 숫자

앞 장이 세운 것은 **창이 무한히 커질 수 없다**는 것이었다. 산수와 메모리와 청구서가 그것을 막는다. 그리고 그 장은 마지막에 선을 하나 그었다 — 커지지 못하는 것과 커진 만큼 잘 쓰지 못하는 것은 다른 문제이고, 후자는 아직 증명되지 않았다.

이 장부터 그것을 **측정**으로 세운다.

앞 장의 숫자가 **만드는 쪽**의 것 — 학습 길이와 비용 — 이었다면, 이 장의 숫자는 **쓰는 쪽**의 것이다. 광고된 길이 안에서, 품질은 어디까지 유지되는가.

모델 사양서에 적힌 숫자가 있다. 컨텍스트 윈도우 20만 토큰, 128만 토큰, 1000만 토큰.

그리고 다른 숫자가 있다. 그 안에서 품질이 실제로 유지되는 범위.

두 숫자는 같지 않다. 그리고 지금까지 벤치마크된 모든 모델에서 후자가 더 작았다.

> **토큰 (token)**
>
> 언어 모델이 텍스트를 다루는 최소 단위. 단어보다 작고 글자보다 크다 — 영어에서는 대략 4글자, 한국어에서는 대략 1~2글자가 한 토큰이다. 코드에서는 들여쓰기와 기호 때문에 밀도가 달라진다. 거칠게 잡아 **소스코드 1줄 ≈ 10~15토큰**, 즉 10만 줄짜리 프로젝트는 100만 토큰 규모이거나 그 이상이다. 컨텍스트 윈도우가 토큰 단위로 광고되는 이유이고, 18장의 숫자를 읽을 때 필요한 환산이다.

자를 만드는 법

엔비디아(NVIDIA)가 만든 벤치마크가 있다. 이름은 RULER다.[1]

방식은 이렇다. 검색(건초더미 변형), 다중 키-값 조회, 변수 추적, 집계, 질의응답 — 다섯 범주 열세 과제를 컨텍스트 길이를 늘려가며 시킨다. 그리고 기준선을 하나 정한다. Llama-2-7B가 4K 컨텍스트에서 내는 성능이다. 어떤 모델이 길이를 늘려가다가 그 기준선 아래로 떨어지는 지점이, 그 모델의 실효 컨텍스트(effective context length)다.

> **실효 컨텍스트 (effective context length)**
>
> "광고된 최대 길이"가 아니라 **품질이 유지되는 길이**. RULER의 정의는 구체적이다 — Llama-2-7B가 4K에서 내는 점수(85.6%)를 임계선으로 잡고, 길이를 늘려가다 그 아래로 떨어지기 직전 길이를 그 모델의 실효 컨텍스트로 삼는다. 절대 점수를 묻지 않고 **짧은 컨텍스트의 작은 모델만큼도 못 하게 되는 지점**을 묻는 것이 이 정의의 요령이다.

기준선을 이렇게 잡은 것이 이 벤치마크의 요령이다. 절대 점수를 묻지 않는다. 짧은 컨텍스트에서 잘하던 작은 모델만큼도 못 하게 되는 지점이 어디냐고 묻는다.

결과

열일곱 개 모델을 측정한 초기 연구의 숫자가 이랬다.

한 모델은 20만 토큰을 광고했고 실효는 3만 2천이었다. 16%다.

다른 모델은 12만 8천을 광고했고 실효는 6만 4천이었다. 50%다.

테스트한 모델 중 절반만 3만 2천 토큰에서 만족스러운 성능을 유지했다.

그리고 이 연구에서 가장 자주 인용되는 문장이 따로 있다. **건초더미에서 바늘 찾기 테스트에서는 완벽한 점수를 받은 모델들이, RULER의 다른 과제에서는 길이가 늘어나면 성능을 유지하지 못한다.**

여기서 한 가지를 명시해야 한다. 이 결과를 "쓸 수 있는 컨텍스트는 광고된 것의 절반쯤"이라는 경험칙으로 요약하는 글이 많다. **그 요약은 논문에 없고, 데이터가 그것을 지지하지도 않는다.** 같은 표 안에서 비율이 두 자릿수 배로 흩어진다. 가장 높은 것이 50%(광고 12만 8천, 실효 6만 4천)이고, 100만을 광고한 세 모델의 실효는 각각 6만 4천·1만 6천·4천 미만이다. 마지막 것은 0.4%가 안 된다.[2] 하나의 백분율로 줄일 수 있는 양이 아니다.

이 책이 이 표에서 취하는 것은 비율이 아니라 방향이다. **광고된 숫자는 상한이고, 실효는 그보다 작으며, 얼마나 작은지는 모델마다 두 자릿수 배로 다르다.**

건초더미의 문제

바늘 찾기 테스트가 왜 후한 점수를 주는지를 짚을 필요가 있다.

> **건초더미에서 바늘 찾기 (Needle in a Haystack, NIAH)**
>
> 긴 텍스트("건초더미") 안 어딘가에 무관한 문장 하나("바늘")를 심어 놓고, 모델이 그것을 찾아내는지 보는 테스트. 2023년에 널리 퍼져 긴 컨텍스트 모델의 표준 홍보 지표가 됐다. **문제는 이 과제가 너무 쉽다는 것**이다 — 질문과 정답 사이에 단어가 겹치므로 의미를 이해하지 않고 문자열만 맞춰도 풀린다. NIAH 만점은 "긴 컨텍스트를 잘 쓴다"의 증거가 아니라 "긴 컨텍스트에서 문자열 검색은 된다"의 증거다.

그 테스트는 긴 텍스트 안에 문장 하나를 심어놓고 그것을 찾게 한다. 심는 문장과 묻는 질문 사이에 **단어가 겹친다**. "샌프란시스코에서 가장 좋은 일은 샌드위치를 먹으며 앉아 있는 것"이라는 문장을 심고 "샌프란시스코에서 가장 좋은 일이 뭐냐"고 묻는다.

모델은 의미를 이해하지 않아도 된다. 겹치는 단어를 찾으면 된다.

단서를 뺀다

어도비 리서치(Adobe Research)와 뮌헨 대학(LMU Munich)이 만든 벤치마크가 그 겹침을 제거한다. 이름은 NoLiMa(No Literal Matching)다.[3]

질문과 정답 사이의 글자 그대로의 단어 겹침을 없앤다. 정답을 찾으려면 의미를 따라가야 한다.

결과가 이렇다.

열세 개 모델 중 열한 개가 3만 2천 토큰에서 짧은 컨텍스트 기준선의 50% 아래로 떨어진다.

그리고 개별 숫자 하나. 장문 처리에서 강한 편에 속하는 모델 하나가, 짧은 컨텍스트에서 99.3%를 내다가 3만 2천 토큰에서 69.7%로 내려간다. 그 모델이 광고한 길이는 12만 8천이고, 같은 논문이 매긴 실효 길이는 **8천**이다.

정리되어 있으면 낫지 않은가

또 다른 연구가 반직관적인 결과를 하나 내놓는다.

열여덟 개 프런티어 모델을 대상으로 입력 길이에 따른 저하를 측정했다. 전부 저하됐다. 과제 종류와, 질문과 근거의 의미적 유사도와, 방해 요소의 유무와, 건초더미의 구조에 따라 다르게 저하됐고, 어느 길이에서 무너진다고 말할 수 있는 고정된 임계값은 없었다. 다만 **근거의 위치는 이 실험에서 유의한 차이를 만들지 않았다.**[4]

그리고 이 부분이다. **논리적으로 잘 구조화된 문서보다 무작위로 섞은 문서에서 성능이 더 좋았다.**

실험 설계를 정확히 옮기면 이렇다. 건초더미 쪽에만 두 조건을 걸었다. 하나는 문단의 논리적 흐름을 그대로 둔 것, 다른 하나는 같은 주제의 문장들을 무작위로 재배열해 국소적 연결을 깨뜨린 것. **후자가 이겼다.** 저자들의 사전 예상은 반대였다.

저자들은 메커니즘 설명을 유보했다. 어텐션과 관계가 있을 수 있다고만 적고, 규명은 범위 밖이라고 명시한다.

그럴듯한 설명을 하나 붙일 수는 있다. 논리적 구조는 더 그럴듯한 방해 요소를 만든다는 것 — 관련 없는 문단이 관련 있는 문단과 비슷하게 생겼을 때 모델이 헷갈린다는. 다만 이것은 이 책의 추측이지 논문의 주장이 아니다.

이 결과를 코드베이스에 옮기면 불편한 함의가 나온다. 잘 정리된 대형 코드베이스가 어지러운 것보다 오히려 어려울 수 있다. 비슷하게 생긴 모듈이 많기 때문이다.

정리를 잘 하는 것이 답이 아니라는 뜻이다.

하나의 숫자가 아니다

실효 범위를 단일 퍼센트로 말하는 것에 대해서도 반론이 있다.

한 연구자가 최대 실효 컨텍스트 윈도우(Maximum Effective Context Window, MECW)라는 개념을 제안한다. 사실들을 여러 위치에 심어놓고 여러 종류의 과제로 찾게 한다. 못 찾으면 그 모델은 실효 범위를 넘은 것이다.[5]

과제는 네 종류였다. 바늘 하나 찾기, 바늘 여럿 찾아 더하기, 전체 합계 요약하기, 그리고 걸러서 정렬해 이어붙이기. 마지막 것이 가장 복잡한 과제라고 논문은 적는다.

결과는 단일 숫자 프레이밍을 깬다. **실효 범위가 과제 유형에 따라 달라진다.**

그리고 그 크기가 작다. 논문의 문장을 그대로 옮기면 이렇다. 최상위권 모델 몇몇은 **컨텍스트에 100토큰밖에 없는 상태에서도 실패했고, 대부분은 1000토큰 지점에서 이미 정확도가 심하게 무너졌다.** 모든 모델이 자기가

광고한 최대 컨텍스트 윈도우에 한참 못 미쳤으며, 그 격차가 99%를 넘는 경우도 있었다.

의외의 결과도 하나 붙어 있다. 요약 과제 — 그저 모든 항목의 값을 더하라고 시킨 것 — 에서 모든 모델이 여러 바늘을 찾는 과제보다 못했다. 저자들도 예상하지 못했다고 적는다.

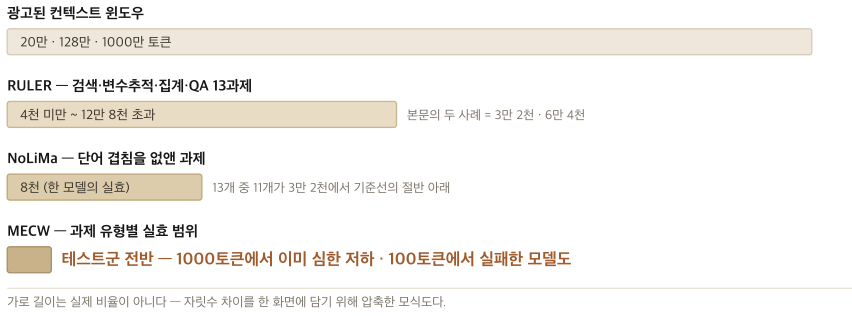


그림 14 — 세 층의 숫자. 아래로 내려갈수록 과제가 "찾기"에서 "관계 유지"로 옮겨가고, 쓸 수 있는 길이는 자릿수 단위로 줄어든다.

이 숫자들이 이 책에 중요하다.

아키텍처 판단은 검색이 아니다. 이 함수를 고치면 어디가 영향받는지, 이 변경이 저 불변식을 깨는지, 세 모듈의 상태가 동시에 일관적인지를 따지는 일이다. **관계를 유지하며 처리하는 쪽**이다.

13장에서 사람의 숫자가 그랬다. 가만히 들고 있는 것은 일곱 개, 조작해야 하면 네 개.

아직 측정되지 않은 것

한 가지 공백을 명시해야 한다.

2026년 중반에 나온 최신 플래그십 모델들에 대해서는 RULER의 표준 점수가 아직 발표되지 않았다. 1000만 토큰을 광고하는 모델들의 실효 범위에 대해서도 공개된 측정이 없다. (100만급은 측정이 있다 — 그리고 그 결과가 위에 적힌 6만 4천·1만 6천·4천 미만이다.)

그러므로 이 책은 그 숫자들을 품질 보증이 아니라 **상한선**으로 취급한다.

대가

이 장이 만드는 결론에는 유보가 붙는다.

실효 범위가 광고보다 작다는 것은, 지금 시점의 측정이다. 측정 방식이 바뀌면 숫자가 바뀐다. 모델이 바뀌면 확실히 바뀐다.

그리고 이 장의 증거들은 전부 **길이**에 대한 것이다. 길이가 문제라면 짧게 만들면 된다. 필요한 것만 골라서 넣으면 된다.

그것이 지금 모든 코딩 에이전트가 하고 있는 일이다. 전체를 넣지 않는다. 검색해서 필요한 조각만 가져온다.

그렇다면 이 장의 숫자들은 이미 해결된 문제에 대한 것 아닌가.

다음 장이 그 질문에 답한다.

주

1 Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, Boris Ginsburg, "RULER: What's the Real Context Size of Your Long-Context Language Models?", arXiv:2404.06654, COLM 2024. 〈https://arxiv.org/abs/2404.06654〉 · 코드: 〈https://github.com/NVIDIA/RULER〉 — 본문의 두 데이터포인트는 Yi-34B(광고 200K, 실효 32K)와 GPT-4-1106-preview(광고 128K, 실효 64K)다. ↩

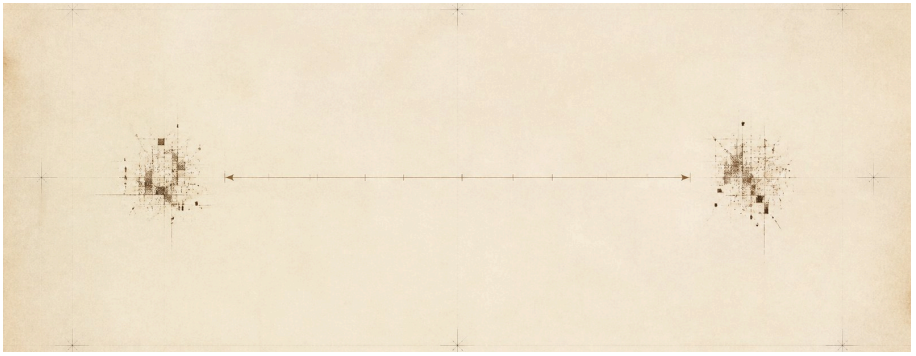
2 RULER 논문에는 "실효 컨텍스트는 광고의 몇 퍼센트"라는 형태의 일반화가 없다. 논문 자신의 요약은 정성적이다 — "테스트한 모델 중 절반만이 32K 길이에서 만족스러운 성능을 유지한다." 본문이 든 100만급 세 모델은 GLM4(실효 64K)·GradientAI/Llama3(16K)·LWM(4K 미만)이다. 더 극단적인 비율이 도는 것은 논문 이후 모델까지 포함한 깃허브 리더보드 값이며, 논문 표와 섞어 쓰면 안 된다. 2차 자료에서 흔히 보이는 "50~65%"류 수치는 산출 방식이 확인되지 않는 요약이며, 논문의 주장이 아니다. ↩

3 Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Trung Bui, Ryan A. Rossi, Seunghyun Yoon, Hinrich Schütze, "NoLiMa: Long-Context Evaluation Beyond Literal Matching", arXiv:2502.05167, ICML 2025 (PMLR 267:44554–44570). 〈https://arxiv.org/abs/2502.05167〉 — 본문의 개별 수치는 GPT-4o(기준 99.3% → 32K에서 69.7%, 광고 128K, 실효 8K)다. **잣대 주의:** NoLiMa의 실효 길이 임계선은 RULER의 Llama-2-7B@4K가 아니라 **각 모델 자신의 base 점수 × 0.85**다. 두 벤치마크의 "실효 길이"를 같은 자로 잰 값처럼 나란히 놓으면 안 된다. **버전 주의:** arXiv v1(2025-02)은 "12개 중 10개"로 적혀 있고 깃허브 README도 아직 그 수치다. 본문은 v2/v3 및 ICML 판본의 "13개 중 11개"를 따른다. ↩

4 Kelly Hong, Anton Troynikov, Jeff Huber, "Context Rot: How Increasing Input Tokens Impacts LLM Performance", Chroma 기술 리포트, 2025-07-14. 〈https://www.trychroma.com/research/context-rot〉 · 재현 코드: 〈https://github.com/chroma-core/antext-rot〉 — 원문: "we find that structural coherence consistently hurts model performance" / "Shuffling the haystack and removing local coherence consistently improves performance." 저자들은 어텐션 메커니즘과의 관계를 가능성으로만 언급하고 메커니즘 규명은 범위 밖이라고 명시한다. ↩

5 Norman Paulsen, "Context Is What You Need: The Maximum Effective Context Window for Real World Limits of LLMs", arXiv:2509.21361, 2025-09-21. 〈https://arxiv.org/abs/2509.21361〉 · 저널판: *Advances in Artificial Intelligence and Machine Learning* 6(1), 2026-01, pp. 4853–4878. 모델 11종, 과제 4종. 과제별 MECW 값은 논문에 표가 아니라 그래프로만 실려 있으므로, 특정 과제의 정확한 토큰 수를 이 논문에서 ↩

인용하는 것은 그래프 독해다 — 이 책은 논문이 산문으로 명시한 수치("as little as 100 tokens", "severe degradation ... by 1000 tokens", ">99%")만 쓴다.



17장

공백 2만 5천 개

통념

긴 컨텍스트에서 모델이 실패하는 이유에 대해 널리 받아들여진 설명이 있었다.

검색 실패다.

입력이 길면 관련 있는 정보를 못 찾는다. 관련 없는 것들 사이에 묻힌다. 그러니 검색을 개선하면 된다. 필요한 것만 정확히 골라서 넣어주면, 모델은 짧은 입력을 받은 것과 똑같이 잘할 것이다.

이 가정 위에 산업 하나가 섰다. 검색 증강 생성(RAG), 벡터 데이터베이스, 리랭커, 청킹 전략. 그리고 코딩 에이전트의 파일 탐색 도구들.

RAG (Retrieval-Augmented Generation, 검색 증강 생성)

질문이 들어오면 먼저 외부 문서 저장소에서 관련 조각을 찾아온 뒤, 그 조각들을 프롬프트에 붙여 모델에게 답하게 하는 방식. 2020년 논문에서 이름이 붙었고[1] 이후 널리 채택됐다. 딸린 부품들이 있다 — 문서를 조각내는 **청킹**(chunking), 의미가 비슷한 조각을 찾아주는 **벡터 데이터베이스**, 찾아온 후보를 다시 줄 세우는 **리랭커**(reranker). 이 산업 전체가 하나의 전제 위에 서 있다: **검색만 잘하면 된다.**

전제는 하나였다. **검색이 완벽하면 문제가 사라진다.**

실험

2025년의 한 논문이 그 전제를 시험한다.[2]

제목이 결론이다. 「검색이 완벽해도 컨텍스트 길이만으로 LLM 성능이 저하된다」 (Context Length Alone Hurts LLM Performance Despite Perfect Retrieval).

방식은 이렇다. 수학과 질의응답과 코딩 과제를 준다. 다섯 개의 모델을 쓴다. 오픈소스와 상용을 섞는다. 그리고 입력 길이를 늘려간다. 광고된 컨텍스트 한도 안에서만 늘린다.

중요한 조작이 여기 있다. **모델이 관련 정보를 전부 정확하게 찾아냈는지를 따로 확인한다.** 조건들과 질문을 전부 제대로 추출했는지 검사한다.

결과

검색이 완벽한 경우에도 성능이 떨어진다.

13.9%에서 85%.

과제와 모델에 따라 폭이 넓지만, 방향은 일정하다. 관련 정보를 다 찾았는데도 답이 틀린다.

세 번의 절제

이 논문이 강한 이유는 결과가 아니라 이어지는 실험들에 있다. 가능한 설명을 하나씩 제거한다.

> **절제 실험 (ablation study)**
>
> 어떤 결과의 원인을 좁히기 위해, 후보 원인을 하나씩 **제거하고도 결과가 남는지** 보는 실험 설계. 원인 A를 없앴는데 결과가 그대로면 A는 원인이 아니다. 이 논문이 강한 이유가 여기 있다 — 결과를 보여주는 데서 그치지 않고, "그건 사실 방해 요소 탓 아니냐" "위치 탓 아니냐" 같은 반론을 하나씩 실험으로 잘라냈다.

첫 번째. 방해 요소가 문제인가. 긴 입력에는 관련 없는 내용이 많이 들어 있고, 그것이 모델을 헷갈리게 하는 것 아닌가.

그래서 관련 없는 토큰을 **공백으로 바꾼다.**

논문의 첫 그림이 이 실험이다. 공백 2만 5천 개를 끼워 넣는다. 모델은 조건들과 질문을 전부 정확히 추출한다. 그리고 틀린 결과에 도달한다.[3]

공백에는 의미가 없다. 헷갈릴 내용이 없다. 그런데도 틀린다.

두 번째. 그래도 공백을 처리하느라 뭔가 소모되는 것 아닌가.

그래서 관련 없는 토큰을 **전부 마스킹한다.** 모델이 관련 있는 토큰만 보도록 강제한다.

여전히 틀린다. 3만 토큰의 방해 요소를 전부 가린 상태에서도 최소 7.9%가 떨어졌다.

세 번째. 그럼 위치 문제인가. 관련 정보가 앞이나 뒤에 있으면 잘 보고 가운데 있으면 놓친다는 현상이 알려져 있다.

> **가운데서 길을 잃다 (Lost in the Middle)**
>
> 2023년에 발표되고 2024년 TACL에 실린 논문이 보고한 현상. 긴 입력에서 정답 근거가 **맨 앞이나 맨 뒤**에 있으면 잘 찾고 **가운데**에 있으면 놓친다 — 성능을 위치에 따라 그리면 U자 곡선이 나온다. 저자들은 그 두 끝을 앞부분 편향(primacy)과 뒷부분 편향(recency)이라 불렀다 — 곡선을 기술하는 이름이지 원인의 규명은 아니다. 다만 효과는 모델마다 균일하지 않고, 이후 반증 연구도 나왔다.[4] 이 장의 논문이 세 번째로 제거하는 것이 바로 이 설명이다.

그래서 모든 근거를 **질문 바로 앞에 놓는다.** 위치 문제가 생길 수 없는 배치다.

같은 저하가 나온다. 3만 토큰의 공백 조건에서 최대 17%(Mistral)와 최대 20%(Llama)까지 떨어졌다.

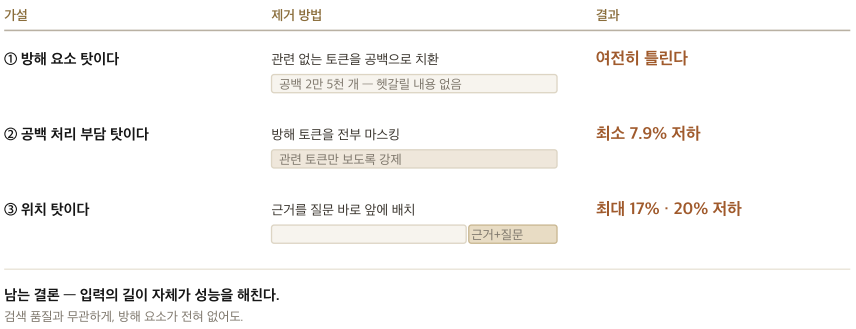


그림 15 — 세 번의 절제. 가능한 설명을 하나씩 제거해도 저하가 남는다.

남는 문장

세 가지를 제거하고 나면 결론이 하나 남는다.

입력의 길이 자체가 성능을 해친다. 검색 품질과 무관하게, 방해 요소가 전혀 없어도.

논문이 제안한 완화책은 이 결론과 일관적이다. 긴 컨텍스트 과제를 짧은 컨텍스트 과제로 바꾸는 것이다. 모델에게 문제를 풀기 전에 찾아낸 근거를 먼저 **다시 적게** 한다. 그러면 실제 추론이 일어나는 시점의 입력이 짧아진다. 논문은 이것을 "찾고 나서 풀기"(Retrieve then Solve)라고 부른다.

효과는 있었다. RULER에서 일관된 개선이 나왔다. 다만 개선폭은 이미 강한 기준선 위에서 최대 4% 정도다.

이 장이 하는 일

이 책의 3부는 하나의 전제 위에 서 있다. 기계가 한 번에 감당할 수 있는 양이 유한하다는 것.

그 전제에 대한 가장 강한 반론이 있다. 16장 끝에서 나온 그것이다.

전체를 넣지 않으면 되지 않는가. 필요한 것만 골라 넣으면 되지 않는가. 인간도 코드베이스를 통째로 머리에 담은 적이 없다. 우리가 만든 것은 더 작은 모듈이 아니라 grep과 언어 서버와 콜그래프 같은 **탐색 도구**였다. 에이전트도 같은 길을 갈 것이다.

이 반론은 컨텍스트 윈도우가 커지지 않아도 성립한다. 그래서 강하다.

이 장의 논문이 그 반론에 대한 답이다.

탐색이 완벽해져도 남는 부하가 있다. 검색을 100% 성공시키고, 방해 요소를 전부 제거하고, 위치까지 최적으로 배치해도, 길이가 길면 틀린다.

완전히 죽지는 않는다

정직하게 남길 것이 있다.

이 논문이 보여준 것은 탐색이 소용없다는 게 아니다. 탐색은 실제로 부하를 줄인다. 짧게 만드는 것은 도움이 된다. 논문 자신의 완화책이 그것이다.

이 논문이 보여준 것은 **탐색으로 환원되지 않는 부하가 있다**는 것이다. 아무리 잘 골라도, 골라낸 것이 많으면 그 자체로 값을 치른다.

그러니 반론은 죽지 않는다. 약해진다.

주장을 정확히 다시 쓰면 이렇게 된다. 탐색이 소용없다는 것이 아니라, **탐색해도 한 번에 판단해야 하는 순간의 부하는 남는다**는 것.

왜 이런 일이 일어나는가

이 논문은 현상을 보여주고 메커니즘을 확정하지 않는다.

어텐션(attention)이 길이에 따라 흐려진다는 설명이 있고, 위치 인코딩(positional encoding)의 성질 때문이라는 설명이 있고, 학습 데이터의 길이 분포 때문이라는 설명이 있다.

> **어텐션 (attention)**
>
> 트랜스포머(Transformer) 모델의 핵심 연산. 어떤 토큰을 처리할 때 입력 안의 다른 모든 토큰을 얼마나 참조할지 가중치를 매긴다. 그 가중치의 총합은 정해져 있으므로, **입력이 길어질수록 각 토큰이 받는 몫이 평균적으로 얇아진다.** "길이가 길면 흐려진다"는 직관적 설명이 여기서 나온다. 다만 이것이 실제 저하의 원인인지는 이 장의 논문이 확정하지 않는다.

이 책은 그 중 하나를 고르지 않는다. 고를 근거가 없다.

다만 한 가지는 말할 수 있다. **원인이 무엇이든, 그것은 창을 넓히는 것으로 사라지지 않았다.** 광고된 한도 안에서 일어난 일이기 때문이다.

남은 질문

이것이 지금 아키텍처에 무엇을 요구하는가.

벤치마크의 숫자와 실제 코드베이스는 다르다. 다섯 개 모델과 네 종류 과제에서 나온 결과가, 수천 개 파일이 얽힌 저장소에서 어떻게 나타나는가.

다음 장이 그 자리에서 측정된 숫자다.

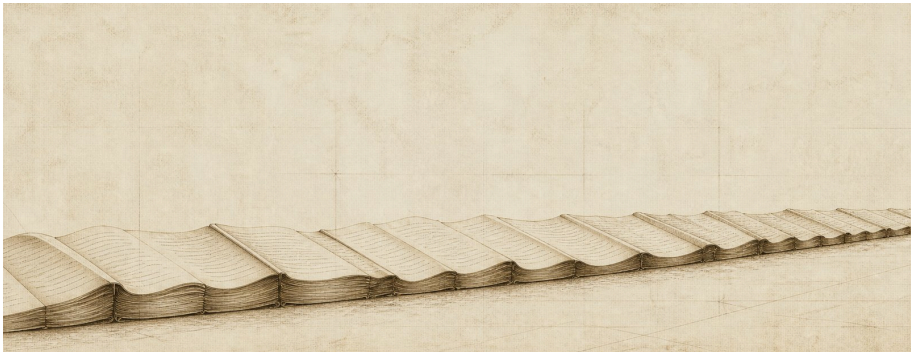
주

1 Patrick Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", *NeurIPS 2020*. arXiv:2005.11401 〈https://arxiv.org/abs/2005.11401〉 ↩

2 Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A. Huerta, Hao Peng, "Context Length Alone Hurts LLM Performance Despite Perfect Retrieval", arXiv:2510.05381, Findings of EMNLP 2025. 〈https://arxiv.org/abs/2510.05381〉 — 모델 5종(Llama-3.1-8B-Instruct, Mistral-v0.3-7B-Instruct, GPT-4o, Claude-3.7-Sonnet, Gemini-2.0), 과제 4종(VarSum, GSM8K, MMLU, HumanEval) 및 완화책 평가용 RULER QA. ↩

3 정확히는 2만 5천이라는 수는 논문 **Figure 1의 예시 설정**이다. 본문의 스윕은 0·7,500·15,000·3만 토큰이고, 부록의 세밀 스윕이 3,750토큰 간격으로 3만까지 간다. 이 장이 그 그림을 제목으로 삼은 것은 그것이 이 논문의 논증을 한 장면으로 압축하기 때문이지, 2만 5천이 실험의 대표값이어서가 아니다. ↩

4 Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, Percy Liang, "Lost in the Middle: How Language Models Use Long Contexts", *TACL* 12, 2024, pp. 157–173. 〈https://doi.org/10.1162/tacl_a_00638〉 · 반증 시도: Mingyang Song, Mao Zheng, Xuan Luo, "Counting-Stars: A Multi-evidence, Position-aware, and Scalable Benchmark for Evaluating Long-Context Large Language Models", arXiv:2403.11802, COLING 2025. §4.2(절 제목이 "Lost in the Middle"이다) — "our findings can not strongly corroborate the lost-in-the-middle phenomenon." 이 책은 이 현상을 수치와 함께 인용하지 않는다. 흔히 도는 "20~30 퍼센트포인트 하락"과 "6개 모델 계열"은 둘 다 2차 요약의 각색이다. ↩



18장

16만 줄 앞에서

실제 저장소

벤치마크의 숫자와 실제 코드베이스는 다르다.

앞 두 장의 실험들은 통제된 환경에서 나왔다. 심어놓은 사실, 정해진 방해 요소, 측정 가능한 정답. 실제 저장소는 그렇게 생기지 않았다.

그래서 실제 저장소에서 측정한 결과가 필요하다.

과제

저장소 현대화를 측정하는 벤치마크가 있다. 이름은 RepoMod-Bench다.[1]

방식이 이렇다. 실제 오픈소스 프로젝트를 가져온다. 에이전트에게 낡은 부분을 현대적인 형태로 바꾸라고 시킨다. 그리고 테스트를 돌린다.

저장소 현대화 (repository modernization)

낡은 코드베이스를 현대적인 형태로 옮기는 작업. 이 벤치마크의 과제는 대개 **언어 이식**이다 — C++로 쓰인 21만 줄짜리 계산기를 Go로, 16만 줄짜리 코드 정형기를 러스트로 옮기는 식이다. 한 파일씩 기계적으로 번역해서는 안 되고, 원본이 무엇을 하는 프로그램인지를 이해한 뒤 대상 언어의 관용에 맞게 다시 지어야 한다. **한 함수의 정답이 아니라 시스템 전체의 일관성을 요구하는 과제**라는 점이 이 장에 중요하다.

이 벤치마크가 구현 무관 테스트(implementation-agnostic testing)를 쓴다는 점이 중요하다. 에이전트가 어떤 방식으로 고쳤는지를 채점하지 않는다. 고친 결과가 동작하는지만 본다. 정답이 하나가 아니어도 되는 채점 방식이다.

규모는 이렇다. 21개 저장소, 8개 언어, 합계 160만 줄, 테스트 11,616개. 가장 작은 프로젝트가 14줄, 가장 큰 것이 21만 1천 줄이다. 에이전트는 네 종류를 붙였다.

숫자

규모별로 잘라 보면 곡선이 하나 나온다.

1만 줄 미만에서 평균 통과율 91.3%. 1만에서 5만 줄 사이가 66.9%. 5만 줄 이상이 15.3%.

그리고 개별 숫자들.

10만 줄이 넘는 프로젝트는 대부분 평균 통과율이 20% 아래다.

16만 2천 줄짜리 프로젝트 하나에서는 네 개의 에이전트가 전부 0%를 기록한다.

가장 큰 프로젝트는 21만 1천 줄이었고, 거기서 가장 잘한 에이전트가 19.5%를 냈다.

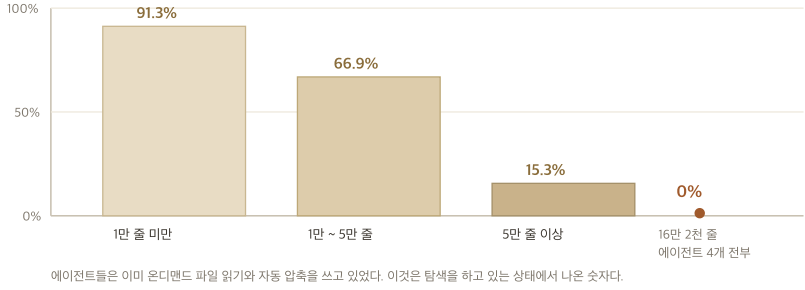


그림 16 — 규모에 따른 붕괴. 완만한 저하가 아니라 계단이다.

저자들이 붙인 문장

숫자보다 중요한 것이 그 옆에 붙은 해석이다.

논문의 저자들은 이것이 컨텍스트 윈도우 문제가 **아니라고** 적는다.

근거는 에이전트들의 능력에 있다. 현대적인 에이전트는 파일을 필요할 때 골라 읽을 수 있고, 컨텍스트가 차면 자동으로 압축한다. 임의로 큰 코드베이스를 다룰 수 있다.

그런데도 저하가 지속된다.

저자들이 지목한 어려움은 다른 곳에 있다. **수천 개의 서로 의존하는 파일에 걸쳐 일관된 아키텍처 이해를 유지하는 것.** 그리고 그것은 원시 컨텍스트

윈도우 병목과는 질적으로 다른 문제라고 적는다.[2]

> **자동 압축 (context compaction)**
>
> 에이전트의 대화 맥락이 컨텍스트 한도에 가까워지면, 지금까지의 대화를 요약해 짧게 줄이고 그 요약으로 계속 진행하는 기법. 클로드 코드를 비롯한 주요 에이전트가 기본으로 쓴다. 덕분에 에이전트는 **한도보다 훨씬 큰 작업을 수행할 수 있다** — 이 장의 결과가 컨텍스트 윈도우 크기로 설명되지 않는 이유이자, 다음 절에서 압축 자체가 성능을 깎는다는 결과가 나오는 이유이기도 하다.

용어

이 문장이 이 책의 용어 선택을 정당화한다.

이 책은 컨텍스트 윈도우라고 쓰지 않는다. **인지부하**라고 쓴다.

컨텍스트 윈도우는 용량이다. 벤더가 파는 숫자이고, 매년 커진다. 그 숫자를 근거로 아키텍처를 논하면 반론이 즉시 온다. 커지고 있잖아.

인지부하는 다르다. 한 번에 관계를 유지하며 판단할 수 있는 양이다. 13장의 코완이 저장과 조작을 나눈 것과 같은 구분이다. 파일을 읽을 수 있다는 것과 그 파일들 사이의 관계를 동시에 붙들 수 있다는 것은 다른 능력이다.

10만 줄짜리 저장소에서 에이전트가 실패하는 것은 파일을 못 읽어서가 아니다. 읽은 것들 사이의 관계를 유지하지 못해서다.

이어지는 작업

다른 종류의 측정도 있다. ChainSWE라는 벤치마크다.[3]

같은 코드베이스에서 **순차적으로 의존하는** 버그 수정을 시킨다. 첫 번째 버그를 고치고, 그 위에서 두 번째를 고치고, 다시 그 위에서 세 번째를 고친다. 54개 파이썬 프로젝트의 이슈 304건에서 체인 100개를 뽑았고, 체인의 평균 길이는 3.04다.

체인이 깊어질수록 버그당 해결률이 떨어진다. 논문의 초록은 최대 70% 하락이라고 적는다.

여기서 정밀하게 읽어야 한다. **70%는 상대 하락률이고, 그것도 완화책을 쓴 구성의 숫자다.** 저장소를 매번 초기화한 기준 조건 대비, 체인 3번째 위치에서 아무 완화책도 안 쓴 구성은 59% 떨어졌고, 요약이나 위임을 쓴 구성이 70% 떨어졌다.

절대값이 더 선명하다. 체인 3번째 위치의 버그 해결률은 이렇다.

원본 이력 유지 27.7% · 컨텍스트 요약 20.6% · 편집 위임 17.5%.

세 가지를 정확히 해야 한다.

첫째, 기준 구성은 "아무 장치도 안 쓴 것"이 아니라 **이전 수정의 원본 이력을 그대로 들고 가는 것**이다. 둘째, 여기서 "위임"은 역할을 나눈 멀티에이전트가 아니라 **파일 조회와 편집을 자연어로 작은 편집 모델에 넘기는 컨텍스트 오프로딩**이다. 셋째, 하류 실패의 48%가 체인 자체에서 온 오류라, 초기화 조건과의 격차를 순수한 전달 비용으로 읽으면 안 된다.

그리고 위임 구성은 부하가 쌓여서 나빠진 것이 아니다. 저장소를 매번 초기화한 기준 조건에서부터 이미 낮았고(49.5 대 58.6), 체인이 깊어지면서 더 낮아졌다. **전 구간에서 낮고, 부하가 쌓이면 더 낮아진다**가 정확한 서술이다.

그리고 이 책이 주목하는 부분이 그 순서다. 부하를 줄이려고 넣은 장치 두 개가 기준선보다 나빴다.

컨텍스트를 요약해서 넘겼다. **성능이 떨어졌다.**

편집을 다른 모델에 넘겼다. **성능이 더 떨어졌다.**

둘 다 부하를 줄이려고 넣은 장치인데, 원본을 그대로 들고 가는 것보다 나빴다.

> **서브에이전트 (sub-agent)**
>
> 주 에이전트가 하위 작업을 떼어 별도의 에이전트에게 맡기고 결과만 돌려받는 구조. 주 에이전트의 컨텍스트에 하위 작업의 세부가 쌓이지 않으므로 부하가 줄 것이라는 기대가 있었다. 이 벤치마크에서는 **가장 나쁜 구성**으로 나왔다. 논문의 해석은 정보 손실과 불일치다 — 위임받은 쪽이 왜 그렇게 고쳐야 하는지를 모르고, 돌려준 결과가 주 에이전트가 들고 있던 그림과 어긋난다.

쪼개기만으로는 안 된다

이 결과가 이 책의 논증을 정밀하게 만든다.

3부의 주장은 쪼개야 한다는 것이다. 그런데 위 실험은 쪼개는 흔한 방법 두 가지가 오히려 나빴다고 말한다.

모순처럼 보이지만 아니다. 요약과 위임은 **작업을 나눈 것**이지 **시스템을 나눈 것**이 아니다.

같은 얽힌 코드베이스를 그대로 두고 작업만 조각내면, 각 조각이 여전히 바깥을 알아야 한다. 요약은 그 바깥을 압축해서 넘기려는 시도이고, 압축

과정에서 필요한 것이 빠진다. 위임도 마찬가지다. 받는 쪽이 맥락을 모른다.

부하를 줄이는 것은 **작업의 크기가 아니라 판단의 완결성**이다.

이 구분이 다음 장의 주제다.

반론이 다시 온다

여기서 17장의 반론을 다시 꺼내야 한다. 이번에는 더 강한 형태로.

에이전트는 전체를 담지 않는다. grep을 하고, 인덱스를 만들고, 서브에이전트를 던진다. 이해가 **상주**가 아니라 **탐색**으로 달성된다면, 한 번에 담을 수 있는 양은 아키텍처를 강제하는 힘이 아니다.

인간이 그랬다. 우리는 코드베이스를 통째로 외운 적이 없다. 우리가 만든 것은 더 작은 모듈만이 아니라 그것을 뒤지는 도구들이었다. 태그 파일(ctags), 언어 서버(LSP), 호출 계층 뷰, 전문 검색.

기계도 같은 길을 갈 것이다. 그리고 그 길을 가는 데는 컨텍스트 윈도우가 커질 필요조차 없다.

답의 세 겹

이 반론에 대한 답은 앞의 세 장에 나눠 놓여 있다.

첫째, 17장의 논문이다. 검색이 100% 완벽하고, 방해 요소가 공백으로 치환되고, 위치까지 최적일 때도 길이 자체가 성능을 깎았다. 탐색이 완벽해져도 남는 부하가 있다.

둘째, 이 장의 벤치마크다. 저하가 일어난 에이전트들은 **이미 온디맨드 파일 읽기와 자동 압축을 쓰고 있었다.** 탐색을 하고 있는 상태에서 나온 0%다. 그리고 저자들이 직접 컨텍스트 윈도우 문제가 아니라고 못 박았다.

셋째, 이 장 앞부분의 요약과 위임 실험이다. 탐색의 자연스러운 확장인 두 방법이 오히려 나빴다.

그래도 죽지 않는다

이 반론은 완전히 죽지 않는다. 그 점을 이 책은 숨기지 않는다.

탐색은 실제로 부하를 줄인다. 좋은 인덱스와 좋은 검색은 에이전트를 확실히 낫게 만든다. 그리고 이 분야는 빠르게 움직인다.

그래서 주장은 이렇게 좁혀진다.

탐색이 소용없다는 것이 아니다. 탐색해도 한 번에 판단해야 하는 순간이 있고, 그 순간의 부하는 남는다.

그 순간이란 이런 것이다. 이 변경이 저 불변식을 깨는가. 세 모듈의 상태가 동시에 일관적인가. 이 함수의 계약을 바꾸면 어디까지 파급되는가.

검색으로 답할 수 없는 질문들이다.

반대 방향의 증거

공정하게 놓을 것이 하나 더 있다.

모든 측정이 규모에 따른 단조 저하를 보이는 것은 아니다. 한 연구는 몇몇 모델이 쉬운 시나리오보다 전문가 수준 시나리오에서 오히려 더 잘했다고

보고한다.[4]

저자들의 가설이 흥미롭다. 50만에서 100만 토큰 규모의 전문가 수준 프로젝트들은 **명시적인 아키텍처 문서와 더 명확한 모듈 경계**를 가지고 있고, 거대한 코드베이스 앞에서 에이전트가 더 체계적인 탐색 전략을 택한다는 것이다.

저자들 자신이 이 설명을 가설로 표시했다는 점은 그대로 옮겨야 한다. "그럴 것이다", "그럴 수 있다"는 어투이고 검증된 바 없다. 그리고 더 큰 유보가 하나 있다. 그 벤치마크의 코드베이스는 생성 파이프라인에서 나온 것이라, 모듈 경계가 깨끗한 것이 실제 대형 시스템의 성질이 아니라 **합성 과정의 부산물**일 수 있다.

그래서 이 자료는 이 책의 주장을 지지하는 증거로 쓰지 않는다. 같은 방향을 가리키는 정황으로만 놓는다. **경계가 명확하면 크기가 커도 다룰 수 있다** — 이 문장은 여기서 확인된 것이 아니라 아직 열려 있다.

다만 이 자료가 확실히 하는 것이 하나 있다. 규모와 성능의 관계가 **단조롭지 않다**는 것. 줄 수만으로는 설명되지 않는 무엇인가가 개입한다. 이 책은 그것을 얽힘이라고 부르지만, 그렇게 부르는 것과 그것을 보인 것은 다르다.

남은 질문

10만 줄에서 무너진다면, 그 아래는 안전한가.

숫자 하나를 기준선으로 삼고 싶은 유혹이 있다. 그런데 16장의 마지막 연구가 그것을 막는다. 실효 범위는 과제에 따라 달라지고, 걸러서 정렬하고

합계 내는 정도의 관계 처리 과제에서는 그 범위가 수천 토큰이 아니라 **1000토큰 언저리**로 내려간다.

기준은 줄 수가 아니다.

그렇다면 무엇인가.

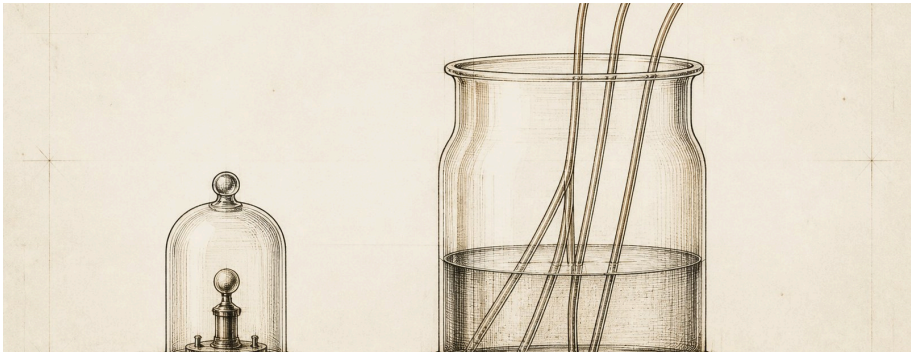
주

1 Xuefeng Li, Nir Ben-Israel, Yotam Raz, Belal Ahmed, Doron Serebro, Antoine Raux, "RepoMod-Bench: A Benchmark for Code Repository Modernization via Implementation-Agnostic Testing", arXiv:2602.22518, 2026-02-26. 〈https://arxiv.org/abs/2602.22518〉 · 벤치마크: 〈https://github.com/Modelcode-ai/mcode-benchmark〉 — 0%를 기록한 것은 uncrustify(C++→러스트, 16만 2천 줄, 테스트 2,024개), 19.5%가 최고였던 것은 qalculate(C++→Go, 21만 1천 줄, 테스트 564개)다. 에이전트 4종의 전체 평균은 클로드 코드(Opus 4.5) 48.2 · 코덱스 CLI(GPT-5.2) 30.4 · 오픈코드+클로드 42.0 · 오픈코드+GPT-5.2 43.0. 논문의 공식 규모 구간은 1만 미만 / 1만–5만 / 5만 이상이며, "10만 줄"은 큰 구간을 설명하는 산문 속 문장이지 별도 구간이 아니다. 2026년 8월 현재 arXiv 프리프린트이며 학회 게재는 확인되지 않았다. ↩

2 원문: "This degradation persists despite the fact that modern coding agents can read files on demand and manage arbitrarily large codebases through automatic context compaction. The challenge therefore lies not in fitting code into context, but in maintaining coherent architectural understanding across thousands of interdependent files — a qualitatively different problem from the context-window bottleneck of raw LLMs." ↩

3 Qirui Jin 외 15인, "ChainSWE: Benchmarking Coding Agents on Multi-Bug Software Maintenance", arXiv:2607.02606, 2026-07-01. 〈https://arxiv.org/abs/2607.02606〉 — RepoMod-Bench와 마찬가지로 2026년 8월 현재 학회 게재 여부는 확인되지 않았다. — 원문: "summarizing the context or delegating file edits to a sub-agent consistently degrades performance." 측정 주의사항: 논문 Table 3의 캡션에 한 모델(Claude-Opus)의 위치별 행이 실측이 아니라 추정치라고 명시되어 있고, 59%/70% 비교는 그 평균 위에서 계산된다. Oracle(매번 초기화) 조건의 값은 Baseline 58.6 / Summarize 64.3 / Sub-Agent 49.5로, 위임 구성은 부하가 없는 지점에서도 이미 기준보다 낮다. ↩

4 Jielin Qiu et al. (Salesforce AI Research), "LoCoBench-Agent", arXiv:2511.13998, 2025-11-17, § 5.4. 〈https://arxiv.org/abs/2511.13998〉 — 원문은 "several models actually improve from Easy to Expert scenarios"라 적고, 그 이유를 전문가 티어 프로젝트가 "more explicit architectural documentation, clearer module boundaries, and comprehensive README files"를 갖기 때문일 것이라고 **추측한다**("likely stems", "may employ"). 검증된 설명이 아니다. 또한 이 벤치마크의 코드베이스는 LoCoBench의 생성 파이프라인 산출물이므로, 경계의 명확성이 합성의 부산물일 가능성이 논문 자체에서 해소되지 않는다. ↩



19장

작은 것이 아니라 닫힌 것

잘못된 기준

앞 장이 남긴 질문은 이것이었다. 기준이 줄 수가 아니라면 무엇인가.

줄 수를 기준으로 삼으면 곧바로 막힌다. 500줄짜리 파일이 스무 곳을 알아야 이해되는 경우가 있고, 5천 줄짜리 파일이 안에서 완결되는 경우가 있다.

둘 중 어느 쪽이 에이전트에게 어려운가.

닫힌 계

이 책이 쓰는 어휘는 이것이다.

닫힌 계란, 그 안에서 판단이 완결되는 단위다.

닫힌 계 (closed system) — 이 책의 용법

물리학에서 닫힌 계는 물질이 드나들지 않는 계를 뜻하지만, 이 책은 그 말을 **인식론적으로** 쓴다: 어떤 단위가 옳은지를 말하기 위해 **그 단위 바깥을 볼 필요가 없으면** 그것은 닫혀 있다. 완전히 고립되어 있다는 뜻이 아니다 — 입출력은 있다. 다만 그 입출력이 계약으로 명시되어 있어서, 안을 판단할 때 바깥의 구현을 들여다볼 필요가 없다는 뜻이다. 반대말은 "작은 것"이 아니라 **"열린 것"**이다.

바깥을 몰라도 안이 옳은지 말할 수 있으면 닫힌 계다. 이 함수가 맞게 동작하는지 확인하려고 다른 모듈의 구현을 열어봐야 한다면, 그것은 닫혀 있지 않다.

작음과 닫힘은 다른 축이다. 작음은 크기의 문제고, 닫힘은 **완결성**의 문제다.

그림 17 — 크기와 닫힘은 직교한다. 에이전트에게 어려운 것은 왼쪽 위가 아니라 오른쪽 아래다.

왜 완결성인가

13장에서 코완의 구분이 나왔다. 저장할 수 있는 양과 조작할 수 있는 양이 다르다. 가만히 들고 있는 것은 일곱 개, 관계를 유지하며 처리해야 하면 서너 개.

16장에서 같은 구분이 기계 쪽에서 나왔다. 단순 검색은 수천 토큰 규모에서 되는 반면, 걸러서 정렬하고 합계를 내는 정도의 관계 처리 과제에서는 최상위 모델도 1000토큰 언저리에서 이미 심하게 무너졌다.

부하를 만드는 것은 항목의 개수가 아니라 **동시에 유지해야 하는 관계의 수**다.

닫힌 계는 그 관계의 수를 자른다. 안에서 완결된다는 것은, 안을 판단할 때 바깥과의 관계를 들고 있지 않아도 된다는 뜻이다.

500줄이 스무 곳과 얽혀 있으면 관계가 스무 개다. 5천 줄이 안에서 끝나면 관계가 0개다.

파나스가 말한 것

2장의 정보 은닉이 여기서 다시 온다.

파나스가 감추라고 한 것은 바깥에서 그 결정이 존재한다는 것조차 모르게 하는 것이었다. 모른다는 것은 관계를 들고 있지 않아도 된다는 뜻이다.

다만 그가 감출 것을 고른 기준은 달랐다. **바뀔 것 같은 것**을 감췄다. 사람이 무엇을 고칠지 안다는 전제 위에서.

지금 감춰야 할 기준은 다르다. **판단할 때 몰라도 되는 것**을 감춘다. 무엇이 바뀔지가 아니라, 무엇을 안 봐도 되는지가 기준이다.

두 기준은 자주 같은 선을 그린다. 그런데 항상은 아니다. 자주 바뀌지 않지만 이해하려면 반드시 알아야 하는 것들이 있다. 전역 설정, 공유 상수, 암묵적

순서 의존. 파나스의 기준으로는 감출 이유가 없고, 이 책의 기준으로는 감춰야 한다.

쪼개기만으로는 안 된다

18장에서 나온 반증 자료를 여기 놓아야 한다.

컨텍스트를 요약해서 넘기면 성능이 떨어졌다. 서브에이전트에게 위임하면 더 떨어졌다.

이것은 쪼개기가 나쁘다는 뜻이 아니다. 그 두 방법이 시스템을 쪼갠 것이 아니라 **작업만 쪼갠 것**이라는 뜻이다.

얽힌 코드베이스를 그대로 두고 작업을 나누면, 나눠진 각 조각이 여전히 바깥을 알아야 한다. 요약은 그 바깥을 압축해서 전달하려는 시도다. 압축은 손실이고, 무엇이 손실되었는지는 받는 쪽이 모른다.

작업을 나누는 것은 쉽고, 시스템을 나누는 것은 어렵다. 그래서 사람들은 쉬운 쪽을 먼저 시도한다. 그리고 그것은 도움이 안 된다.

어떻게 아는가

한 단위가 닫혀 있는지 어떻게 판별하는가.

한 가지 실용적 기준이 있다. **그것을 검증할 수 있는가.**

어떤 모듈이 옳게 동작하는지를 그 모듈만 가지고 확인할 수 있으면, 그것은 닫혀 있다. 확인하려고 다른 모듈을 실제로 띄워야 한다면, 닫혀 있지 않다.

목과 픽스처 (mock / fixture)

목은 테스트에서 진짜 의존 대상 대신 끼워 넣는 가짜 객체다 — 결제 모듈을 테스트하려는데 실제 결제사에 요청을 보낼 수 없으니 "성공했다고 답하는 가짜"를 넣는다. **픽스처**는 테스트가 돌기 위해 미리 갖춰 놓아야 하는 데이터와 상태다. 이 둘의 크기가 유용한 계측기인 이유가 있다 — **목을 몇 개 만들어야 하는지가 그 단위가 바깥과 맺은 관계의 수**이고, 픽스처가 얼마나 큰지가 그 단위가 전제하는 세계의 크기다. 테스트를 짜기 고통스럽다는 감각은 취향이 아니라 측정값이다.

이 기준이 유용한 이유는 기계적이기 때문이다. 테스트를 짜 보면 알 수 있다. 목을 몇 개 만들어야 하는지, 픽스처가 얼마나 큰지가 그 단위의 열림 정도를 말해 준다.

그리고 이 기준은 3부의 다음 장으로 이어진다. 검증할 수 있다는 것과 계약이 있다는 것은 같은 말이다.

크기는 여전히 문제다

닫힘이 크기를 대체하지는 않는다.

완전히 닫힌 단위라도 그 안이 10만 줄이면 여전히 다룰 수 없다. 안에서 완결된다는 것과 안이 단순하다는 것은 다르다.

그래서 이 책의 처방은 두 낱말로 되어 있다. **최대한 작은 닫힌 계.** 두 조건이 다 필요하다.

닫혀 있으면서 작은 것. 닫혀 있지만 크면 안이 여전히 부하고, 작지만 열려 있으면 바깥이 부하다.

대가

닫힌 계로 나누면 값을 치른다. 13장에서 열거한 것과 같은 종류다.

전체를 아는 존재가 없어진다. 각 조각을 판단할 수 있게 만든 대가로, 조각들이 합쳐졌을 때 무슨 일이 일어나는지를 판단할 수 있는 자리가 사라진다.

그리고 닫으려면 중복이 생긴다. 두 모듈이 같은 개념을 각자 정의하게 된다. 공유하면 관계가 생기고, 관계가 생기면 닫히지 않는다. 닫으려면 중복하고, 중복하면 두 곳이 어긋난다.

> **DRY와의 충돌**
>
> "같은 것을 반복하지 마라"(Don't Repeat Yourself)는 1999년 『실용주의 프로그래머』가 정식화한 이래[1] 가장 널리 받아들여진 원칙 중 하나다. 그런데 그 원칙은 **공유를 늘리는 방향**으로 작동하고, 공유는 관계를 만든다. 닫힘을 우선하면 DRY를 일부 포기해야 한다. 이 충돌은 새것이 아니다 — 마이크로서비스 진영에서도 "서비스 간에는 코드를 공유하지 말고 차라리 복사하라"는 조언이 오래 있었다.[2] 다만 이 책의 기준에서는 그 조언의 근거가 팀 자율성이 아니라 **판단의 완결성**이 된다.

이 교환은 피할 수 없다. 어긋남을 감수하고 닫을 것인가, 관계를 감수하고 공유할 것인가.

남은 질문

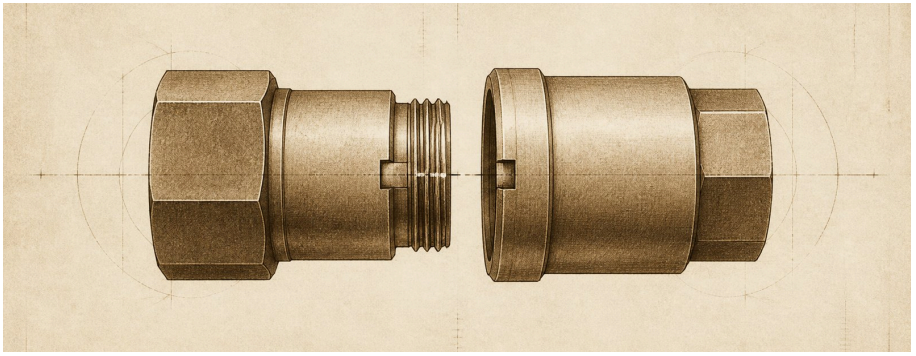
닫힌 계를 만들었다고 하자.

그것들을 이어붙일 때 안을 열어봐야 한다면, 지금까지 한 일은 무엇이 되는가.

주

1 Andrew Hunt, David Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999. 원문의 정의는 "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"이며, 코드 중복 자체가 아니라 **지식의 중복**을 겨냥한 것이었다. 이 구분은 이 절의 충돌을 완화한다 — 두 모듈이 같은 개념을 각자 정의하는 것이 언제 "지식의 중복"이고 언제 "우연한 형태의 일치"인지가 판단의 대상이다. ↩

2 Sam Newman, *Building Microservices*, 2nd ed., O'Reilly, 2021 — "서비스 경계를 넘는 코드 재사용을 결합으로 취급하라"는 조언이 여러 장에 걸쳐 나온다. 구글의 자바 스타일 가이드와 Go 커뮤니티의 "a little copying is better than a little dependency"(Rob Pike, Go Proverbs)도 같은 계열의 조언이다. ↩



20장

내부를 모른 채 꽂는다

되돌아오는 부하

닫힌 계를 스무 개 만들었다고 하자.

그것들을 조립할 때 각각의 안을 열어봐야 한다면, 조립하는 쪽의 부하는 스무 개의 합이 된다. 나누기 전과 같아진다.

쪼개기는 절반이다. 나머지 절반은 **안을 모른 채 이을 수 있어야 한다**는 것이다.

1장의 정원 호스가 여기서 다시 온다. 호스를 이을 때 앞 세그먼트 안에서 물이 어떤 경로로 흘렀는지 묻지 않는다. 나사산만 맞으면 된다.

계약

안을 모른 채 이을 수 있게 하는 것이 계약이다.

계약이 하는 일은 두 가지다. 무엇을 주면 무엇이 나오는지를 정한다. 그리고 그 외의 것에 대해서는 **알 필요가 없다고 선언한다.**

두 번째가 더 중요하다. 계약은 정보를 주는 문서이면서 동시에 정보를 차단하는 벽이다. 이 선 안쪽은 당신이 알 바 아니라고 말하는 장치다.

파나스의 정보 은닉과 같은 것을 반대편에서 본 것이다. 감추는 쪽에서 보면 은닉이고, 쓰는 쪽에서 보면 계약이다.

7장의 문장

계약에 대해 이 책이 이미 도달한 문장이 있다.

계약은 선언된다고 계약이 되지 않는다. 강제될 때만 계약이다.

REST가 그것을 보여줬다. 명세가 있고 논증이 있고 저자가 직접 오해를 정정했는데, 그것을 어겨도 아무 일도 일어나지 않았다. 그래서 대부분의 구현이 그 계약 바깥에 있다.

semver는 반대편이다. 규칙을 어겼는지 사람이 검사하지 않는다. 패키지 매니저가 그 숫자를 읽고 행동한다. 강제하는 것이 도구다.

12장은 한 회사가 이 차이를 자기 코드베이스에서 실험으로 확인한 기록이다. 관측 도구는 위반을 알려줬고, 알려주는 것만으로는 부족했다. 정적 분석기는 위반을 머지 전에 거부했다.

기계가 쓸 때 무엇이 달라지는가

사람이 코드를 쓸 때, 계약은 반쯤 강제되어도 작동했다.

문서에 적힌 규칙을 사람은 어느 정도 지킨다. 코드 리뷰가 잡아낸다. 팀에 오래 있은 사람이 왜 그 규칙이 있는지를 안다. 규칙을 어기면 누군가 화를 낸다.

이 장치들의 공통점이 하나 있다. **사람이 맥락을 축적한다는 전제**다.

12장의 후속 글이 그 전제를 문제 삼았다. 중앙 팀이 좋은 설계를 만들어내도, 그 팀이 관심을 다른 데로 돌리는 순간 설계는 퇴화한다는 것. 맥락을 쥐고 있던 주의력이 옮겨가면 경계도 같이 흐려진다.

기계는 그 상황의 극단이다. 매번 새로 들어오고 매번 떠난다. 지난주에 왜 그렇게 했는지를 기억하지 않는다. 규칙이 문서에 있으면 그 문서를 읽었을 수도 있고 안 읽었을 수도 있다. 읽었어도 그것이 왜 중요한지에 대한 감각은 없다.

> **에이전트는 왜 "매번 새로 들어오는 팀원"인가**
>
> 사람 팀원은 지난달의 코드 리뷰에서 지적받은 것을 기억하고, 그 규칙이 어떤 장애 때문에 생겼는지 알고, 어기면 누가 화를 낼지 안다. 에이전트에게는 이 셋이 다 없다. 세션이 끝나면 맥락이 사라지고, 다음 세션은 저장소에 적혀 있는 것에서 다시 시작한다. 규칙 파일(`CLAUDE.md`, `AGENTS.md` 같은 것)을 두는 관행이 생긴 이유이기도 하지만, **그것은 문서지 강제가 아니다** — 읽을 수도 있고 안 읽을 수도 있으며, 읽어도 그 규칙의 무게를 모른다.

그러므로 반쯤 강제되는 계약은 기계에게 강제되지 않는 계약이다.

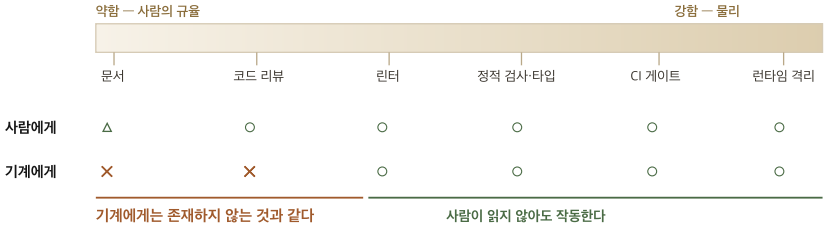


그림 18 — 강제 수단의 스펙트럼. 사람에게 통하던 왼쪽 절반이 기계에게는 통하지 않는다.

그래서 서비스가 된다

작은 닫힌 계를 만들고, 그것들을 계약으로만 잇고, 그 계약을 어길 수 없게 만든다.

이것을 실제로 구현하는 방법을 나열해 보면 익숙한 목록이 나온다.

각각을 독립적으로 실행 가능하게 만든다. 각각을 독립적으로 배포한다. 상태를 공유하지 않는다. 통신은 명시적인 인터페이스로만 한다. 인터페이스를 어기면 실패한다.

작은 닫힌 계 + 계약으로 조립 = 우리가 이미 마이크로서비스라고 부르는 것.

물러서지 않는다

여기서 흔한 절충이 하나 있다. 논리적으로는 서비스처럼 나누되 물리적으로는 한 덩어리로 두자는 것. 네트워크 없는 마이크로서비스.

이 책은 그 절충을 주장하지 않는다.

이유는 두 가지다.

첫째, 그 절충을 택하는 순간 강제 수단이 약해진다. 프로세스가 나뉘어 있으면 다른 프로세스의 메모리를 읽을 방법이 없다. 같은 프로세스 안에 있으면 방법이 있고, 그것을 막는 것은 정적 분석기나 타입 검사기다. 강제되기는 하지만 우회 가능한 강제다.

둘째, 그 절충을 택하면 주장이 자명해진다. 논리적 경계를 잘 그으라는 말은 1972년부터 있었다. 그것만 말할 거면 이 책이 필요 없다.

이 책의 테제가 말하는 것은 논리적 경계가 아니라 서비스다. 그 지점에서 물러서지 않는다.

반론 — 조정 비용

여기서 정면으로 받아야 할 반론이 있다.

마이크로서비스가 존재한 이유는 **사람 사이의 조정이 비쌌기 때문**이다. 팀이 서로 기다리지 않게 하려고 배포를 나눴다. 팀이 서로의 코드를 몰라도 되게 하려고 인터페이스를 세웠다.

에이전트 사이의 조정 비용은 0에 가깝다. 회의가 없다. 서로 기다리지 않는다. 롤백은 그냥 다시 실행하는 것이다.

콘웨이를 낳은 경제학이 사라진다면, 그 산물인 서비스 경계도 사라져야 하지 않는가.

답

이 책의 주장은 조정 비용에 걸려 있지 않다. **인지부하**에 걸려 있다.

두 제약은 다르다. 조정이 공짜가 되어도 한 번에 판단할 수 있는 양은 늘지 않는다. 17장의 실험이 그것을 보여줬다. 아무도 조정하지 않는 단일 모델이, 아무 방해 없이, 검색까지 완벽한 상태에서 길이 때문에 틀렸다.

그리고 하나 더 있다. 생산이 싸지고 조정이 싸지면, **남는 비싼 것은 검증**이다.

코드를 만드는 비용이 내려가면 만들어진 것이 옳은지 확인하는 일이 병목이 된다. 그리고 확인은 경계 단위로만 가능하다. 무엇이 옳은지 말하려면 무엇에 대해 옳은지를 정해야 하고, 그것이 계약이다.

19장의 판별 기준이 여기서 다시 온다. 닫혀 있는지를 아는 방법은 그것만 가지고 검증할 수 있는지 보는 것이었다.

검증 단위와 경계 단위는 같은 것이다.

대가

계약으로만 잇는 데에도 값이 붙는다. 9장에서 열거한 것들이 그대로 돌아온다.

경계를 넘는 호출은 실패할 수 있다. 트랜잭션이 경계를 넘지 못한다. 디버깅이 여러 곳에 흩어진다.

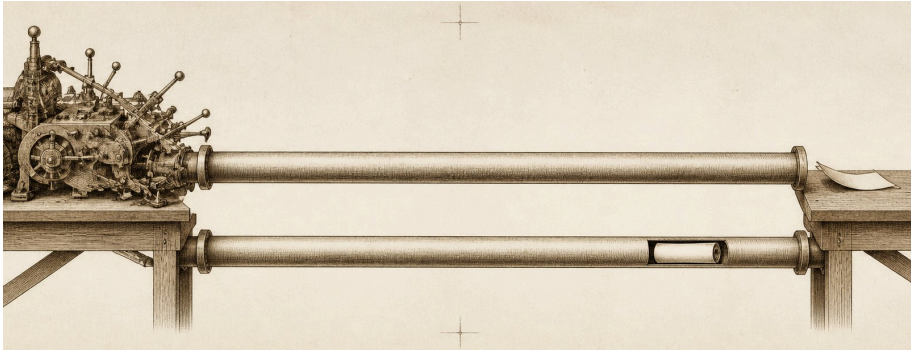
그리고 하나 더 있다. 계약을 바꾸는 일이 어려워진다. 안을 바꾸는 것은 자유롭지만 계약을 바꾸려면 양쪽이 같이 움직여야 한다. 경계는 굳는다.

굳는 것이 이득이면서 동시에 비용이다. 굳어야 안을 모른 채 쓸 수 있고, 굳었기 때문에 잘못 그은 선을 고치기 어렵다.

남은 질문

여기까지가 이 책의 처방이다. 그런데 처방을 내놓았으면 반례를 봐야 한다.

AI가 코드를 고치는 현장에서 실제로 이기고 있는 것은 분업이 아니다. 다음 장이 그 표를 먼저 놓고 시작한다.



21장

통로가 좁을 때만

불편한 리더보드

앞 장까지의 논증은 이렇게 끝났다. 작은 닫힌 계를 만들고 계약으로만 잇는다. 그것이 우리가 이미 마이크로서비스라고 부르는 것이다.

여기서 자연스러운 다음 문장이 하나 있다. **AI도 그러니 여러 개로 나뉠 것이다.** 하나가 전체를 감당 못 하니 편성이 되고, 편성이 시스템 구조를 정한다는 것.

그 문장을 쓰기 전에 봐야 할 표가 있다.

SWE-bench Verified는 실제 오픈소스 저장소의 이슈를 에이전트에게 주고 고치게 하는 벤치마크다. 리더보드 전체를 훑으면 이렇게 되어 있다.[1]

해결률	시스템	구조
79.2%	live-SWE-agent	**단일 에이전트** — bash 하나로, 필요한 도구는 실행 중에 스스로 만든다
73.2%	Tools + Claude 4 Opus	단일, 맨몸 스캐폴드
72.8%	**mini-SWE-agent**	**100줄. bash 외 도구 없음. 이력이 완전히 선형**
63.4%	AgentScope	역할 분업
51.6%	SpecRover	역할 분업
32.6%	MASAI	역할 분업
28.3%	CodeR	역할 분업

상위권에 협업형 멀티에이전트가 없다. 100줄짜리 선형 에이전트가 정교한 역할 편성을 40포인트 앞선다.

상위권에 "멀티에이전트"라는 이름이 붙은 것들도 협업이 아니다. 독립적인 시도를 여러 개 만든 뒤 그중 하나를 고르는 생성-선별 구조다. **서로 이야기하지 않는다.**

이 표를 빼고 "AI도 분산 시스템이 된다"를 쓰면, 이 책이 6장과 8장에서 지적한 일을 하는 것이 된다.

모델 안쪽도 증거가 아니다

또 하나의 유혹적인 재료가 있다. **혼합 전문가**(Mixture-of-Experts, MoE) 구조다. 요즘 큰 모델들은 내부가 수십 개의 "전문가"로 나뉘어 있고, 토큰마다 그중 일부만 켜진다. 최신 모델 하나는 6,710억 파라미터 중 토큰당 370억만 쓴다 — 5.5%다.

"모델조차 이미 내부적으로 분업하고 있다"고 쓰고 싶어진다.

> **혼합 전문가 (Mixture-of-Experts, MoE)**
>
> 신경망 층 하나를 여러 개의 하위 신경망("전문가")으로 나누고, 게이팅 망이 입력마다 그중 소수만 골라 켜는 구조. 2017년 논문이 제안했다.[2] 이름이 "전문가"라 분업처럼 들리지만, 원 논문이 세운 문제는 **"파라미터를 늘리되 계산량은 늘리지 않는 법"**이다. 신경망이 흡수하는 정보량은 파라미터 수에 묶여 있는데 파라미터를 늘리면 계산이 같이 느는 문제를, 예제마다 망의 일부만 켜서 떼어냈다. 논문이 내건 성과도 성능이 아니라 비율이다 — 계산 비용을 낮추면서 모델 용량 1000배 이상.

이건 용량의 공학이지 인지의 분업이 아니다. 원 논문의 문제 설정이 그렇고, 9년 뒤 현행 모델의 자랑도 여전히 비용이다. "이미 분업하고 있다"로 읽는 것은 저자들이 하지 않은 말을 대신 해 주는 일이다.

같은 이유로 파이프라인 병렬화도 증거가 아니다. 큰 모델을 여러 GPU에 쪼개 올리는 것은 **기반을 나눈 것**이지 판단을 나눈 것이 아니다.

그러면 무엇이 가르는가

멀티에이전트가 이긴 사례도 분명히 있다. 리서치 과제에서는 편성이 단일 에이전트를 큰 폭으로 앞선다. 분해 가능한 추론에서도 그렇다. 그런데 코드 연쇄 수정에서는 20장에서 본 것처럼 오히려 나빴다.

같은 기술이 어디서는 이기고 어디서는 진다면, **가르는 축이 따로 있다는 뜻이다.**

2026년의 한 연구가 그 축을 정보 병목 문제로 형식화했다. 출발점이 결정적이다.

> 통로의 대역폭이 무한하다면, 어떤 단일 에이전트도 상류의 전체 맥락을 그대로 전달하는 멀티에이전트로 모사할 수 있다.[3]

즉 통로가 무한하면 둘은 **구별되지 않는다.** 따라서 멀티에이전트의 이득도, 손해도, **통로가 좁을 때만** 생긴다. 그리고 통로가 좁으면 압축이 일어나고, 압축에는 근본적인 교환이 붙는다.

축이 하나로 정리된다. 하위 작업이 상류로 돌려보내야 하는 것이, 돌려보낼 수 있는 크기에 들어가는가.

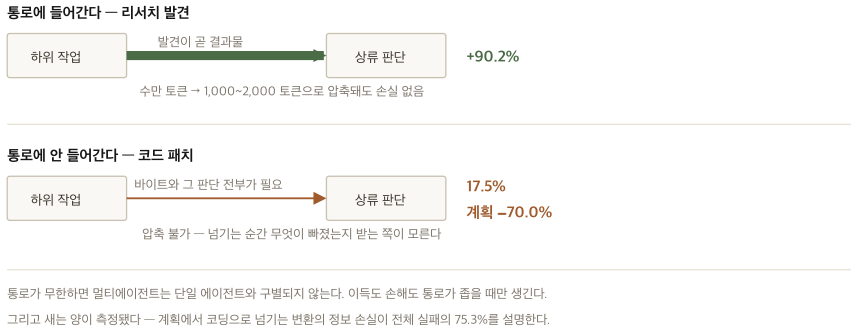


그림 21 — 같은 편성이 어디서는 이기고 어디서는 지는 이유. 축은 협업의 정교함이 아니라 통로의 폭이다.

이 축이 사례들을 전부 설명한다.

리서치는 발견이 곧 결과물이다. 하위 에이전트가 수만 토큰을 읽고 알아낸 것을 천 몇백 토큰으로 적어 올려도, 상류가 필요로 하는 것은 그 안에 다 들어간다. 통로 손실이 사실상 0이다.

코드 패치는 그렇지 않다. 바이트 자체와, 왜 그렇게 고쳤는지에 대한 판단 전부가 하류에 필요하다. 요약할 수 있는 것이 아니다. 18장에서 본 결과 — 요약 전달과 편집 위임이 원본 이력을 그대로 들고 가는 것보다 나빴던 것 — 이 여기서 설명된다.

그리고 통로에서 실제로 얼마가 새는지도 측정됐다. **계획 단계에서 코딩 단계로 넘어가는 변환에서 발생하는 정보 손실이 전체 실패의 75.3%를 설명한다.**

그런데 한 랩이 이 책의 처방을 제품으로 출하했다

여기까지는 논증이다. 그리고 논증만 있으면 이 책은 또 콘웨이가 1967년에 있던 자리에 있게 된다.

2026년 7월, 한 프런티어 랩이 새 모델의 시스템 카드를 193쪽으로 냈다. 그 안에 편성 실험 절이 있고, 첫 문장이 이렇다.

> **멀티에이전트 하네스가 가장 높은 점수를 냈고 점수-지연 프런티어를 파레토 지배한다. 모든 멀티에이전트 변형이 최고의 단일 에이전트 변형과 같거나 그것을 넘었으며, 10-에이전트 팀이 우리의 최고 점수인 93.6%에 도달했다** — 최고 단일 에이전트 기준선 대비 +3.1%p.[4]

100만 토큰을 쓰는 단일 에이전트 기준선 대비 5.9배 빨랐다.

앞의 리더보드와 정반대로 보인다. 그런데 **편성의 설계를 읽으면 정반대가 아니다.** 카드가 적은 구성이 이 책의 어휘 그대로다.

정보 은닉 — *"각 하위 에이전트는 리드가 제공한 지시만 볼 뿐, 원래 과제 설명은 보지 못한다."*

닫힌 계와 계약 — *"각 에이전트는 과제 저장소의 자기 체크아웃 안에서 작업하고, 코드는 깃을 통해 다른 에이전트와 공유한다."* 각자 자기 사본 안에서 완결하고, 경계를 넘는 것은 커밋이다.

이건 임의로 나눈 편성이 아니다. **통로를 좁게 만들어 놓고 나눈 편성**이다. 하위 에이전트에게 넘어가는 것은 리드가 추린 지시뿐이고, 돌아오는 것은 커밋이다. 둘 다 통로에 들어가는 형태다.

그리고 더 결정적인 것이 다른 절에 있다. 같은 카드의 "긴 컨텍스트" 절에 벤치마크가 딱 하나 있고, 채점 방식이 이렇다.

> *"5개 에피소드. 각 에피소드는 이전 에피소드의 코드베이스에서 이어지며, 최대 100만 토큰의 **새 컨텍스트 예산**으로 시작한다."*

83%에서 93%로 오른다. 그리고 카드는 이것을 장문맥 코딩 성능의 강한 척도라고 부른다.

이 랩이 정의하는 장문맥 능력이 "컨텍스트를 리셋하고, 상태를 코드베이스에 남기고, 다시 시작하는 것"이다. 창을 크게 쓰는 능력이 아니라, 창을 버리고 밖에 남긴 것으로 이어가는 능력이다.

이 책이 20장에서 도달한 문장을 다시 놓으면 겹친다 — 기계는 매번 새로 들어오고 매번 떠나므로, 경계는 그것을 모르는 존재도 어길 수 없는 형태로 있어야 한다. **상태를 밖에 남기고 컨텍스트를 리셋한다는 것이 정확히 그 처방의 구현이다.**

다만 카드 자신이 단서를 단다. **비용 승리가 아니다** — 에이전트 수가 늘면 비용이 오른다고 명시한다. 지연시간은 벽시계 측정이 아니라 파생값이고, 실험은 사전 릴리즈 구성에서 안전장치를 끈 채 돌았으며, 카드는 이 수치가

상대적 성능이지 절대적 성능이 아니라고 적는다. 그리고 무엇보다 **벤더가 자기 모델에 대해 낸 숫자다.**

이 축은 이 책이 이미 쓴 것이다

여기서 되짚어야 할 것이 있다. 19장이 닫힘을 판별하는 실용 기준으로 제시한 것은 **검증할 수 있는가**였다. 그리고 20장은 이렇게 닫았다 — **검증 단위와 경계 단위는 같은 것이다.**

위 형식화에서 그 문장이 정리로 나온다.

검증은 정확성에 대한 무손실 압축이다. 어떤 조각이 옳은지를 합격/불합격 한 비트로 실어 나를 수 있으면, 하류가 그 조각에 대해 필요로 하는 유일한 속성이 통로를 온전히 통과한다. 전체를 넘기지 않아도 된다.

독립성은 그 축의 또 다른 특수해다. 서로 독립인 하위 작업은 통로 손실이 구조적으로 0이다 — 하류가 필요로 하는 것이 애초에 없기 때문이다.

즉 이 책이 사람의 이유로 도달했던 두 기준 — 닫혀 있을 것, 계약으로 이을 것 — 이 기계 쪽에서 **같은 축의 두 특수해**로 다시 나온다. 우연이 아니라 같은 정리의 두 얼굴이다.

그래서 리더보드는 무엇을 재고 있었나

이제 앞의 불편한 표로 돌아갈 수 있다.

SWE-bench가 재는 것은 에이전트의 편성이지 코드베이스의 경계가 아니다. 과제가 이미 저장소 하나, 이슈 하나로 잘려서 들어온다. 무엇을 어떻게 잘라 줬는지는 채점되지 않는다.

경계가 이미 그어진 문제를 주고 "누가 더 잘 푸나"를 물으면, 답은 **한 마리에 좋은 도구다.** 통로를 만들 이유가 없는데 통로를 만들면 손해만 나기 때문이다. 100줄짜리 선형 에이전트가 이기는 것이 당연하다.

이 책의 주장은 그 자리에 있지 않다. **경계를 누가 그어 주느냐**에 있다. 그리고 18장에서 본 것처럼, 16만 2천 줄이 통째로 들어오면 네 에이전트가 전부 0을 받는다.

리더보드는 잘 잘린 문제를 푸는 능력을 재고, 이 책은 자르는 일을 이야기한다. 둘은 다른 질문이다.

가장 강한 반론 — 능력이 오르면 부호가 바뀐다

이 장에 대한 가장 무거운 반론은 동료심사를 통과했고, 제목이 곧 반론이다.

「유능한 언어 모델은 협업의 이득을 넘어서 자란다」[5]

연구진은 260개 구성을 통제한 뒤 뒤집히는 지점을 숫자로 냈다. **단일 에이전트의 정확도가 45%를 넘는 과제에서는, 에이전트를 추가하면 음의 수익이 난다.** 이 임계값은 SWE-bench Verified와 Terminal-Bench 검증 구성의 94%에서 편성 효과의 방향을 맞혔다. 단일 대비 성능은 +80.8%에서 −70.0%까지 벌어진다.

같은 연구가 조정 오버헤드를 초선형으로 측정했다. 지수 1.724 — 브룩스의 2.0보다는 완만하지만 여전히 선형 위다.

왜 이것이 무거운가. 이 책은 기계의 인지부하가 유한하므로 분산으로 간다고 말한다. 그런데 이 결과는 **분산의 이득이 모델 능력의 함수**이고, 능력이

오르면 그 이득이 **부호를 바꾼다**고 말한다. 이득은 능력과 함께 줄고 조정 비용은 그대로 붙으니, 교차점은 반드시 온다.

예산 쪽에서도 같은 방향의 결과가 있다. 사고 토큰을 고정해 놓고 비교하면 단일 에이전트가 일관되게 같거나 낫고, 보고된 멀티에이전트의 이점은 구조적 이점이 아니라 **계산되지 않은 연산량**으로 더 잘 설명된다는 것이다.[6]

이 책의 답은 하나뿐이고, 그것은 방어가 아니라 인정이다.

분산은 종착지가 아니라 **현재 능력 수준에서의 국소해**다. 그리고 그것이 이 책이 처음부터 말해 온 것이다 — 24장의 시효가 이 문장을 위해 있다. 다만 이 연구는 그 시효에 **숫자를 하나 준다.** "3년"이라는 근거 없는 어림 대신, **단일 에이전트 정확도 45%**라는 관측 가능한 임계값이 생겼다.

그래서 24장의 만료 조건에 이것을 추가한다. **대상 과제에서 단일 에이전트 정확도가 안정적으로 45%를 넘고, 그 위에서 편성이 계속 지는 것이 재현되면, 이 장은 끝난다.**

그러면 AI는 분산 시스템이 되는가

정확한 형태로 다시 쓰면 이렇게 된다.

하나의 AI가 통째로 다룰 수 없는 규모의 시스템에서, 그것을 다루려면 통로에 들어가는 단위로 잘라야 한다. 그리고 통로에 들어가게 만드는 방법이 곧 계약이고 검증 가능성이다.

이건 "AI가 여러 마리로 나뉜다"와 다른 주장이다. 에이전트를 몇 마리 쓰느냐는 편성의 문제이고, 위 문장은 **시스템의 형태**에 대한 것이다. 실제로 리더보드가 보여주듯, 잘 잘린 시스템에서는 한 마리로 충분할 수도 있다.

나뉘어야 하는 것은 에이전트가 아니라 시스템이다. 20장의 구분 — 작업을 나눈 것과 시스템을 나눈 것은 다르다 — 이 여기서 마지막 형태를 얻는다.

대가

이 축에도 값이 붙는다.

통로에 들어가게 만들려면 경계를 통로 폭에 맞춰 그어야 한다. 그런데 도메인이 실제로 그 선에서 갈리지 않을 수 있다. 13장에서 본 구분이 여기서 돌아온다 — 지울 수 있는 얽힘과 도메인이 진짜로 결합된 얽힘은 다르고, 후자를 억지로 자르면 자른 자리에서 계속 새는 통로가 생긴다.

그리고 이 축은 **측정하기 어렵다.** 어떤 조각이 통로에 들어가는지를 미리 아는 방법이 없다. 알 수 있는 것은 사후뿐이다 — 나눠 보고 나빠지면 안 들어간 것이다.

남은 질문

통로가 넓어지면 이 모든 것이 사라진다.

무한 대역폭에서 멀티에이전트가 단일 에이전트와 구별되지 않는다면, 그 역도 참이다. **충분히 넓은 통로에서는 나눌 이유가 없다.**

그러면 다음 질문은 이것이다. 지금 통로를 좁게 만들고 있는 것은 무엇인가. 그것은 물리인가, 아니면 이번 세대의 공학인가.

주

1 SWE-bench Verified 공개 리더보드(2026-08 시점 115개 항목). live-SWE-agent와 mini-SWE-agent는 도구를 최소화한 단일 에이전트 계열이고, 역할 분업형(AgentScope·SpecRover·MASAI·CodeR)은 전부 그 아래에 있다. 상위권의 일부 "멀티에이전트" 표기 시스템은 독립 시도를 N개 생성한 뒤 선별하는 구조로, 에이전트 간 통신이 없다. ↩

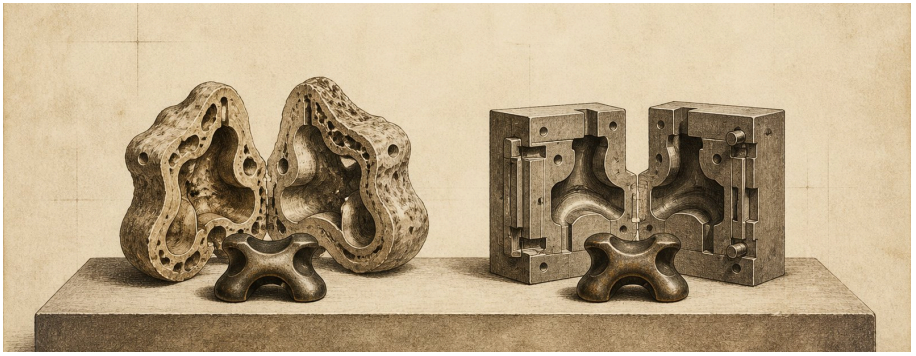
2 Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, Jeff Dean, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer", arXiv:1701.06538, ICLR 2017. 5.5% 활성화 수치는 DeepSeek-V3 기술 보고서(arXiv:2412.19437)의 6,710억 총 파라미터 / 토큰당 370억 활성화다. ↩

3 Yu et al., arXiv:2607.16133, 2026. 멀티에이전트 시스템의 이득을 통로 대역폭 제약 하의 정보 병목으로 형식화한 연구. 원문의 출발 명제: "under infinite relay bandwidth, any SAS can be simulated by a MAS that transmits the full upstream context" → "the nontrivial advantage of MAS arises under bounded relays" → "compression introduces a fundamental trade-off." 본문의 +90.2% · 17.5% · −70.0% · 75.3%는 이 계열 연구들의 보고값이며, 이 책은 개별 수치를 논증의 하중에 쓰지 않고 **방향과 축**만 취한다. ↩

4 Anthropic, Claude Opus 5 시스템 카드, 2026-07-24, 193쪽. §8.11(편성 실험) 및 §8.9(긴 컨텍스트). 인용은 원문 번역이며, 같은 카드가 명시한 유보를 함께 옮겼다 — 에이전트 수에 따라 비용이 오르고, 지연시간은 파생값이며, 사전 릴리즈 구성에서 안전장치를 끈 채 측정했고, 상대적 성능이지 절대적 성능이 아니다. 벤더가 자사 모델에 대해 낸 수치라는 점도 등급에 반영해야 한다. ↩

5 Kim 외 19인, "Capable language models can outgrow the benefits of collaboration", *Nature Machine Intelligence* 8, 2026, pp. 1157–1172. 이 장이 인용하는 자료 중 **유일하게 동료심사를 통과한 것**이다. ↩

6 Tran & Kiela, arXiv:2604.02460 (스탠퍼드). 사고 토큰 예산을 고정한 통제 실험. 그리고 널리 도는 "멀티에이전트 오류 증폭 17배"라는 수치는 흔히 MAST 연구에 귀속되는데 **그 논문에 그 숫자가 없다** — 출처는 위 Kim 외이고, 저자들은 통제 후 그 효과가 유의성을 잃는다고 명시한다. ↩



22장

같은 모양, 다른 이유

목록

20장 끝에서 나온 목록을 다시 놓아 보자. 앞 장이 단서를 달았다. 이 목록이 이기는 것은 통로가 좁을 때만이다.

작은 단위. 명시적 계약. 격리된 상태. 실패의 국소화. 안을 모른 채 이을 수 있는 연결. 단, 그 연결의 통로는 좁아야 한다.

이 목록은 이 책이 처음 만든 것이 아니다.

1964년

맥일로이의 메모에 있던 것이 안을 모른 채 꽂는 연결이었다.

정원 호스의 세그먼트를 이을 때 앞 세그먼트의 내부를 묻지 않는다. 나사산만 맞으면 된다. 다른 방식으로 데이터를 주무를 필요가 생기면 하나

더 끼운다.

지금 에이전트에게 필요한 것이 정확히 그것이다. 이 모듈이 무엇을 받고 무엇을 내는지만 알면 되고, 그 안에서 어떻게 하는지는 열어보지 않아도 되는 상태.

1972년

파나스가 감추라고 한 것이 정보 은닉이었다.

바뀔 것 같은 결정을 모듈 안에 넣고, 바깥에서는 그 결정이 존재한다는 것조차 모르게 한다.

기준이 달라졌을 뿐이다. 그는 무엇이 바뀔지를 기준으로 삼았고, 지금 우리는 무엇을 안 봐도 되는지를 기준으로 삼는다. 감추는 행위 자체는 같다.

1973년

휴잇의 액터가 사적 상태와 메일함이었다.

상태는 공유되지 않는다. 경계를 넘는 것은 메시지뿐이다. 한 번에 하나씩 처리한다.

지금 여러 에이전트를 편성해서 일을 시키는 구조를 그려 보면 같은 그림이 나온다. 각 에이전트는 자기 컨텍스트를 가지고 있고, 그 컨텍스트는 다른 에이전트와 공유되지 않는다. 오가는 것은 메시지다. 요청과 결과와 핸드오프 문서.

에릭슨의 교환기

슈퍼바이저 트리와 let it crash가 있었다.

말단이 죽는다. 부모가 그것을 보고 재시작하거나 멈추거나 위로 올린다. 모르는 오류를 처리하려 들지 않는다.

여러 에이전트를 돌려 본 사람이 그리게 되는 구조가 이것이다. 오케스트레이터가 있고 그 아래에 작업 에이전트들이 있다. 하나가 실패하면 그 결과를 버리고 다시 시킨다. 실패한 에이전트가 스스로를 복구하려고 애쓰는 것보다 그편이 낫다.

통신 교환기를 위해 만들어진 구조가, 2026년에 코드를 쓰는 기계들의 편성으로 다시 나타난다.

구조	1964-73 왜 필요했나	2012- 왜 필요했나	2026- 왜 필요한가
작은 단위	사람의 이해 한계	한 팀 소유 크기	기계의 판단 한계
명시적 계약	언어·기계가 달라서	팀 간 협상 비용	맥락 없는 소비자
격리된 상태	데이터 경합 제거	독립 배포	컨텍스트 오염 차단
실패의 국소화	교환기 무중단	장애 전파 차단	재실행이 더 쌈
안을 모르는 연결	조립 자유	팀 자율성	검증 단위 확보

왼쪽 두 열의 이유는 사라지지 않는다(사람의 상한은 1956년 이후 그대로다). 오른쪽 열의 이유에는 날짜가 붙어 있다.

그림 19 — 같은 다섯 부품이 세 시대에 다른 이유로 요구된다.

발명이 아니다

이 목록을 나란히 놓으면 하나가 분명해진다.

우리는 발명하고 있는 게 아니라 재발견하고 있다.

마이크로서비스는 인간의 인지부하 때문에 만들어졌다. 한 팀이 담을 수 있는 크기. 한 사람이 이해할 수 있는 범위. 조정 비용이 감당 가능한 규모.

이제 같은 모양이 기계의 인지부하 때문에 다시 요구된다.

그러나 값싼 수사가 아니다

여기서 조심할 것이 있다.

역사는 반복된다는 문장은 아무것도 설명하지 않는다. 이 장이 하려는 말은 그것이 아니다.

제약의 종류가 우연히 같은 형태를 요구한 것이다.

인간의 작업 기억에 상한이 있어서 모듈이 생겼다. 기계의 실효 처리 범위에 상한이 있어서 같은 모듈이 다시 필요해졌다. 두 상한은 완전히 다른 것이다. 하나는 생물학이고 하나는 어텐션 메커니즘의 성질이다. 크기도 다르고 성장 속도도 다르다.

같은 것은 상한이 있다는 사실뿐이고, 상한이 있으면 나눠야 한다는 것은 거의 논리적 필연에 가깝다.

그래서 유효 기간이 다르다

그리고 이 구분에서 이 책의 결론이 나온다.

형태가 같은 이유가 제약 때문이라면, **제약이 사라지면 형태도 사라진다.**

인간의 작업 기억 상한은 사라지지 않았다. 1956년에 측정된 것이 2026년에도 같다. 그래서 인간을 위한 모듈화는 70년째 유효하다.

기계의 상한은 다르다. 그것은 생물학이 아니다. 엔지니어링이고, 매년 움직인다.

그러므로 이 책의 후반부에는 유효 기간이 붙어 있다.

대가

재발견에는 위험이 하나 따라온다.

같은 모양이라고 해서 같은 이유로 좋은 것은 아니다. 마이크로서비스를 인간의 이유로 정당화하는 논리들 — 팀 자율성, 독립 배포 주기, 조직 확장 — 은 기계에게는 적용되지 않는다.

그 논리들을 그대로 들고 와서 쓰면 잘못된 결정을 하게 된다. 예를 들어 팀 경계를 따라 서비스를 나누는 것은 인간의 이유로는 옳지만, 기계의 이유로는 근거가 없다. 기계의 이유로는 **판단이 완결되는 선**을 따라 나눠야 한다.

같은 모양을 만들되 다른 지도를 보고 만들어야 한다.

남은 질문

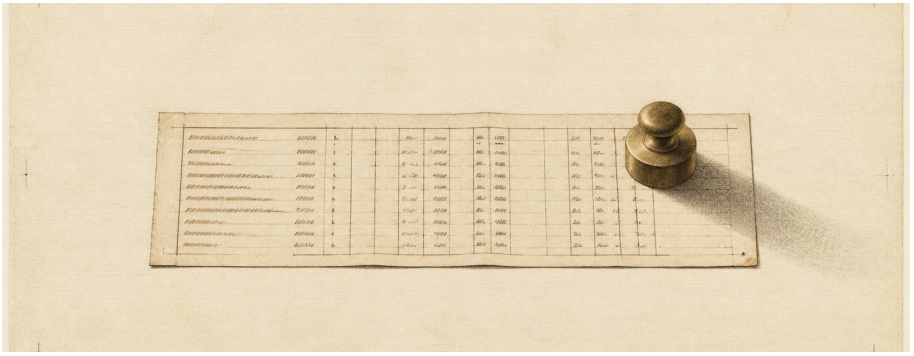
19세기의 어떤 기계와 21세기의 어떤 기계가 같은 형태를 가졌다면, 그것은 물리 법칙이 같아서인가 아니면 만든 사람이 같은 책을 읽어서인가.

지금 우리가 에이전트 편성을 그리면서 액터 모델과 비슷한 그림에 도달하는 것은, 그것이 유일한 답이어서인가 아니면 우리가 그 그림을 이미 알고 있어서인가.

4 부

시효

이 진단에는 날짜가 붙어 있다.



23장

청구서는 그대로다

이 책이 말하지 않는 것

지금까지의 논증은 마이크로서비스가 좋다고 말하지 않았다.

말한 것은 다른 것이다. 지금으로서는 그럴 수밖에 없다는 것.

이 구분이 이 장의 전부다.

그리고 앞 장들의 단서가 여기에도 붙는다. 이 청구서는 **통로가 좁은 동안만** 낼 가치가 있는 청구서이고, 좁은 동안에도 금액이 줄지는 않는다.

청구서

9장과 10장에서 열거한 것들이 사라지지 않았다.

분산 트랜잭션이 없다. 여러 서비스에 걸친 작업의 원자성을 데이터베이스가 보장해주지 않는다. 보상 트랜잭션을 직접 설계하거나 최종 일관성을 받아들여야 한다. 둘 다 어렵고, 둘 다 틀리기 쉽다.

지연이 생긴다. 함수 호출이 나노초였던 자리가 네트워크 왕복이 된다. 요청 하나가 다섯 서비스를 지나면 지연이 다섯 번 쌓인다.

디버깅이 흩어진다. 사고의 이야기가 여러 곳의 로그에 나뉜다. 4장에서 액터 모델이 겪은 것과 같은 종류의 문제이고, 분산 추적(distributed tracing) 도구들이 그것을 부분적으로만 해결했다.

> **분산 추적 (distributed tracing)**
>
> 요청 하나에 고유한 추적 ID를 붙이고, 그것이 여러 서비스를 지나갈 때마다 각 구간(span)을 기록해 하나의 타임라인으로 다시 꿰는 기법. 구글의 대퍼(Dapper, 2010) 논문이 계보의 시작이고, 지금은 OpenTelemetry가 사실상 표준이다.[1] **"부분적으로만"인 이유**는 추적이 어디를 지났고 얼마나 걸렸는지는 알려주지만 왜 그 값이 나왔는지는 알려주지 않기 때문이다 — 사고의 경로는 복원되고 사고의 이유는 복원되지 않는다.

운영이 곱해진다. 서비스가 서른 개면 배포와 모니터링과 온콜이 서른 벌이다. 쿠버네티스가 그 곱셈을 나눗셈으로 바꿨지만, 쿠버네티스 자체를 운영하는 일이 새로 생겼다.

그리고 이 청구서가 바로 11장의 되감기를 만들고 있는 힘이다.

데이터

가장 덜 풀린 반론이 하나 있다. 이 책은 그것을 여기 놓고 답하지 않는다.

마이크로서비스의 실질적 경계는 API가 아니라 **데이터베이스**였다는 관찰이다. 서비스마다 자기 데이터를 갖고, 남의 데이터를 직접 읽지 않는다. 그것을 지키지 않은 시스템에는 분산 모놀리스(distributed monolith)라는 이름이 붙었고, 대체로 실패로 취급되었다.

> **분산 모놀리스 (distributed monolith)**
>
> 서비스로 나눠 놓았지만 실질적으로는 하나처럼 움직이는 시스템. 전형적인 증상은 **공유 데이터베이스**다 — 서비스 A와 B가 같은 테이블을 직접 읽고 쓰면, 스키마를 바꿀 때 둘이 동시에 배포되어야 하고 하나가 죽으면 다른 하나도 못 돈다. 마이크로서비스의 세금은 전부 내면서 이득은 하나도 못 받는 상태라, 업계에서 가장 흔히 인용되는 안티패턴이 됐다.

데이터를 나누는 것이 진짜 경계였다면, 기계의 시대에 그것을 나눌 이유가 있는가.

인간의 시대에 데이터를 나눈 이유는 소유였다. 이 테이블을 누가 이해하고 있고 누가 스키마 변경에 책임을 지는가. 팀이 그 단위였다.

에이전트에게는 그런 소유가 없다. 그렇다면 데이터는 하나로 두고 코드만 나누면 되는 것 아닌가.

답하지 않는 이유

이 반론에 대해 이 책이 가진 것은 가설뿐이다.

가설은 이렇다. 데이터를 나눈 이유가 소유만은 아니었다. 이 상태의 불변식(invariant)을 **한 번에 판단할 수 있는가**도 이유였다. 스무 개 테이블이 서로

참조하는 상태에서 어떤 변경이 안전한지를 말하려면 스무 개의 관계를 동시에 들고 있어야 한다.

> **불변식 (invariant)**
>
> 시스템이 어떤 상태에 있든 항상 참이어야 하는 명제. "계좌 잔액은 음수가 될 수 없다", "주문의 합계는 항목 금액의 합과 같다", "발송된 주문은 취소될 수 없다" 같은 것들이다. **불변식은 경계를 정의한다** — 한 불변식을 지키기 위해 동시에 봐야 하는 데이터의 집합이 곧 한 트랜잭션의 범위이고, 대개 한 서비스의 범위다. 19장의 "닫힘"을 데이터 쪽에서 말하면 "이 안의 불변식은 이 안만 보고 지킬 수 있다"가 된다.

그것이 맞다면 데이터 경계도 인지부하의 문제이고, 같은 이유로 남는다.

그런데 이것은 증거로 세운 것이 아니다. 논리적으로 그럴듯할 뿐이다.

그래서 이 책은 여기에 표시를 남긴다. **가설.**

되감기가 말하는 것

11장의 되감기를 이 장의 관점에서 다시 보면 다르게 읽힌다.

되감고 있는 조직들은 마이크로서비스가 틀렸다는 것을 발견한 것이 아니다. **자기 규모에서 그 청구서가 이득보다 크다는 것**을 발견한 것이다.

팀이 다섯 개인 회사에서 서비스 마흔 개를 운영하면, 팀 자율성이라는 이득은 없고 운영 세금만 있다. 그 계산을 다시 하면 합치게 된다.

이것은 이 책의 논증과 충돌하지 않는다. 오히려 같은 구조다. **경계는 그것을 유지하는 비용을 감당할 수 있을 때만 존재한다.**

10장의 문장이 그것이었다. 마이크로서비스는 세금을 낼 수 있게 된 다음에 퍼졌다.

방향이 바뀌는 문장

그리고 여기서 그 문장의 방향이 바뀐다.

세금을 낼 수 있게 되어서 퍼졌다면, **세금을 낼 이유가 사라지면 사라진다.**

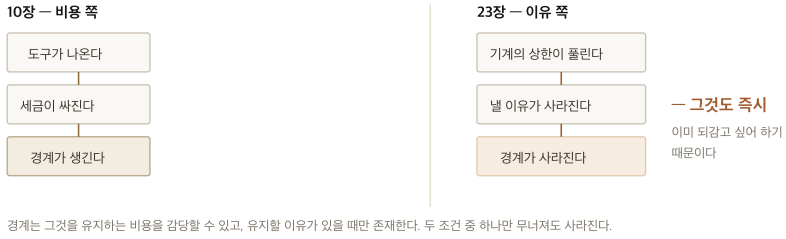


그림 20 — 같은 문장의 두 방향. 10장은 왼쪽, 이 장은 오른쪽을 읽는다.

지금 그 세금을 내야 하는 이유가 기계의 인지부하다. 그 이유가 없어지면, 조직들은 지금 되감고 있는 것보다 훨씬 빠르게 되감을 것이다.

왜냐하면 그들은 이미 되감고 싶어 하기 때문이다. 청구서가 비싸다는 것을 알고 있고, 지금은 다른 이유 때문에 참고 있을 뿐이다.

세금이 비싸다는 사실이 여기서 예측이 된다.

강제된 선택

이 책의 주장을 가장 정확하게 쓰면 이렇게 된다.

마이크로서비스는 지금 최적해가 아니다. **강제된 선택**이다.

최적해라면 제약이 풀려도 남는다. 강제된 선택은 제약이 풀리면 사라진다.

그리고 이 구분이 다음 장의 전제다.

남은 질문

지금 이 형태를 택하고 있는 조직들 중 얼마가, 그것이 강제된 선택이라는 것을 알고 있는가.

최적해라고 믿고 그 위에 조직과 프로세스와 커리어를 얹은 경우, 제약이 풀렸을 때 되돌아가는 비용은 누가 지불하는가.

주

1 Benjamin H. Sigelman et al., "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure", Google Technical Report dapper-2010-1, 2010-04. 〈https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/〉 · OpenTelemetry는 2019년 OpenTracing과 OpenCensus가 합쳐져 만들어진 CNCF 프로젝트다. 〈https://opentelemetry.io/〉 ↩

24장

3년

문장

이 책의 주장을 한 문단으로 줄이면 이렇다.

지금으로서는 마이크로서비스로 갈 수밖에 없다. AI가 한 번에 감당할 수 있는 인지부하가 유한하기 때문이다. 최대한 작은 닫힌 계를 만들고 그것을 블랙박스로 조립하는 것 외에 방법이 없다.

그리고 이 문장이 붙는다.

3년쯤 뒤에는 이것이 완전히 해소될 수도 있다. 지금은 그렇다는 것이다.

왜 이 문장이 필요한가

이 문장이 없으면 앞의 스무 장이 슬로건이 된다.

어떤 관측으로도 반박할 수 없는 주장은 주장이 아니다. 지금 이 형태가 옳다고만 말하면, 나중에 형태가 바뀌었을 때 그것은 다른 이유 때문이었다고 말하면 그만이다. 무엇을 봐도 이길 수 있는 논증은 아무것도 말하지 않은 것과 같다.

> **반증 가능성 (falsifiability)**
>
> 칼 포퍼(Karl Popper)가 1934년 『탐구의 논리』에서 세운 기준.[1] 어떤 주장이 과학적인지를 가르는 것은 그것을 **확증**할 증거가 있는지가 아니라, 그것을 **반박**할 수 있는 관측이 원리적으로 존재하는지다. 어떤 결과가 나와도 설명할 수 있는 이론은 강한 것이 아니라 공허한 것이다. 이 장이 만료 조건을 미리 못 박는 이유가 이것이다 — 무엇을 보면 이 책이 틀린 것인지를 저자가 먼저 정해 두지 않으면, 독자는 이 책을 검사할 수단이 없다.

시효를 붙이면 진다. 그리고 질 수 있어야 주장이다.

만료 조건

다음 중 하나라도 성립하면 이 책의 후반부는 끝난다.

첫째, 실효 인지부하의 상한이 시스템 전체를 덮는다.

16장의 숫자들이 뒤집히는 경우다. 광고된 컨텍스트가 아니라 실효 범위가, 아키텍처 판단 같은 복잡한 과제에서도 수십만 토큰으로 올라간다면. 그리고 그것이 RULER나 NoLiMa 같은 방식으로 독립적으로 측정된다면.

둘째, 탐색이 상주를 완전히 대체한다.

17장의 논문이 뒤집히는 경우다. 검색과 인덱싱과 컨텍스트 조립이 충분히 좋아져서, 경계 없는 대형 코드베이스에서도 에이전트 산출물의 결함률이 경계 있는 코드베이스와 통계적으로 같아진다면.

이 방향의 연구는 이미 활발하다. 컨텍스트 조립을 최적화 문제로 다루는 접근들이 벤치마크 점수를 크게 올렸다는 보고가 있다. 이 조건은 넷 중 가장 먼저 성립할 가능성이 있다.

셋째, 검증이 전역으로 가능해진다.

20장의 논리가 뒤집히는 경우다. 이 책은 검증이 경계 단위로만 가능하다고 전제했다. 형식 검증(formal verification)이나 전역 불변식 증명이 실용적인 규모로 가능해지면, 계약면이 없어도 옳음을 말할 수 있게 된다.

> **형식 검증 (formal verification)**
>
> 테스트로 몇몇 경우를 확인하는 대신, 프로그램이 명세를 **모든 입력에 대해** 만족한다는 것을 수학적으로 증명하는 방법. 실제로 쓰인 사례가 있다 — seL4 마이크로커널(약 1만 줄, 증명에 수십 인년), CompCert C 컴파일러, AWS의 분산 프로토콜 검증.[2] 문제는 규모다. 수십만 줄짜리 애플리케이션에 이것을 적용하는 것은 지금 실용적이지 않다. 이 조건이 성립한다는 것은 **그 비용이 무너진다**는 뜻이고, 그렇게 되면 "왜 나눠야 하는가"의 답 하나가 사라진다.

넷째, 통째 재생성이 점증 수정을 대체한다.

코드를 만드는 비용이 충분히 내려가서, 고치는 것보다 다시 만드는 것이 싸지는 경우다. 그러면 안정된 내부 경계를 유지할 이유가 없어진다. 매번 새로 뽑으면 되니까.

그림 21 — 만료 조건 넷과 각각이 무너뜨리는 장. 둘 이상이면 이 책은 철회된다.

다섯째, 편성의 이득이 부호를 바꾼다.

21장의 반론이 뒤집히지 않고 그대로 진행되는 경우다. 대상 과제에서 단일 에이전트 정확도가 안정적으로 45%를 넘고, 그 위에서 편성이 계속 지는 것이 재현되면, 나누기는 현재 능력 수준의 국소해였던 것으로 확정된다.

이 조건은 앞의 넷과 성질이 다르다. **바깥에서 이미 측정된 임계값이 있다.** "3년"이라는 근거 없는 어림 대신 관측 가능한 숫자가 하나 생긴 셈이고, 그래서 이 조건이 넷 중 가장 먼저 판정될 가능성이 높다.

몇 개가 성립해야 하는가

다섯 중 **둘 이상**이면 이 책은 철회된다.

하나만으로는 부족하다. 예를 들어 둘째만 성립하면, 탐색이 좋아져도 한 번에 판단해야 하는 순간의 부하는 남는다는 17장의 유보가 여전히 작동한다.

한 가지는 밝혀 둔다. 이 기준 — 다섯 중 둘 — 에는 외부 근거가 없다. 원래의 진단은 "3년쯤 뒤에는 이것이 완전히 해소될 수도 있다"는 한

문장이었고, 몇 개의 조건이 몇 개 성립해야 하는지는 거기 없었다. 기계적 기준을 붙인 것은 앞 절에서 말한 이유 때문이다. 숫자가 없으면 아무도 이 책을 검사할 수 없다.

왜 3년인가

근거가 없다.

이것을 그대로 쓴다.

3년이라는 숫자에 대해 이 책이 가진 것은 인상이다. 이 분야가 지난 3년 동안 움직인 폭을 보고 다음 3년을 어림한 것이다. 그것은 추정이지 예측이 아니다.

그러면 왜 숫자를 붙이는가.

숫자가 없으면 시효도 없기 때문이다. "언젠가는 달라질 것이다"는 아무 약속도 하지 않는 문장이다. 모든 주장이 언젠가는 달라진다.

3년은 정확한 예측이 아니라 **선언**이다. 이 글은 지금 시점의 진단이며, 지금 시점이 지나면 다시 봐야 한다는 선언.

근거 없는 숫자를 근거 있는 척하지 않는 것이, 이 책이 지킬 수 있는 정직함이다.

그리고 되돌아갈 것이다

23장 끝의 문장이 여기서 완성된다.

제약이 풀리면 조직들은 즉시 되돌아갈 것이다. 천천히가 아니라 즉시다.

이유는 그들이 이미 되감고 싶어 하기 때문이다. 청구서를 보고 있고, 그것이 비싸다는 것을 알고 있고, 지금은 다른 이유로 참고 있다.

11장에서 본 42%는 그 이유가 아직 오지 않은 조직들의 숫자다. 인지부하 때문에 서비스를 나눠야 한다는 압력을 아직 받지 않은 곳들. 그들에게 마이크로서비스는 순수하게 세금이었고, 그래서 되감고 있다.

무엇이 남는가

만료 조건이 전부 성립해도 남는 것이 있다.

파나스의 질문이다. 무엇이 함께 변하는가.

그 질문은 기계의 상한과 무관하다. 시스템에 어떤 선을 그을 것인가는, 그 선을 넘는 비용이 0이 되어도 여전히 질문이다. 다만 답이 달라진다.

이 책이 만료되는 것은 답이지 질문이 아니다.

남은 질문

3년 뒤에 이 책이 틀렸다는 것이 확인되면, 그것은 이 책의 실패인가 성공인가.

날짜를 붙인 진단이 그 날짜에 만료되는 것은 예정된 일이다. 예정된 일이 일어났을 때 무엇을 배우는가.

주

1 Karl Popper, *Logik der Forschung*, Julius Springer, 1934 (영역 *The Logic of Scientific Discovery*, Hutchinson, 1959). 반증 가능성을 과학과 비과학을 가르는 **구획 기준**(demarcation criterion)으로 제시했다. 이 기준 자체에 대한 비판도 오래됐다 — 특히 어떤 이론도 홀로 검사되지 않는다는 뒤앙-콰인 논제가 그렇다. 이 책은 포퍼를 과학철학의 결론으로 인용하지 않고, **저자가 스스로에게 부과하는 규율**로 쓴다. ↩

2 Gerwin Klein et al., "seL4: Formal Verification of an OS Kernel", *SOSP '09*, 2009. 〈https://sel4.systems/〉 · Xavier Leroy, "Formal verification of a realistic compiler", *CACM* 52(7), 2009. 〈https://compcert.org/〉 · Chris Newcombe et al., "How Amazon Web Services Uses Formal Methods", *CACM* 58(4), 2015. 〈https://dl.acm.org/doi/10.1145/2699417〉 ↩

에필로그

증명은 44년 뒤에 왔다

편집자

1967년에 하버드 비즈니스 리뷰의 어느 편집자가 원고를 하나 반려했다.

사유는 테제를 증명하지 못했다는 것이었다.

그 원고는 조직이 시스템을 설계할 때, 그 조직의 커뮤니케이션 구조를 복제한 설계를 내놓게 된다고 주장했다. 예시는 컴파일러와 군용 무기 체계였다.

편집자가 요구한 것은 데이터였다. 그리고 원고에는 데이터가 없었다.

그가 옳았다

이 책을 여기까지 읽고 다시 보면, 그 반려는 정확했다.

1967년의 그 원고에는 관찰과 예시와 설득력 있는 논증이 있었다. 그중 어느 것도 증명이 아니다. 세 개의 컴파일러를 보고 규칙을 말하는 것과, 그 규칙이 참임을 보이는 것은 다른 일이다.

편집자가 몰랐던 것은 하나뿐이다. 그 증명에 필요한 도구가 그 시점에 없었다는 것.

조직의 커뮤니케이션 구조를 측정 가능한 형태로 만들고, 코드의 결합도를 계량하고, 둘 사이의 관계를 통계적으로 다루는 일. 그것을 할 수 있게 되기까지 44년이 걸렸다.

그 사이

44년 동안 그 관찰은 증명되지 않은 채 쓰였다.

인용되었고, 조직 개편의 근거가 되었고, 나중에는 역으로 이용되었다. 원하는 아키텍처를 먼저 정하고 팀을 그 모양으로 나누는 방식에 이름이 붙었다.

업계는 검증되지 않은 관찰 위에 조직 설계를 얹었다. 그리고 그 관찰은 결국 맞는 것으로 밝혀졌다.

이 결말은 안심이 되기도 하고 불편하기도 하다. 맞았기 때문에 안심이 되고, **맞을지 모르는 상태로 40년을 썼기 때문에** 불편하다.

이 책이 서 있는 자리

이 책의 3부는 같은 종류의 주장을 한다.

인간 조직의 인지부하가 시스템의 형태를 정했다면, 기계의 인지부하도 그럴 것이라는 유비다.

그 유비를 뒷받침하는 것은 벤치마크 숫자들이다. 광고된 컨텍스트의 6분의 1까지 실효 범위가 떨어진 모델이 있다는 측정. 검색이 완벽해도 길이가 성능을 깎는다는 실험. 16만 줄짜리 프로젝트에서 네 에이전트가 전부 0을 기록했다는 결과.

이것들은 증거다. 그리고 증명이 아니다.

측정된 것은 모델이 긴 입력에서 나빠진다는 것이지, **그래서 시스템을 서비스로 나눠야 한다**는 것이 아니다. 그 사이에는 논증이 있고, 논증은 증명이 아니다.

콘웨이가 세 개의 컴파일러를 보고 규칙을 말했다면, 이 책은 다섯 개의 벤치마크를 보고 아키텍처를 말한다.

검증 기간 0

차이가 하나 있다.

콘웨이의 유비는 인간 조직 안에서 이루어졌다. 사람이 설계하고 사람이 소통하는 세계에서, 한 종류의 인간 활동에서 다른 종류의 인간 활동으로 옮긴 유비였다. 그것도 44년이 걸렸다.

이 책의 유비는 인간에서 기계로 건너간다. 작업 기억의 상한에서 어텐션 메커니즘의 성질로. 두 상한은 완전히 다른 것이고, 같은 것은 상한이 있다는 사실뿐이다.

그리고 그 유비에 대해 우리가 가진 검증 기간은 0이다.

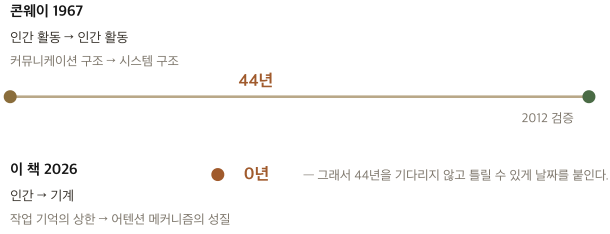


그림 22 — 두 유비의 검증 기간. 왼쪽은 44년이 걸렸고, 오른쪽은 아직 0년이다.

그래서 날짜를 붙인다

이 책이 할 수 있는 일은 예언하지 않는 것이다.

대신 날짜를 붙인다. 만료 조건 넷을 앞에 적어두고, 그중 둘이 성립하면 철회한다고 미리 말해둔다.

그것이 44년을 기다리지 않고 틀릴 수 있는 방법이다.

문장

2026년, 기계가 한 번에 감당할 수 있는 양은 우리가 생각하는 것보다 훨씬 작다.

그래서 우리는 다시 작게 쪼개고 있다. 62년 전에 벨 연구소의 메모 한 장이 제안한 것과 같은 모양으로. 54년 전에 파나스가 감추라고 한 것과 같은 방식으로. 53년 전에 휴잇이 그린 것과 같은 구조로.

전혀 다른 이유로.

남은 질문

리치의 벽에 붙어 있던 종이는 아홉 페이지를 잃고 살아남은 한 페이지였다.

거기 적힌 네 항목 중 셋은 해결되었고, 해결된 뒤에 잊혔다. 남은 하나는 해결되지 않았고, 매 세대마다 다시 물어졌다. 프로시저 호출로, 원격 객체로, 메시지 큐로, HTTP 요청으로, 그리고 지금은 에이전트 사이의 핸드오프로.

60년째 같은 질문이 돌아오고 있다면, 그것은 아직 답이 없다는 뜻인가.

아니면 이 질문이 원래 매번 다르게 답해야 하는 종류의 질문이고, 우리가 그 사실을 60년째 다시 배우고 있는 것인가.

부록 A

이 책이 틀리는 조건

이 책의 3부와 4부는 아래 조건 중 **둘 이상이 관측되면 철회된다.**

1. 실효 인지부하 상한이 시스템 전체를 덮는다. 광고된 컨텍스트가 아니라 실효 범위가, 아키텍처 판단 같은 관계 유지 과제에서도 수십만 토큰 규모로 올라간다. RULER나 NoLiMa 계열의 독립 측정으로 확인되어야 한다.

2. 탐색이 상주를 완전히 대체한다. 경계 없는 대형 코드베이스에서도 에이전트 산출물의 결함률이 경계 있는 코드베이스와 통계적으로 같아진다. 넷 중 가장 먼저 성립할 가능성이 있는 조건이다.

3. 검증이 전역으로 가능해진다. 형식 검증이나 전역 불변식 증명이 실용적 규모로 가능해져서, 계약면 없이도 옳음을 말할 수 있게 된다.

4. 통째 재생성이 점증 수정을 대체한다. 코드 생산 비용이 충분히 내려가 고치는 것보다 다시 만드는 것이 싸진다. 안정된 내부 경계를 유지할 이유가 사라진다.

5. 편성의 이득이 부호를 바꾼다. 단일 에이전트 정확도가 대상 과제에서 안정적으로 45%를 넘고, 그 위에서 편성이 계속 지는 것이 재현된다. 넷과 달리 **바깥에서 측정된 임계값이 이미 있는** 조건이다(21장).

"둘 이상"이라는 기준과 "3년"이라는 시효는 둘 다 저자가 정한 것이며 외부 근거가 없다. 24장에 그 사실을 적어두었다.

◆

연도	사건
1956	밀러(George A. Miller), 「마법의 숫자 7 플러스마이너스 2」
1958 / 1963	콘웨이(Melvin Conway), coroutine 고안 / 활자 최초 등장 (CACM 6(7))
1964-10-11	맥일로이(Doug McIlroy)의 벨 연구소 메모 — 정원 호스
1967	콘웨이의 원고, HBR 반려
1968-04	콘웨이, 「위원회는 어떻게 발명하는가」, Datamation 14(5):28–31
1972	리치의 회상 — "Pipes appeared in Unix in 1972" (2판 매뉴얼엔 없음)
1972-12	파나스(David Parnas), 「시스템을 모듈로 분해할 때 사용해야 할 기준」, CACM 15(12):1053–1058
1973-01	유닉스에 파이프 존재 확인
1973	휴잇(Carl Hewitt)·비숍·스타이거, 액터 형식론, IJCAI
1975	브룩스(Fred Brooks) 『맨먼스 미신』 — "콘웨이의 법칙" 명명
1985–86	아가(Gul Agha), 액터 형식화 (미시간대 박사논문 1985 · MIT Press 1986)
1988	스웰러(John Sweller), 「문제 해결 중의 인지부하」
1989	OMG 출범 (11개사)

연도	사건
1991-10	CORBA 1.0 — ORB 간 통신 프로토콜 부재
1994	도이치(Peter Deutsch), 분산 컴퓨팅의 오류 7개 (구전)
1996-08	CORBA 2.0 — IIOP
1997경	고슬링(James Gosling), 여덟 번째 오류 추가
2000	필딩(Roy Fielding), REST 박사논문 (UC Irvine)
2001	코완(Nelson Cowan), 「단기기억의 마법의 숫자 4」
2002경	아마존 서비스 전환 (1차 문서 없음)
2003	에번스(Eric Evans), 『도메인 주도 설계』 — 바운디드 컨텍스트
2005	코번(Alistair Cockburn), 헥사고날 아키텍처
2006	로템갈오즈(Arnon Rotem-Gal-Oz), 분산 컴퓨팅의 오류 최초 문서화
2006	헤닝(Michi Henning), 「CORBA의 흥망」, ACM Queue 4(5)
2006~2010	REST 실제 채택 구간
2008-10-20	필딩, "REST API는 하이퍼텍스트 주도여야 한다"
2010	프레스턴워너(Tom Preston-Werner), semver 제안
2011-05	베네치아 근교 워크숍 — 마이크로서비스 논의
2011	예게(Steve Yegge)의 구글 내부 메모 — 베조스 칙령 회상
2011-09	semver 1.0.0
2012-03	루이스(James Lewis), "Microservices — Java, the Unix Way" (크라쿠프)
2012-05	마이크로서비스 명명 확정
2012	맥코맥·볼드윈·러스낙, 미러링 가설 검증 (Research Policy 41(8):1309–1324, DOI 10.1016/j.respol.2012.04.011)

연도	사건
2013-03-15	도커 공개 (파이콘 US 라이트닝 토크, 솔로몬 하이크스)
2013-06	semver 2.0.0
2014-03-25	파울러·루이스, 마이크로서비스 아티클
2014-06-06	쿠버네티스 발표 (구글 블로그 · 최초 커밋) · 6-10 도커콘 무대 공개
2015-07	쿠버네티스 1.0 · CNCF 출범
2017	쇼피파이 컴포넌트화 프로젝트 시작
2018-07	세그먼트, 「마이크로서비스여 안녕」
2019-02-21	웨스테인데, 「모놀리스 해체하기」
2020-09-16	쇼피파이 후속편 — 웨지 폐기, 경계 검사는 Packwerk·진입점 계약은 소르베로
2023-03-22	프라임 비디오 VQA 사례
2023-07	「Lost in the Middle」 (arXiv:2307.03172 → TACL 12, 2024)
2024-04	RULER (arXiv:2404.06654, COLM 2024)
2025-02	NoLiMa (arXiv:2502.05167, ICML 2025)
2025-07-14	Context Rot (Chroma 기술 리포트)
2025-09-21	MECW (arXiv:2509.21361)
2025-10-06	「검색이 완벽해도 컨텍스트 길이만으로 성능이 저하된다」 (arXiv:2510.05381, Findings of EMNLP 2025)

연도	사건
2025-11-11	CNCF/SlashData 클라우드 네이티브 개발 현황 Q3 2025 발표
2026-02-26	RepoMod-Bench (arXiv:2602.22518)
2026-07-01	ChainSWE (arXiv:2607.02606)
2026	이 책

◆

이 책의 사실 주장에 붙는 등급이다. 각 항목의 출처는 해당 장의 각주에 있다.

[F] 1차 출처 확인

논문 원문, 저자 본인 사이트, 공식 발표를 직접 대조한 것.

역사 앵커. 파나스 1972 (CACM, DOI 10.1145/361598.361623) · 콘웨이 1968 (Datamation 14(5), 반려 경위는 저자 본인 사이트) · 맥일로이 메모 1964-10-11 (리치 소장본) · 헤닝 2006 (ACM Queue) · 필딩 2000·2008 · 파울러·루이스 2014 · 쇼피파이 2019·2020 엔지니어링 블로그 · 프라임 비디오 2023-03-22 · 맥코맥 외 2012 (Research Policy).

인지 상한. 밀러 1956 (Psychological Review 63(2)) · 코완 2001 (BBS 24(1)) · 스웰러 1988 (Cognitive Science 12(2)).

기계 상한 — 2026-08-22에 원문 대조 완료. 이 여섯 건은 이 책 3부의 하중이 걸린 인용이므로 별도로 대조했다.

인용	판정	비고
arXiv:2510.05381 (길이만으로 저하)	확정	13.9~85%, 절제 3종, 완화책 4% 모두 원문과 일치
RepoMod-Bench (arXiv:2602.22518)	확정	91.3→66.9→15.3% 곡선, 0%, 19.5%, "컨텍스트 윈도우 문제 아님" 문장 축자 확인
ChainSWE (arXiv:2607.02606)	확정 (귀속 정정)	70%는 *완화책 구성*의 상대 하락률. 기준 구성은 59%
RULER (arXiv:2404.06654)	확정 (요약 기각)	두 데이터포인트는 정확. **"광고의 절반~3분의 2"는 논문에 없다**
NoLiMa (arXiv:2502.05167)	확정	13개 중 11개는 v2/v3 기준. v1과 깃허브는 12개 중 10개
Context Rot (Chroma)	확정	구조적 일관성이 오히려 해롭다는 결과 원문 확인

[F-조건부] 조건을 명시해야만 성립

프라임 비디오 2023. 한 팀의 한 워크로드이며, 이동 축은 마이크로서비스→모놀리스가 아니라 서버리스 오케스트레이션→단일 프로세스다. 이 두 조건 없이 인용하면 허위가 된다. 11장은 이 사례를 되감기의 증거가 아니라 **밈이 사실을 이긴 사례**로 쓴다.

[전승] 1차 문서 없음 — 본문에서 그 사실을 함께 밝힌다

분산 컴퓨팅의 오류 여덟 개. 정식 출판 없음. 2006년에 최초 문서화. 저자 귀속이 소스마다 충돌한다. "Deutsch 1994 논문"이라는 표기는 오류다 — 그런 논문은 없다.

베조스 API 칙령. 1차 문서 없음. 유일 출처는 2011년의 회상이며 회상자 본인이 시점을 헤지했다.

[2차 인용, 검증 필요] 논증의 하중에 쓰지 않는다

"마이크로서비스 도입 조직의 42%가 통합 중" 및 후회율 수치. 재단·컨설팅사 보고서를 블로그가 2차 인용한 것이며 소스 간 숫자가 어긋난다. 방향은 참일 가능성이 높으나 크기는 확인되지 않았다.

"실효 컨텍스트는 광고의 50~65%" — RULER 논문에 없는 2차 요약이다. 이 책은 쓰지 않는다.

"복잡한 정렬·요약은 400~1,200 토큰에서 실패" — MECW 논문에 없는 2차 요약이다. 한 상용 블로그의 자체 일반화이며 그 블로그 자신도 "directional estimates"라고 밝히고 있다. 이 책은 논문이 산문으로 명시한 수치만 쓴다.

"Lost in the Middle에서 20~30 퍼센트포인트 하락, 6개 모델 계열" — 둘 다 2차 요약의 각색이다. 논문의 실제 표현은 한 모델에 대한 "20% 이상"이고, 주 실험은 모델 6종·계열 4종이다. 이 책은 이 현상을 수치 없이 서술한다.

[가설] 증거로 세우지 않았다

데이터 소유권 경계가 인지부하 때문에 남는다는 23장의 설명. 논리적으로 그럴듯할 뿐 측정된 바 없다.

"3년"이라는 시효의 구체적 숫자와 **"만료 조건 넷 중 둘"이라는 기준.** 둘 다 저자가 정했고 외부 근거가 없다.

외부 검증에서 나온 것 — 이 책이 저지른 오류의 유형

이 원고는 집필 뒤 장별로 외부 팩트체크를 받았고, 거기서 나온 정정을 본문에 반영했다. 정정 항목 자체는 각 장의 각주에 흡수되었으므로 여기 다시 열거하지 않는다. 남길 것은 **오류의 유형**이다.

찾아낸 오류의 대부분은 출처를 못 찾은 것이 아니었다. 출처를 확보한 뒤 본문으로 옮기는 과정에서 생긴 것이었고, 방향이 한결같았다.

최대치가 점값이 되었다. 논문의 "최대 17%까지 떨어졌다"가 본문에서 "17% 떨어졌다"가 되었다.

리더보드가 논문 표가 되었다. 벤치마크 저장소의 최신 수치를 논문의 표에서 나온 것처럼 적었다. 둘은 모델 구성이 다르다.

정황이 증거가 되었다. 저자들이 "그럴 것이다"라고 추측한 문단을, 이 책의 주장을 지지하는 자료로 썼다.

한 도구가 한 일이 다른 도구에 귀속되었다. 12장의 경계 검사는 Packwerk가 한 것인데 소르베가 한 것으로 적혀 있었다. 장 제목까지 그 오해 위에 있었다.

출처 없는 문장이 인용 부호를 얻었다. 12장에는 웨스테인데의 말로 제시된 문장이 하나 있었는데, 인용된 두 글 어디에도 없었다. 정리된 요지가 인용의 형식을 갖춰 버린 것이다.

본문이 자기 각주와 어긋났다. 16장은 "100만 토큰급 모델의 실효 범위는 측정된 바 없다"고 적었는데, 같은 장의 각주가 그 측정치를 인용하고 있었다.

이 목록이 여기 있는 이유가 있다. 6장과 8장이 지적한 것 — 규범이 옮겨 다니면서 정돈되고 강화되고 출처를 잃는 과정 — 은 남의 이야기가 아니었다. **같은 일이 이 원고를 쓰는 동안 일어났고, 외부 검증이 없었으면 그대로 인쇄되었을 것이다.**

특히 마지막 항목이 그렇다. 8장은 "정돈된 기억은 원래보다 깔끔하다"고 적는다. 이 책의 오류들도 전부 원본보다 깔끔한 쪽으로 어긋나 있었다.

이 책에 없는 것

저자 자신의 측정이 없다. 이 책의 3부는 전부 공개된 벤치마크에 기대고 있다. 여러 에이전트를 실제로 편성해 운영하면서 얻은 1차 데이터 — 같은 작업을 에이전트 하나에 줬을 때와 여럿에 나눴을 때의 결함 검출률 차이, 다중 엔진 리뷰에서 단일 엔진이 놓치는 결함의 비율, 서브에이전트 도구 호출 수 대비 산출물 품질 — 이 이 책에 한 줄도 들어가지 않았다.

그것이 이 책의 가장 큰 공백이다. 밝혀 두는 이유는 다음 판에서 그 자리를 채우기 위해서다.

◆

이 책은 아키텍처 통사가 아니다. 하나의 질문만 추적했다. 경계를 누가 그었고 무엇이 그것을 강제했는가.

그래서 중요하지만 그 질문에 직접 답하지 않는 계보들을 의도적으로 뺐다.

메인프레임과 시분할 · 프론트엔드와 모바일 아키텍처 · 스트리밍과 데이터 파이프라인 · 이벤트 소싱과 CQRS · 함수형 프로그래밍과 불변성 ·

데이터베이스 중심 아키텍처 (23장에서 반론으로 한 번만 등장) · 서버리스 (11장에서 프라임 비디오 사례의 배경으로만 등장).

이 목록을 밝혀두는 이유는, 빠졌다는 지적을 피하기 위해서가 아니라 **뺀 것이 판단이었다는 것**을 남기기 위해서다.

소프트웨어 아키텍처의 역사

경계를 그은 여섯 개의 손 — 빅히스토리, AI의 혼돈, 그리고 다음 형태

Zhuge Hyuk

1차본 · 2026