THE AICE LIBRARY

The Tower of Abstractions

หอคอยแห่งนามธรรม
สารานุกรมศาสตร์แห่งการคำนวณ

AN ENCYCLOPEDIA OF THE COMPUTING DISCIPLINES

★

Every discipline is a floor that hides the floor below and offers a cleaner surface to the floor above. Every chapter names its floor: what it hides, what it exposes, what leaks.

สำหรับผู้ใหญ่ที่ฉลาดพอจะอยากรู้ แต่ไม่เคยได้เรียนเรื่องนี้ในห้องเรียน

2026-08-01 · 15 FLOORS AND A ROOF

How this book is built

Fifteen self-contained chapters and a closing one, 900–1,400 words each: principles, not surveys. Every fact, date, name and link was verified on the web in the session that wrote it. Read in order to climb the tower, or open any floor on its own — each chapter's first line states where it stands in the stack, and its Connections section says which stairs lead where.

เล่มนี้เรียงแบบจับคู่ทีละย่อหน้า — ภาษาอังกฤษก่อน แล้วภาษาไทยย่อหน้าเดียวกันตามใต้เส้นข้าง ทันที อ่านภาษาเดียวรวดเดียวก็ได้ หรือจะเทียบสองภาษาโดยไม่ต้องพลิกหน้าก็ได้ ฉบับอังกฤษคือ canon ต้นฉบับสำหรับทีมและสำหรับป้อนเข้าดัชนีของ Logy ส่วนฉบับไทยคือฉบับเรียบเรียงสำหรับอ่าน ศัพท์เทคนิคคงเป็นภาษาอังกฤษไว้ตามธรรมเนียมของบ้าน เพราะนั่นคือคำที่ผู้อ่านจะเจอต่อไปในเอกสารจริง

```
SSOT · books/computing/tower-of-abstractions/source/en/ (16 files)
TH contract · source/TH_SPEC.md
rebuild · core/_kit/venv/bin/python source/build_book.py
```

Contents

สารบัญ

CHAPTER 00	**The Map** แผนที่	5
CHAPTER 01	**Theory of Computation** ทฤษฎีการคำนวณ	14
CHAPTER 02	**Algorithms & Data Structures** อัลกอริทึมและโครงสร้างข้อมูล	24
CHAPTER 03	**Information & Coding** สารสนเทศและการเข้ารหัส	33
CHAPTER 04	**Computer Architecture** สถาปัตยกรรมคอมพิวเตอร์	42
CHAPTER 05	**Operating Systems** ระบบปฏิบัติการ	51
CHAPTER 06	**Languages & Compilers** ภาษาและคอมไพเลอร์	60
CHAPTER 07	**Software Engineering** วิศวกรรมซอฟต์แวร์	69
CHAPTER 08	**Databases** ฐานข้อมูล	78
CHAPTER 09	**Networks** เครือข่าย	87
CHAPTER 10	**Distributed Systems** ระบบกระจาย	96
CHAPTER 11	**Security & Cryptography** ความปลอดภัยและวิทยาการรหัสลับ	105

CHAPTER 12	**AI & Machine Learning** ปัญญาประดิษฐ์และการเรียนรู้ของเครื่อง	114
CHAPTER 13	**HCI & Graphics** ปฏิสัมพันธ์มนุษย์–คอมพิวเตอร์ และกราฟิกส์	123
CHAPTER 14	**The Frontier** พรมแดน	132
CLOSING	**Closing — The Tower, Seen Whole** ปิดเล่ม — หอคอยเมื่อมองทั้งหลัง	141

CHAPTER 00

The Map

แผนที่

This chapter is the ground floor of the tower — the map every chapter above it assumes you have seen.

บทนี้คือชั้นล่างสุดของหอคอย — แผนที่ที่ทุกบทข้างบนสันนิษฐานไว้ว่าคุณเคยเห็นมาก่อน

Why is "computer science" neither about computers nor a science — and what are the disciplines, really?

ทำไม "วิทยาการคอมพิวเตอร์ (computer science)" จึงไม่ใช่ทั้งเรื่องคอมพิวเตอร์และไม่ใช่วิทยาศาสตร์ — แล้วสาขาวิชาเหล่านี้คืออะไรกันแน่

In September 1967, three men wrote a letter to the journal *Science* because colleagues kept telling them their field did not exist. Allen Newell, Alan Perlis, and Herbert Simon were not obscure cranks defending a fad: Perlis had received the first Turing Award ever given, the year before; Newell and Simon would share one in 1975; Simon would add a Nobel Prize in economics in 1978. Their defense fit in a sentence — computer science is "the study of computers and all the phenomena surrounding them" (Newell, Perlis & Simon, 1967).

> ในเดือนกันยายน ปี 1967 ชายสามคนเขียนจดหมายถึงวารสาร Science เพราะเพื่อนร่วมวงการพูดกรอกหูซ้ำแล้วซ้ำเล่าว่าสาขาวิชาของพวกเขาไม่มีอยู่จริง Allen Newell, Alan Perlis และ Herbert Simon ไม่ใช่พวกคลั่งไคล้ไร้ชื่อเสียงที่ออกมาปกป้องกระแสชั่วครู่: Perlis เพิ่งได้รับ Turing Award รางวัลแรกที่เคยมอบให้ใครเมื่อปีก่อนหน้านั้น; Newell กับ Simon จะได้รับรางวัลร่วมกันในปี 1975; ส่วน Simon จะได้รางวัลโนเบลสาขาเศรษฐศาสตร์เพิ่มอีกในปี 1978 คำแก้ต่างของพวกเขาอยู่ใน

> ประโยคเดียว — วิทยาการคอมพิวเตอร์ (computer science) คือ "การศึกษาคอมพิวเตอร์และปรากฏการณ์ทั้งหมดที่อยู่รอบตัวมัน" ("the study of computers and all the phenomena surrounding them") (Newell, Perlis & Simon, 1967)

Eighteen years later, the most famous textbook the field ever produced opened by disagreeing with its founders. Abelson and Sussman's *Structure and Interpretation of Computer Programs* declares in its preface a conviction that "computer science" is not a science, and that its significance has little to do with computers (SICP, 1985). A sharper version of the same jab circulates to this day — *computer science is no more about computers than astronomy is about telescopes* — usually pinned on Edsger Dijkstra, though the traceable source is Michael Fellows and Ian Parberry in 1993 (Quote Investigator).

> สิบแปดปีให้หลัง ตำราที่โด่งดังที่สุดเท่าที่สาขานี้เคยผลิตออกมา เปิดเรื่องด้วยการเห็นต่างจากผู้ก่อตั้งสาขาของตัวเอง Structure and Interpretation of Computer Programs ของ Abelson และ Sussman ประกาศไว้ในคำนำว่าพวกเขาเชื่อว่า "วิทยาการคอมพิวเตอร์" ไม่ใช่วิทยาศาสตร์ และความสำคัญของมันแทบไม่เกี่ยวอะไรกับคอมพิวเตอร์เลย (SICP, 1985) คำแดกดันแบบเดียวกันแต่คมกว่าเดิมยังหมุนเวียนอยู่จนถึงทุกวันนี้ — วิทยาการคอมพิวเตอร์ไม่ได้เกี่ยวกับคอมพิวเตอร์ มากไปกว่าที่ดาราศาสตร์เกี่ยวกับกล้องโทรทรรศน์ (computer science is no more about computers than astronomy is about telescopes) — มักถูกยกให้เป็นคำพูดของ Edsger Dijkstra แต่ต้นตอที่สืบย้อนได้จริงคือ Michael Fellows และ Ian Parberry ในปี 1993 (Quote Investigator)

So the founders called it a science of computers, and the field's own catechism says it is neither. Both are right, and seeing how both can be right is the work of this chapter — and the organizing idea of this book.

> ผู้ก่อตั้งสาขาเรียกมันว่าวิทยาศาสตร์ของคอมพิวเตอร์ ส่วนคำสอนของสาขาเองกลับบอกว่ามันไม่ใช่ทั้งคู่ ทั้งสองฝ่ายพูดถูก และการมองให้เห็นว่าทั้งสองฝ่ายถูกได้พร้อมกันอย่างไร คืองานของบทนี้ — และคือแนวคิดที่จัดวางทั้งเล่มนี้ไว้

The first thing to understand is that "computing" is not one discipline wearing five name tags. Each discipline is a different *question* asked about the same machines. The official map was drawn by the professional computing societies: the CC2005 report (ACM, AIS, and the IEEE Computer Society) fixed the five classical disciplines, and its successor CC2020 — a fifty-member task force from twenty countries, reporting at the end of 2020 — inherited them and welcomed two arrivals. The five ask:

> สิ่งแรกที่ต้องเข้าใจคือ "การคำนวณ (computing)" ไม่ใช่สาขาวิชาเดียวที่สวมป้ายชื่อห้าใบ แต่ละสาขาคือ คำถาม ที่ต่างกันซึ่งถูกถามกับเครื่องจักรชุดเดียวกัน แผนที่ที่เป็นทางการถูกวาดขึ้นโดยสมาคมวิชาชีพด้านคอมพิวเตอร์: รายงาน CC2005 (ACM, AIS และ IEEE Computer Society) กำหนดสาขาดั้งเดิมทั้งห้าไว้ และรายงานรุ่นต่อมาอย่าง CC2020 — คณะทำงานห้าสิบคนจากยี่สิบประเทศ ซึ่งรายงานผลเมื่อปลายปี 2020 — รับช่วงสาขาเดิมทั้งห้ามาต่อ พร้อมต้อนรับผู้มาใหม่อีกสอง ห้าสาขานั้นถามว่า:

Computer science (CS): what can be computed at all, and at what cost? **Computer engineering (CE):** how does electricity become logic — and where exactly does hardware end and software begin? **Software engineering (SE):** what happens to a program when it must live for years and be changed by many hands? **Information systems (IS):** what does computing *do* inside an organization — to its decisions, records, and power? **Information technology (IT):** who keeps real systems running for real users, and how? The arrivals are **cybersecurity** (curricular guidelines: CSEC2017) and **data science** (ACM competencies, 2021) — younger, and, as we will see, shaped differently from the elder five.

> **วิทยาการคอมพิวเตอร์ (computer science, CS):** อะไรบ้างที่คำนวณได้ และด้วยต้นทุนเท่าไร? **วิศวกรรมคอมพิวเตอร์ (CE):** ไฟฟ้ากลายเป็นตรรกะได้อย่างไร — และฮาร์ดแวร์จบตรงไหน ซอฟต์แวร์เริ่มตรงไหนกันแน่? **วิศวกรรมซอฟต์แวร์ (SE):** เกิดอะไรขึ้นกับโปรแกรมเมื่อมันต้องอยู่ต่อไปหลายปีและถูกแก้ไขโดยคนจำนวนมาก? **ระบบสารสนเทศ (IS):** การคำนวณ ทำ อะไรบ้างภายในองค์กร — ต่อการตัดสินใจ ต่อบันทึกข้อมูล และต่ออำนาจ? **เทคโนโลยีสารสนเทศ (IT):** ใครคือคนที่ทำให้ระบบจริงยังทำงานอยู่เพื่อผู้ใช้จริง และทำอย่างไร? ผู้มาใหม่สองรายคือ **ความ**

> **มั่นคงปลอดภัยไซเบอร์ (cybersecurity)** (ข้อแนะนำหลักสูตร: CSEC2017) และ **วิทยาการข้อมูล (data science)** (ACM competencies, 2021) — อายุน้อยกว่า และดังที่จะเห็นต่อไป มีรูปทรงต่างจากห้าสาขาดั้งเดิม

Watch one artifact — a hospital's patient-records system — and the five become five professions. The computer scientist asks whether the record-matching algorithm scales to ten million patients. The computer engineer asks why the storage controller corrupts writes during power loss. The software engineer asks how to change the billing module without breaking the pharmacy's. The IS specialist asks why nurses route around the system with paper notes. The IT professional is the one paged at 3 a.m. when it goes down. Same machine; five questions.

> จับตาดูสิ่งประดิษฐ์ชิ้นเดียว — ระบบเวชระเบียนผู้ป่วยของโรงพยาบาล — แล้วห้าสาขาจะกลายเป็นห้าอาชีพ นักวิทยาการคอมพิวเตอร์ถามว่าอัลกอริทึมจับคู่เวชระเบียนจะขยายรองรับผู้ป่วยสิบล้านคนได้ไหม วิศวกรคอมพิวเตอร์ถามว่าทำไมตัวควบคุมที่เก็บข้อมูลถึงทำให้การเขียนข้อมูลเสียหายเมื่อไฟดับ วิศวกรซอฟต์แวร์ถามว่าจะแก้มอดูลเรียกเก็บเงินอย่างไรโดยไม่ทำให้ระบบของฝ่ายเภสัชกรรมพัง ผู้เชี่ยวชาญด้านระบบสารสนเทศถามว่าทำไมพยาบาลถึงหลบเลี่ยงระบบด้วยการจดใส่กระดาษแทน ส่วนผู้เชี่ยวชาญด้านเทคโนโลยีสารสนเทศคือคนที่ถูกตามตัวตอนตีสามเมื่อระบบล่ม เครื่องเดียวกัน ห้าคำถาม

This book does not walk that map department by department. It makes a different cut — the cut the field itself quietly makes every day. Computing's one great trick, the trick underneath every discipline, is **abstraction**: build a floor, hide its machinery, and offer the floor above a surface simpler than what it conceals. Electric charge is hidden by the transistor; transistors by logic gates; gates by the processor; the processor by the operating system; the machine's bare instructions by programming languages; single machines by networks; and on upward until a child drags a photo across a screen, commanding billions of switches with one finger and needing to understand none of them. Nobody holds the whole tower in mind at once. *That is not a failure of education — it is the design.* Dijkstra, in his 1972 Turing Award lecture, put his finger on why the trick works: the purpose of ab-

stracting is not to be vague, but "to create a new semantic level in which one can be absolutely precise" (EWD340). Each floor is not a blurrier view of the one below; it is a *new language*, exact on its own terms.

> หนังสือเล่มนี้ไม่ได้เดินตามแผนที่นั้นทีละแผนก แต่ตัดมุมมองอื่นออกมาแทน — มุมที่สาขานี้เองตัดอย่างเงียบๆ อยู่ทุกวัน กลเม็ดใหญ่หนึ่งเดียวของการคำนวณ กลเม็ดที่อยู่ใต้ทุกสาขาวิชา คือ **นามธรรม (abstraction)**: สร้างชั้นหนึ่งขึ้นมา ซ่อนกลไกของมันไว้ แล้วยื่นผิว (surface) ที่เรียบง่ายกว่าสิ่งที่มันปิดบังไว้ให้ชั้นบน ประจุไฟฟ้าถูกซ่อนไว้โดยทรานซิสเตอร์ (transistor); ทรานซิสเตอร์ถูกซ่อนโดยเกตตรรกะ (logic gate); เกตถูกซ่อนโดยตัวประมวลผล (processor); ตัวประมวลผลถูกซ่อนโดยระบบปฏิบัติการ (operating system); คำสั่งดิบของเครื่องถูกซ่อนโดยภาษาโปรแกรม (programming language); เครื่องเดี่ยวถูกซ่อนโดยเครือข่าย (network); และไล่ขึ้นไปเรื่อยๆ จนกระทั่งเด็กคนหนึ่งลากรูปถ่ายข้ามหน้าจอ สั่งสวิตช์นับพันล้านตัวด้วยนิ้วเดียว โดยไม่ต้องเข้าใจสวิตช์เหล่านั้นแม้แต่ตัวเดียว ไม่มีใครถือทั้งหอคอยไว้ในหัวพร้อมกันทั้งหมด นั่นไม่ใช่ความล้มเหลวของการศึกษา — มันคือการออกแบบ Dijkstra ในปาฐกถา Turing Award ปี 1972 ของเขา ชี้ตรงจุดว่าทำไมกลเม็ดนี้ถึงได้ผล: จุดประสงค์ของการทำ abstraction ไม่ใช่เพื่อให้คลุมเครือ แต่เพื่อ "สร้างระดับความหมายใหม่ขึ้นมาระดับหนึ่ง ที่ซึ่งเราสามารถแม่นยำได้อย่างสมบูรณ์" ("to create a new semantic level in which one can be absolutely precise") (EWD340) แต่ละชั้นไม่ใช่มุมมองที่พร่ามัวกว่าของชั้นล่าง มันคือ ภาษาใหม่ ที่แม่นยำในเงื่อนไขของตัวมันเอง

The proof that the tower is climbable is that students climb it. In Nisan and Schocken's course-in-a-book *The Elements of Computing Systems* (nand2tetris.org), learners start with a single logic gate and build — gates, chips, a machine, an assembler, a compiler, an operating system, a game — in one term. One person can build the whole stack only because each floor needs nothing from the floor below except its surface.

> หลักฐานที่ว่าหอคอยนี้ปีนได้จริง คือการที่นักเรียนปีนมันจริงๆ ในตำรา-คอร์สเรียนเล่มเดียวของ Nisan และ Schocken ชื่อ The Elements of Computing Systems (nand2tetris.org) ผู้เรียนเริ่มจาก logic gate ตัวเดียว แล้วสร้างขึ้นไป — เกต ชิป

> เครื่องจักร แอสเซมเบลอร์ (assembler) คอมไพเลอร์ ระบบปฏิบัติการ และเกมหนึ่งเกม — ภายในหนึ่งเทอม คนคนเดียวสร้างทั้งชั้นซ้อน (stack) ได้ทั้งหมดก็เพราะแต่ละชั้นไม่ต้องการอะไรจากชั้นล่างเลย นอกจากผิวของมัน

But the surfaces are not perfect, and this is the second idea the whole book leans on. Joel Spolsky named it the Law of Leaky Abstractions: "all non-trivial abstractions, to some degree, are leaky" (Spolsky, 2002). Your app promises that a message "was sent," until the network below it disagrees. Two loops, logically identical, run ten times apart in speed because of how memory hardware three floors down caches data. When an abstraction leaks, someone must descend the stairs — and knowing *which* floor the leak comes from is most of what expertise in computing is. That is why every chapter in this book answers three questions about its floor: what it hides, what surface it offers, and where it leaks.

> แต่ผิวเหล่านั้นไม่ได้สมบูรณ์แบบ และนี่คือไอเดียที่สองที่ทั้งเล่มพิงอยู่ Joel Spolsky ตั้งชื่อมันว่ากฎว่าด้วย abstraction ที่รั่วเสมอ (the Law of Leaky Abstractions): "abstraction ที่ไม่ธรรมดาทุกตัว รั่วไม่มากก็น้อย" ("all non-trivial abstractions, to some degree, are leaky") (Spolsky, 2002) แอปของคุณสัญญาว่าข้อความ "ถูกส่งแล้ว" จนกว่าเครือข่ายชั้นล่างจะไม่เห็นด้วย ลูปสองลูปที่เหมือนกันทุกประการในทางตรรกะ กลับทำงานเร็วช้าต่างกันสิบเท่า เพราะวิธีที่ฮาร์ดแวร์หน่วยความจำสามชั้นข้างล่างแคช (cache) ข้อมูลไว้ เมื่อ abstraction รั่ว ใครสักคนต้องลงบันไดไปดู — และการรู้ว่าการรั่วนั้นมาจากชั้นไหนกันแน่ คือส่วนใหญ่ของสิ่งที่เรียกว่าความเชี่ยวชาญในการคำนวณ นั่นคือเหตุผลที่ทุกบทในเล่มนี้ตอบคำถามสามข้อเกี่ยวกับชั้นของตัวเอง: ชั้นนี้ซ่อนอะไร ยื่นผิวแบบไหนให้ชั้นบน และรั่วตรงไหน

Now the two 1967-versus-1985 claims resolve. Newell, Perlis, and Simon were right: the phenomena are real, and floors like architecture, networks, and machine learning are studied with experiments, like nature. Abelson and Sussman were right too: no floor above the bottom two is about electronics at all — the upper tower is made of pure structure, closer to math-

ematics than to any machine. Computing is not one kind of knowledge. It is a tower in which physics hardens into engineering, engineering into language, and language into something very much like thought.

> ตอนนี้ข้อถกเถียงทั้งสองจากปี 1967 กับปี 1985 คลี่คลายลงได้ Newell, Perlis และ Simon พูดถูก: ปรากฏการณ์เหล่านั้นมีอยู่จริง และชั้นอย่างสถาปัตยกรรม (architecture) เครือข่าย และ machine learning ก็ถูกศึกษาด้วยการทดลอง เหมือนศึกษาธรรมชาติ Abelson กับ Sussman ก็พูดถูกเช่นกัน: ไม่มีชั้นไหนที่อยู่เหนือสองชั้นล่างสุดที่เกี่ยวกับอิเล็กทรอนิกส์เลยแม้แต่น้อย — หอคอยส่วนบนสร้างขึ้นจากโครงสร้างล้วนๆ ใกล้เคียงคณิตศาสตร์มากกว่าเครื่องจักรใดๆ การคำนวณไม่ใช่ความรู้ชนิดเดียว มันคือหอคอยที่ฟิสิกส์แข็งตัวกลายเป็นวิศวกรรม วิศวกรรมกลายเป็นภาษา และภาษากลายเป็นสิ่งที่ใกล้เคียงกับความคิดอย่างมาก

How to read this book: there are fifteen floors and a closing chapter, each self-contained, each 900–1,400 words — principles, not surveys. Chapters 01–03 are bedrock (what is computable, what is efficient, what is information); 04–06 build the machine; 07–10 take it to scale; 11–14 walk the edges, where adversaries, learned behavior, and human beings enter. Read in order to climb the tower, or jump to any floor — every chapter names its place in the stack in its first line, and its Connections section tells you which stairs lead where.

> วิธีอ่านเล่มนี้: มีสิบห้าชั้นและบทปิดเล่มอีกหนึ่งบท แต่ละบทจบในตัวเอง แต่ละบทยาว 900–1,400 คำ — เป็นหลักการ ไม่ใช่การสำรวจภาพรวม บทที่ 01–03 คือชั้นรากฐาน (อะไรคำนวณได้ อะไรมีประสิทธิภาพ อะไรคือสารสนเทศ); 04–06 สร้างตัวเครื่องขึ้นมา; 07–10 พาไปสู่ระดับสเกลใหญ่; 11–14 เดินไปตามขอบ ที่ซึ่งฝ่ายตรงข้าม พฤติกรรมที่เรียนรู้ได้ และมนุษย์เข้ามาเกี่ยวข้อง อ่านตามลำดับถ้าอยากปีนหอคอยขึ้นไป หรือจะกระโดดไปชั้นไหนก็ได้ — ทุกบทบอกตำแหน่งของตัวเองในชั้นซ้อนไว้ในบรรทัดแรก และหัวข้อ "เชื่อมโยง" ของบทนั้นจะบอกว่าบันไดไหนพาไปที่ไหน

Connections · เชื่อมโยง

Below this chapter: nothing — this is the ground. Every other chapter stands on it. The disciplines map onto the floors unevenly, and that is worth noticing as you read: CS saturates chapters 01–03 and 12, CE owns 04, SE owns 07, IS and IT surface in 08–10, and the two young arrivals — security and data science — turn out not to be floors at all but stairwells, cutting vertically through everything (11 and 12 explain why).

> ชั้นล่างของบทนี้: ไม่มี — นี่คือพื้นดิน ทุกบทอื่นยืนอยู่บนมัน สาขาวิชาต่างๆ ทาบทับกับชั้นต่างๆ ไม่เท่ากัน และเป็นเรื่องที่ควรสังเกตขณะอ่าน: CS ครอบคลุมเต็มบทที่ 01–03 และ 12, CE เป็นเจ้าของบทที่ 04, SE เป็นเจ้าของบทที่ 07, IS กับ IT โผล่ขึ้นมาในบทที่ 08–10 ส่วนผู้มาใหม่สองรายที่อายุน้อยกว่า — security กับ data science — กลับไม่ใช่ชั้นเลยสักชั้น แต่เป็นบันไดที่ตัดขึ้นลงในแนวดิ่งผ่านทุกอย่าง (บทที่ 11 และ 12 อธิบายว่าทำไม)

Open questions · คำถามที่ยังเปิดอยู่

Is computing a science, a branch of engineering, or a new kind of mathematics? The 1967 letter did not settle it, and sixty years on the field still funds all three answers. Will data science and cybersecurity harden into disciplines with their own floors, or dissolve into competencies every computing professional needs — the reading CC2020's competency-first framing quietly favors? And is the tower finished growing? Chapter 12 raises the live possibility that natural language itself is becoming the newest floor — a surface under which everything in this book is hidden.

> การคำนวณคือวิทยาศาสตร์ แขนงหนึ่งของวิศวกรรม หรือคณิตศาสตร์แบบใหม่กันแน่? จดหมายปี 1967 ไม่ได้ฟันธงเรื่องนี้ และหกสิบปีผ่านไป สาขาวิชานี้ก็ยังให้ทุนกับคำตอบทั้งสามแบบอยู่ data science กับ cybersecurity จะแข็งตัวกลายเป็นสาขาวิชาที่มีชั้นของตัวเอง หรือจะละลายไปเป็นสมรรถนะ (competency) ที่มืออาชีพด้านการคำนวณทุกคนต้องมี — ซึ่งเป็นการอ่านที่กรอบคิดแบบ competency-

> first ของ CC2020 เอียงเอนไปทางนั้นอย่างเงียบๆ? แล้วหอคอยนี้หยุดเติบโตแล้วหรือยัง? บทที่ 12 หยิบยกความเป็นไปได้ที่ยังมีชีวิตอยู่ว่าภาษาธรรมชาติเองกำลังจะกลายเป็นชั้นใหม่ล่าสุด — เป็นผิวที่ซึ่งทุกอย่างในเล่มนี้ถูกซ่อนอยู่ข้างใต้

Go deeper · อ่านต่อ

- Charles Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2nd ed., Microsoft Press, 2022 — the lower tower, built floor by floor for a general reader; interactive companion at codehiddenlanguage.com.

  > Charles Petzold, Code: The Hidden Language of Computer Hardware and Software, 2nd ed., Microsoft Press, 2022 — หอคอยชั้นล่าง สร้างขึ้นทีละชั้นสำหรับผู้อ่านทั่วไป; มีคู่มือโต้ตอบได้ที่ codehiddenlanguage.com

- Allen Newell, Alan Perlis & Herbert Simon, "Computer Science," *Science* 157, no. 3795 (1967): 1373–1374 — science.org.

  > Allen Newell, Alan Perlis & Herbert Simon, "Computer Science," Science 157, no. 3795 (1967): 1373–1374 — science.org

- ACM & IEEE Computer Society, *Computing Curricula 2020: Paradigms for Global Computing Education* — full report PDF.

  > ACM & IEEE Computer Society, Computing Curricula 2020: Paradigms for Global Computing Education — รายงานฉบับเต็ม PDF

CHAPTER 01

Theory of Computation

ทฤษฎีการคำนวณ

This chapter is the bedrock of the tower — the floor that decides what every floor above it is permitted to attempt.

บทนี้คือชั้นรากฐานของหอคอย — ชั้นที่กำหนดว่าทุกชั้นข้างบนได้รับอนุญาตให้ลองทำอะไรได้บ้าง

What can be computed at all — and what never, on any hardware?

อะไรบ้างที่คำนวณได้เลย — และอะไรที่คำนวณไม่ได้เด็ดขาด ไม่ว่าจะบนฮาร์ดแวร์ชนิดไหนก็ตาม

Here is a tool every working programmer would like to own. It reads a program and its input and answers one question: will this finish, or will it run forever? No more hung builds, no more servers looping overnight. It need not even *fix* the program, only report on it.

> นี่คือเครื่องมือที่โปรแกรมเมอร์ทำงานจริงทุกคนอยากมีไว้ในมือ มันอ่านโปรแกรมกับข้อมูลนำเข้าของโปรแกรมนั้น แล้วตอบคำถามเดียว: โปรแกรมนี้จะจบ หรือจะวนซ้ำไปตลอดกาล? ไม่ต้องมี build ที่ค้างอีกต่อไป ไม่ต้องมีเซิร์ฟเวอร์วนลูปข้ามคืนอีกต่อไป มันไม่จำเป็นต้อง แก้ โปรแกรมด้วยซ้ำ แค่รายงานผลก็พอ

It cannot exist. Not "nobody has managed it yet," not "it would be too slow" — no such program can be written, on any machine ever built or ever to be built, and the proof was published in 1936, before there was a machine to run it on. Why is the whole of this floor, and the reason the floor sits at the bottom of the tower: it fences the territory everything above builds on.

> มันไม่มีทางมีอยู่จริง ไม่ใช่ "ยังไม่มีใครทำสำเร็จ" ไม่ใช่ "มันคงช้าเกินไป" — ไม่มีโปรแกรมแบบนี้ที่เขียนขึ้นได้ ไม่ว่าจะบนเครื่องไหนที่เคยสร้างมาแล้วหรือจะสร้างขึ้นในอนาคต และบทพิสูจน์ถูกตีพิมพ์ตั้งแต่ปี 1936 ก่อนจะมีเครื่องจักรสักเครื่องให้รันมันด้วยซ้ำ "ทำไม" คือเนื้อหาทั้งหมดของชั้นนี้ และคือเหตุผลที่ชั้นนี้อยู่ล่างสุดของหอคอย: มันกั้นเขตแดนที่ทุกอย่างข้างบนสร้างต่อยอดขึ้นมา

The question came from Hilbert, and was meant to have a happy answer. In 1928 David Hilbert and Wilhelm Ackermann published *Grundzüge der theoretischen Logik*, the first textbook exposition of first-order logic, and posed in it the *Entscheidungsproblem*: is there a definite procedure that takes any statement of logic and decides whether it follows from the axioms (Springer)? Hilbert expected there was. In Königsberg on 8 September 1930 he closed a speech with the line later carved on his gravestone: *Wir müssen wissen — wir werden wissen* — we must know, we will know (MAA Convergence).

> คำถามนี้มาจาก Hilbert และตั้งใจให้มีคำตอบที่มีความสุข ปี 1928 David Hilbert และ Wilhelm Ackermann ตีพิมพ์ Grundzüge der theoretischen Logik ตำราเล่มแรกที่อธิบายตรรกศาสตร์อันดับหนึ่ง (first-order logic) อย่างเป็นระบบ และตั้งคำถาม Entscheidungsproblem ไว้ในนั้น: มีกระบวนการที่แน่นอนซึ่งรับข้อความเชิงตรรกะใดๆ แล้วตัดสินได้ไหมว่ามันสืบเนื่องมาจากสัจพจน์ (axiom) หรือไม่ (Springer)? Hilbert คาดว่ามี ที่เมือง Königsberg วันที่ 8 กันยายน 1930 เขาปิดปาฐกถาด้วยประโยคที่ภายหลังถูกสลักไว้บนหลุมศพของเขา: Wir müssen wissen — wir werden wissen — เราต้องรู้ เราจะต้องรู้ (MAA Convergence)

The day before, at a roundtable closing a conference in the same city, a young logician named Kurt Gödel had mentioned almost in passing that any consistent formal system strong enough to describe arithmetic must con-

tain true statements it cannot prove. The room let it go by; John von Neumann, present, appears to have been the only person who grasped what had happened (Davis, *Notices of the AMS*, 2006). The paper followed in 1931 (*Monatshefte für Mathematik und Physik* 38: 173–198).

> วันก่อนหน้านั้น ในการประชุมโต๊ะกลมที่ปิดท้ายการประชุมวิชาการในเมืองเดียวกัน นักตรรกวิทยาหนุ่มชื่อ Kurt Gödel พูดขึ้นมาเกือบจะเป็นการเอ่ยผ่านๆ ว่าระบบรูปนัย (formal system) ที่สอดคล้องในตัวเองใดๆ ที่แข็งแรงพอจะอธิบายเลขคณิตได้ ต้องมีข้อความจริงบางข้อที่มันพิสูจน์ไม่ได้อยู่ในตัว ห้องประชุมปล่อยผ่านไปเฉยๆ; John von Neumann ซึ่งอยู่ในที่ประชุมด้วย ดูเหมือนจะเป็นคนเดียวที่จับได้ว่าเพิ่งเกิดอะไรขึ้น (Davis, Notices of the AMS, 2006) บทความตามมาในปี 1931 (Monatshefte für Mathematik und Physik 38: 173–198)

Gödel had holed the ship, but the *Entscheidungsproblem* stayed afloat: you cannot prove no procedure exists until you have said what a procedure is.

> Gödel เจาะรูเรือไปแล้ว แต่ Entscheidungsproblem ยังลอยอยู่ได้: คุณพิสูจน์ไม่ได้ว่าไม่มีกระบวนการใดอยู่เลย จนกว่าคุณจะนิยามก่อนว่า "กระบวนการ" คืออะไร

The definition was found by watching a person. Alan Turing's move, at twenty-three and a Fellow of King's College, was less mathematical than anthropological. In 1936 a *computer* was a job title: a human being, usually poorly paid, who calculated. Turing watched one. "Computing is normally done by writing certain symbols on paper," he wrote. "We may suppose this paper is divided into squares like a child's arithmetic book" (*On Computable Numbers*, §9). Then he stripped the human away one faculty at a time: a bound on the symbols taken in at a glance, a bound on the "states of mind" he can hold, nothing to do but look, write, and move. What remains is a tape, a finite table of rules, and a head that reads and writes — the Turing

machine, less a machine than a description of what a person does when they stop thinking and start following instructions. The London Mathematical Society received it on 28 May 1936.

> นิยามนี้ถูกค้นพบจากการเฝ้าดูคน การเคลื่อนไหวของ Alan Turing ตอนอายุยี่สิบสามปีและเป็น Fellow ของ King's College นั้นเป็นแนวมานุษยวิทยามากกว่าคณิตศาสตร์ ในปี 1936 คำว่า computer ยังเป็นชื่อตำแหน่งงาน: มนุษย์คนหนึ่ง มักได้ค่าจ้างต่ำ ที่ทำหน้าที่คำนวณ Turing เฝ้าดูคนคนนั้น เขาเขียนไว้ว่า "การคำนวณโดยปกติทำโดยการเขียนสัญลักษณ์บางอย่างลงบนกระดาษ เราอาจสมมติว่ากระดาษแผ่นนี้ถูกแบ่งเป็นช่องสี่เหลี่ยมเหมือนสมุดเลขคณิตของเด็ก" (On Computable Numbers, §9) จากนั้นเขาก็ค่อยๆ ถอดความสามารถของมนุษย์ออกทีละอย่าง: กำหนดขอบเขตของสัญลักษณ์ที่รับรู้ได้ในการมองแต่ละครั้ง กำหนดขอบเขตของ "สภาวะทางความคิด (states of mind)" ที่เขาจำไว้ได้ ไม่มีอะไรให้ทำนอกจากมอง เขียน และขยับ สิ่งที่เหลืออยู่คือเทป ตารางกฎจำนวนจำกัด และหัวอ่านที่อ่านและเขียนได้ — คือเครื่องทัวริง (Turing machine) ที่เป็นคำอธิบายว่ามนุษย์ทำอะไรเมื่อหยุดคิดแล้วเริ่มทำตามคำสั่ง มากกว่าจะเป็นเครื่องจักรจริงๆ เสียอีก London Mathematical Society ได้รับบทความนี้ไว้เมื่อวันที่ 28 พฤษภาคม 1936

Everyone who defined it independently landed in the same place. Alonzo Church got there first by another road: an April 1936 proof, built on his lambda calculus, that no such procedure exists (*American Journal of Mathematics* 58: 345–363). Emil Post reached a near-identical formalism that year; Gödel and Kleene, general recursive functions. All define exactly the same class, and that convergence is the evidence for the Church–Turing thesis: this class *is* what we informally mean by "effectively calculable." It is a thesis, not a theorem, and cannot be otherwise: it joins an informal human idea to a formal object, and no proof bridges that gap (Stanford Encyclope-

dia of Philosophy). It is also why this floor's results are hardware-proof: build a computer from anything you like — tubes, silicon, qubits — and if it computes at all, it computes this set and no more.

> ทุกคนที่นิยามมันขึ้นมาอย่างเป็นอิสระต่อกัน ต่างมาลงเอยที่จุดเดียวกัน Alonzo Church ไปถึงจุดนั้นก่อนใครด้วยเส้นทางอื่น: บทพิสูจน์เมื่อเดือนเมษายน 1936 ที่สร้างขึ้นบน lambda calculus ของเขา ว่าไม่มีกระบวนการแบบนั้นอยู่จริง (American Journal of Mathematics 58: 345–363) Emil Post ก็ไปถึงระบบรูปนัยที่แทบจะเหมือนกันในปีเดียวกัน; Gödel กับ Kleene ไปถึงด้วย general recursive functions ทั้งหมดนี้นิยามคลาสเดียวกันเป๊ะ และการที่ต่างเส้นทางมาบรรจบกันนี้เองคือหลักฐานของข้อตั้ง Church–Turing (Church–Turing thesis): คลาสนี้คือสิ่งที่เราหมายถึงอย่างไม่เป็นทางการเมื่อพูดว่า "คำนวณได้อย่างมีประสิทธิผล" ("effectively calculable") มันเป็นข้อตั้ง ไม่ใช่ทฤษฎีบท และเป็นอย่างอื่นไปไม่ได้: มันเชื่อมไอเดียของมนุษย์ที่ไม่เป็นทางการเข้ากับวัตถุที่เป็นรูปนัย และไม่มีบทพิสูจน์ใดเชื่อมช่องว่างนั้นได้ (Stanford Encyclopedia of Philosophy) นี่ก็เป็นเหตุผลว่าทำไมผลลัพธ์ของชั้นนี้ถึงพิสูจน์ได้ไม่ว่าจะใช้ฮาร์ดแวร์ชนิดไหน: สร้างเครื่องคำนวณขึ้นจากอะไรก็ได้ที่คุณอยากใช้ — หลอดสุญญากาศ ซิลิคอน คิวบิต (qubit) — ถ้ามันคำนวณได้จริง มันก็คำนวณได้แค่คลาสนี้เท่านั้น ไม่มากไปกว่านี้

One machine can be all machines. Section 6 of Turing's paper opens with the most consequential sentence in it: "It is possible to invent a single machine which can be used to compute any computable sequence." Hand a machine the *description* of another and it behaves as that one. Before this, a calculating device was built for its task — a loom for weaving, a differential analyzer for equations. After, the machine is blank and the task arrives as data. Everything downstream depends on that: software separable from hardware, the compiler, the emulator, the virtual machine, the operating system running programs it was never designed for. Universality is the hinge the tower swings on.

> เครื่องเดียวเป็นเครื่องทุกเครื่องได้ ส่วนที่ 6 ของบทความของ Turing เปิดด้วยประโยคที่ส่งผลสะเทือนที่สุดในบทความทั้งฉบับ: "เป็นไปได้ที่จะประดิษฐ์เครื่องเดี่ยวเครื่องหนึ่งขึ้นมา ซึ่งใช้คำนวณลำดับที่คำนวณได้ใดๆ ก็ได้ทั้งหมด" ("It is possible

> to invent a single machine which can be used to compute any computable sequence.") ป้อนคำอธิบายของเครื่องอื่นให้กับเครื่องเครื่องหนึ่ง แล้วมันจะทำตัวเป็นเครื่องนั้นได้เลย ก่อนหน้านี้ อุปกรณ์คำนวณถูกสร้างขึ้นมาเพื่องานเฉพาะของมันเท่านั้น — กี่ทอผ้าสำหรับทอผ้า เครื่องวิเคราะห์เชิงอนุพันธ์ (differential analyzer) สำหรับสมการ หลังจากนั้น เครื่องเป็นแผ่นว่างเปล่า และงานมาถึงในรูปข้อมูล ทุกอย่างที่ตามมาต่อจากนี้ขึ้นอยู่กับจุดนั้น: ซอฟต์แวร์ที่แยกออกจากฮาร์ดแวร์ได้ คอมไพเลอร์ อีมูเลเตอร์ (emulator) เครื่องเสมือน (virtual machine) ระบบปฏิบัติการที่รันโปรแกรมซึ่งไม่เคยถูกออกแบบมาให้มันรันได้ตั้งแต่ต้น ความเป็นสากล (universality) คือบานพับที่หอคอยทั้งหลังแกว่งอยู่บน

And some questions simply have no procedure. The proof is Cantor's diagonal argument of 1891, turned on machines. Suppose the halting checker exists. Build a program that hands the checker its own description and does the opposite of the prediction: halt if told it loops, loop if told it halts. Ask the checker about that program and it must be wrong. So it never existed.

> และบางคำถามก็ไม่มีกระบวนการตอบเลยจริงๆ บทพิสูจน์คือการทแยงมุม (diagonalization) ของ Cantor จากปี 1891 ที่ถูกหันมาใช้กับเครื่องจักร สมมติว่าตัวตรวจสอบการหยุด (halting checker) มีอยู่จริง สร้างโปรแกรมหนึ่งที่ป้อนคำอธิบายของตัวมันเองให้ตัวตรวจสอบ แล้วทำตรงข้ามกับสิ่งที่ตัวตรวจสอบทำนายไว้: หยุดถ้าถูกบอกว่ามันจะวนลูป วนลูปถ้าถูกบอกว่ามันจะหยุด ถามตัวตรวจสอบเกี่ยวกับโปรแกรมนั้น มันต้องตอบผิดเสมอ ดังนั้นมันจึงไม่เคยมีอยู่จริงตั้งแต่แรก

Two footnotes of honesty. Turing used neither that name nor that proof; he proved undecidability for related problems about his machines, and both name and modern formulation appear later, in Martin Davis's *Computability and Unsolvability* of 1958 (Hamkins & Nenu, 2024). And halting is no exotic special case: H. G. Rice proved in 1953 that *every* non-trivial question about what a program computes is undecidable (*Transactions of the American Mathematical Society* 74: 358–366). That is a fence, not a pothole — why no compiler will promise your loop terminates, why no antivirus can decide by

analysis alone whether a program is malicious, and why every static analyzer answers "yes," "no," or "I cannot tell." That third answer is a theorem, not a bug.

> เชิงอรรถแห่งความซื่อตรงสองข้อ Turing ไม่ได้ใช้ทั้งชื่อนั้นและบทพิสูจน์แบบนั้นเลย เขาพิสูจน์การตัดสินไม่ได้ (undecidability) ของปัญหาที่เกี่ยวข้องกับเครื่องจักรของเขาเท่านั้น ทั้งชื่อ "ปัญหาการหยุด (halting problem)" และรูปแบบสมัยใหม่ปรากฏขึ้นภายหลัง ใน Computability and Unsolvability ของ Martin Davis ปี 1958 (Hamkins & Nenu, 2024) และการหยุดก็ไม่ใช่กรณีพิเศษแปลกๆ เลย: H. G. Rice พิสูจน์ในปี 1953 ว่าคำถามที่ไม่ธรรมดาทุกข้อเกี่ยวกับสิ่งที่โปรแกรมคำนวณนั้นตัดสินไม่ได้ทั้งหมด (Transactions of the American Mathematical Society 74: 358–366) นั่นคือรั้วกั้นเขต ไม่ใช่หลุมบ่อ — คือเหตุผลที่ไม่มีคอมไพเลอร์ตัวไหนจะรับประกันได้ว่าลูปของคุณจะจบ เหตุผลที่ไม่มีโปรแกรมป้องกันไวรัสตัวไหนตัดสินได้จากการวิเคราะห์อย่างเดียวว่าโปรแกรมหนึ่งเป็นอันตรายหรือไม่ และเหตุผลที่ static analyzer ทุกตัวตอบได้แค่ "ใช่" "ไม่ใช่" หรือ "บอกไม่ได้" คำตอบที่สามนั้นคือทฤษฎีบท ไม่ใช่บั๊ก

Past the wall of the possible stands a second wall: the affordable. A problem can be decidable and still cost more than the universe can pay. Stephen Cook showed in 1971 that Boolean satisfiability is NP-complete, and Leonid Levin proved as much independently in 1973 (Cook's own account). The instrument behind such a claim is *reduction*: a translation cheap enough that a fast solver for one problem becomes a fast solver for another — one hard problem made into a lever on all the rest. In 1972 Richard Karp used it to show that twenty more natural problems were the same problem in disguise (Karp, 1972); by 1979 Garey and Johnson's *Computers and Intractability* catalogued three hundred. Whether checking an answer quickly implies finding one quickly — P versus NP — was posed in a letter Gödel wrote to the dying von Neumann on 20 March 1956 (translation), carries a million-dollar Clay Institute prize, and is unsolved (CMI). The wall leaks usefully,

though: the best proven bound for 3-SAT is exponential, yet industrial solvers routinely dispatch structured instances of over a million variables (Gomes et al.). Worst case is not typical case — where the next floor begins.

> พ้นกำแพงแห่งความเป็นไปได้ไปแล้ว ยังมีกำแพงที่สองยืนอยู่: กำแพงแห่งความคุ้มต้นทุน ปัญหาหนึ่งอาจตัดสินได้ (decidable) แต่ยังมีต้นทุนแพงกว่าที่เอกภพทั้งใบจะจ่ายไหว Stephen Cook แสดงให้เห็นในปี 1971 ว่า Boolean satisfiability เป็น NP-complete และ Leonid Levin ก็พิสูจน์เรื่องเดียวกันได้อย่างเป็นอิสระในปี 1973 (บันทึกของ Cook เอง) เครื่องมือเบื้องหลังข้อกล่าวอ้างแบบนี้คือ การลดรูป (reduction): การแปลงที่ถูกพอจนตัวแก้ปัญหาที่เร็วสำหรับปัญหาหนึ่ง กลายเป็นตัวแก้ปัญหาที่เร็วสำหรับอีกปัญหาหนึ่งได้ด้วย — ปัญหายากหนึ่งข้อถูกทำให้กลายเป็นคันโยกที่ยกปัญหาที่เหลือทั้งหมดได้ ปี 1972 Richard Karp ใช้มันแสดงให้เห็นว่าปัญหาที่ดูเป็นธรรมชาติอีกยี่สิบข้อ แท้จริงแล้วคือปัญหาเดียวกันที่แค่ปลอมตัวมา (Karp, 1972); พอถึงปี 1979 หนังสือ Computers and Intractability ของ Garey กับ Johnson ก็รวบรวมได้ถึงสามร้อยปัญหา ส่วนคำถามว่าการตรวจคำตอบได้เร็วบ่งบอกว่าหาคำตอบได้เร็วด้วยหรือไม่ — P vs NP — ถูกตั้งขึ้นในจดหมายที่ Gödel เขียนถึง von Neumann ผู้กำลังจะเสียชีวิตเมื่อวันที่ 20 มีนาคม 1956 (คำแปล) มาพร้อมเงินรางวัลหนึ่งล้านดอลลาร์จาก Clay Institute และยังไม่มีคำตอบมาจนถึงทุกวันนี้ (CMI) แต่กำแพงนี้ก็รั่วอย่างเป็นประโยชน์อยู่บ้าง: ขอบเขตที่พิสูจน์ได้ดีที่สุดสำหรับ 3-SAT คือเชิงเลขชี้กำลัง (exponential) แต่ตัวแก้ปัญหาระดับอุตสาหกรรมกลับจัดการอินสแตนซ์ที่มีโครงสร้างชัดเจนซึ่งมีตัวแปรกว่าล้านตัวได้เป็นประจำ (Gomes et al.) กรณีแย่สุด (worst case) ไม่ใช่กรณีทั่วไป (typical case) — และนี่คือจุดที่ชั้นถัดไปเริ่มต้น

Connections · เชื่อมโยง

Below: 00, the map. Above: 02, which takes every problem this floor calls possible and asks what it costs, turning the same counting arguments into lower bounds. Universality is what 05 sells as virtualization and 06 as compilation. 11 builds on the second wall: cryptography is the deliberate manufacture of problems believed unaffordable.

> ชั้นล่าง: บทที่ 00 แผนที่ ชั้นบน: บทที่ 02 ซึ่งหยิบทุกปัญหาที่ชั้นนี้เรียกว่าเป็นไปได้มาถามต่อว่ามันมีต้นทุนเท่าไร โดยเปลี่ยนข้อโต้แย้งเชิงการนับแบบเดียวกันให้กลายเป็นขอบล่าง (lower bound) ความเป็นสากล (universality) คือสิ่งที่บทที่ 05 ขายในชื่อการทำเสมือน (virtualization) และบทที่ 06 ขายในชื่อการคอมไพล์ (compilation) บทที่ 11 สร้างต่อจากกำแพงที่สอง: วิทยาการรหัสลับ (cryptography) คือการจงใจสร้างปัญหาที่เชื่อกันว่าไม่คุ้มต้นทุนที่จะแก้ขึ้นมา

Open questions · คำถามที่ยังเปิดอยู่

P versus NP, obviously — though a proof that P = NP would be less a theorem than an event: the security of the internet dissolved, much of mathematics handed to a machine. Is the thesis itself safe? Quantum computers compute nothing a Turing machine cannot, but may break its *efficiency* version — chapter 14's territory. And the deepest: undecidability limits machines and, through the thesis, any procedure a human could follow. Does anything escape it, or is the fence around the possible simply the shape of thought?

> P vs NP แน่นอนอยู่แล้ว — แม้ว่าบทพิสูจน์ว่า P = NP จะเป็นเหตุการณ์มากกว่าจะเป็นแค่ทฤษฎีบท: ความปลอดภัยของอินเทอร์เน็ตจะสลายไป คณิตศาสตร์ส่วนใหญ่จะถูกส่งมอบให้เครื่องจักรทำแทน ข้อตั้ง Church–Turing เองปลอดภัยหรือเปล่า? คอมพิวเตอร์ควอนตัม (quantum computer) คำนวณอะไรที่ Turing machine คำนวณไม่ได้เลยสักอย่าง แต่อาจทำลายเวอร์ชันประสิทธิภาพของมันได้ — นั่นคือดินแดนของบทที่ 14 และคำถามที่ลึกที่สุด: การตัดสินไม่ได้ (undecidability) จำกัด

> ขอบเขตของเครื่องจักร และผ่านทางข้อตั้งนี้ ก็จำกัดกระบวนการใดๆ ที่มนุษย์จะทำตามได้ด้วย มีอะไรหนีพ้นมันได้ไหม หรือรั้วที่ล้อมรอบสิ่งที่เป็นไปได้ ก็คือรูปทรงของความคิดเองนั่นแหละ?

Go deeper · อ่านต่อ

— Michael Sipser, *Introduction to the Theory of Computation*, 3rd ed., Cengage, 2012 — the standard course text (author's page).

> Michael Sipser, Introduction to the Theory of Computation, 3rd ed., Cengage, 2012 — ตำราเรียนมาตรฐานของวิชานี้ (หน้าเว็บของผู้เขียน)

— A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, ser. 2, 42 (1936–37): 230–265 — worth reading for §9 alone (PDF).

> A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," Proceedings of the London Mathematical Society, ser. 2, 42 (1936–37): 230–265 — คุ้มค่าจะอ่านแค่ §9 ก็ยังคุ้ม (PDF)

— Richard Karp, "Reducibility Among Combinatorial Problems," 1972 — the paper that turned NP-completeness from a theorem into a map (PDF).

> Richard Karp, "Reducibility Among Combinatorial Problems," 1972 — เปเปอร์ที่เปลี่ยน NP-completeness จากทฤษฎีบทให้กลายเป็นแผนที่ (PDF)

CHAPTER 02

Algorithms & Data Structures

อัลกอริทึมและโครงสร้างข้อมูล

This chapter is the floor immediately above computability: 01 says a task can be done at all, and this floor asks what doing it will cost.

บทนี้คือชั้นที่อยู่เหนือเรื่องคำนวณได้ (computability) ขึ้นมาทันที — บทที่ 01 บอกว่างานหนึ่งทำได้หรือไม่ ส่วนชั้นนี้ถามว่าทำแล้วต้องเสียต้นทุนเท่าไร

Given that it can be computed, how well can it be done?

เมื่อรู้แล้วว่างานนั้นคำนวณได้ จะทำได้ดีแค่ไหน

In 1960, at a seminar he was running at Moscow State University, Andrey Kolmogorov told the room that multiplying two n-digit numbers must cost on the order of n^2 elementary operations — that the method every schoolchild learns, digit against digit, is essentially the best there is. It was a reasonable thing to believe. Every multiplication algorithm anyone had written behaved that way, and Kolmogorov was not a man whose conjectures were casually wrong.

> ปี 1960 ในสัมมนาที่เขาเป็นผู้นำที่มหาวิทยาลัยมอสโก Andrey Kolmogorov บอกกับคนในห้องว่า การคูณเลข n หลักสองจำนวนต้องเสียต้นทุนราว n^2 ครั้งของการกระทำพื้นฐาน — วิธีที่เด็กนักเรียนทุกคนเรียนกัน คูณทีละหลัก คือวิธีที่ดีที่สุดเท่าที่จะมีได้ นี่เป็นเรื่องที่เชื่อได้อย่างมีเหตุผล เพราะอัลกอริทึม (algorithm) การคูณทุกตัวที่มีคนเขียนขึ้นมาล้วนทำงานแบบนั้น และ Kolmogorov ก็ไม่ใช่คนที่ข้อคาดการณ์ของเขาจะผิดกันง่ายๆ

A student in the room, Anatoly Karatsuba, broke it inside a week. His trick is almost embarrassingly small: cut each number in half. The schoolbook method then wants four half-sized products; Karatsuba noticed that three

suffice, because the fourth quantity can be recovered by subtracting the ones you already have. Do that recursively and the cost falls from n^2 to $n^{\log_2 3}$ — about $n^{1.585}$. At the next meeting Kolmogorov reported the refutation himself, and then discontinued the seminar (Karatsuba, *The Complexity of Computations*, 1995).

> นักศึกษาคนหนึ่งในห้องนั้น Anatoly Karatsuba หักล้างมันได้ภายในหนึ่งสัปดาห์ กลเม็ดของเขาเล็กจนน่าอายด้วยซ้ำ คือตัดเลขแต่ละจำนวนออกเป็นครึ่งหนึ่ง วิธีแบบตำราต้องใช้ผลคูณขนาดครึ่งหนึ่งถึงสี่ตัว แต่ Karatsuba สังเกตว่าแค่สามตัวก็พอ เพราะปริมาณตัวที่สี่หาได้จากการลบตัวที่มีอยู่แล้วออกจากกัน ทำเช่นนี้แบบเรียกซ้ำ (recursion) ต้นทุนจะลดจาก n^2 เหลือ $n^{\log_2 3}$ หรือราว $n^{1.585}$ ในการประชุมครั้งถัดมา Kolmogorov รายงานข้อหักล้างนี้ด้วยตัวเอง แล้วก็ยกเลิกสัมมนานั้นไปเลย (Karatsuba, The Complexity of Computations, 1995)

Nothing in the world had changed that week. No new machine, no new physics. A fact about *cost* had been discovered, the way facts about nature are discovered — and that is what this floor studies.

> ไม่มีอะไรในโลกเปลี่ยนไปในสัปดาห์นั้น ไม่มีเครื่องใหม่ ไม่มีฟิสิกส์ใหม่ สิ่งที่ถูกค้นพบคือข้อเท็จจริงเกี่ยวกับ ต้นทุน ในแบบเดียวกับที่ข้อเท็จจริงเกี่ยวกับธรรมชาติถูกค้นพบ — และนี่คือสิ่งที่ชั้นนี้ศึกษา

Cost is a shape, not a number. The notation for it was borrowed from number theory: Paul Bachmann introduced big-O in 1894, and Edmund Landau adopted it in 1909, crediting him (MAA Convergence); Donald Knuth's 1976 note in *SIGACT News* pushed computing to standardize on O, Ω and Θ (Knuth, 1976). Notice what the notation deliberately throws away: constant factors. That is not sloppiness. Constants are the part of the cost that hardware, compilers, and careful coding can move; the growth *rate* is the part the algorithm itself owns, and no amount of engineering below it will

change it. A machine ten times faster buys you a constant. An exponent buys you the future: on Karatsuba's exponents alone, the saving is roughly eighteen-fold at a thousand digits and three hundred-fold at a million.

> ต้นทุนเป็นรูปทรง ไม่ใช่ตัวเลข สัญกรณ์ที่ใช้แทนมันยืมมาจากทฤษฎีจำนวน — Paul Bachmann เสนอ Big-O ขึ้นเป็นครั้งแรกในปี 1894 แล้ว Edmund Landau นำมาใช้ต่อในปี 1909 พร้อมให้เครดิตเขา (MAA Convergence) บันทึกของ Donald Knuth ในปี 1976 ใน SIGACT News ผลักดันให้วงการคอมพิวเตอร์ใช้ O, Ω และ Θ เป็นมาตรฐานร่วมกัน (Knuth, 1976) สังเกตว่าสัญกรณ์นี้ตั้งใจทิ้งอะไรไป — ทิ้งค่าคงที่ นั่นไม่ใช่ความมักง่าย ค่าคงที่คือส่วนของต้นทุนที่ฮาร์ดแวร์ คอมไพเลอร์ และการเขียนโค้ดอย่างระมัดระวังยังขยับได้ ส่วน อัตรา การเติบโตคือส่วนที่อัลกอริทึมเป็นเจ้าของเอง ไม่มีวิศวกรรมชั้นล่างเท่าไรก็เปลี่ยนมันไม่ได้ เครื่องที่เร็วขึ้นสิบเท่าซื้อค่าคงที่ให้คุณได้ แต่เลขชี้กำลังซื้ออนาคตให้คุณ — เฉพาะเลขชี้กำลังของ Karatsuba อย่างเดียว ก็ประหยัดได้ราวสิบแปดเท่าที่พันหลัก และสามร้อยเท่าที่ล้านหลัก

A small catalogue of strategies covers most of the ground. They are worth knowing by name, because almost every algorithm you will meet is one of them wearing local clothes.

> รายการกลยุทธ์เล็กๆ ชุดหนึ่งครอบคลุมพื้นที่ส่วนใหญ่ของสนามนี้ไว้ได้ คุ้มที่จะรู้จักไว้ทีละชื่อ เพราะแทบทุกอัลกอริทึมที่คุณจะเจอ ก็คือหนึ่งในนี้ที่สวมเสื้อผ้าเฉพาะถิ่นเข้ามาเท่านั้น

Divide and conquer splits a problem into smaller copies of itself, solves those, and stitches the answers together — Karatsuba's trick, and merge-sort, which von Neumann wrote out in 1945 in what survives as the earliest example of his coding (Knuth, "Von Neumann's First Computer Program," 1970). Its most famous application is a three-page paper by Volker Strassen with the flatly provocative title "Gaussian elimination is not optimal," which multiplied matrices in about n^2.807 instead of n^3 (*Numerische Mathematik*

13, 1969). The chase it started has run for over half a century and is still running: the published record now stands near n^2.371 (Alman et al., SODA 2025), and nobody knows how far down it goes.

> แบ่งแล้วพิชิต (divide and conquer) แยกปัญหาออกเป็นสำเนาย่อยของตัวมันเอง แก้ปัญหาย่อยเหล่านั้น แล้วเย็บคำตอบเข้าด้วยกัน — นี่คือกลเม็ดของ Karatsuba และ mergesort ซึ่ง von Neumann เขียนไว้ในปี 1945 เป็นตัวอย่างแรกสุดของงานเขียนโค้ดของเขาที่ยังหลงเหลืออยู่ (Knuth, "Von Neumann's First Computer Program," 1970) การประยุกต์ใช้ที่โด่งดังที่สุดคือเปเปอร์สามหน้าของ Volker Strassen ที่ตั้งชื่อยั่วยุตรงๆ ว่า "Gaussian elimination is not optimal" ซึ่งคูณเมทริกซ์ (matrix multiplication) ได้ในราว n^2.807 แทนที่จะเป็น n^3 (Numerische Mathematik 13, 1969) การไล่ล่าที่เปเปอร์นี้จุดชนวนไว้ดำเนินมาแล้วกว่าครึ่งศตวรรษและยังไม่หยุด สถิติที่ตีพิมพ์ล่าสุดอยู่ที่ราว n^2.371 (Alman et al., SODA 2025) และไม่มีใครรู้ว่ามันจะลดลงไปได้อีกแค่ไหน

The *greedy* strategy takes the locally best step and never looks back — either naive or optimal, depending entirely on whether the problem has the structure that rewards it — and when it does, greed is unbeatable. Dijkstra's shortest-path algorithm is the canonical case, published in three pages in 1959 (*Numerische Mathematik* 1: 269–271). He designed it, by his own account, in about twenty minutes in 1956, without pencil or paper, sitting on a café terrace in Amsterdam while out shopping with his fiancée (oral history, Charles Babbage Institute, 2001).

> กลยุทธ์ greedy เลือกก้าวที่ดีที่สุดในจุดนั้นๆ แล้วไม่หันกลับไปมองอีก — จะไร้เดียงสาหรือเหมาะที่สุดก็ขึ้นอยู่กับว่าโจทย์นั้นมีโครงสร้างที่ตอบแทนวิธีนี้หรือเปล่า ล้วนๆ และเมื่อมันมี greedy ก็เอาชนะไม่ได้เลย อัลกอริทึมหาเส้นทางสั้นสุดของ Dijkstra คือตัวอย่างคลาสสิกของเรื่องนี้ ตีพิมพ์ในสามหน้าเมื่อปี 1959 (Numerische Mathematik 1: 269–271) ตามคำบอกเล่าของเขาเอง เขาคิดมันขึ้นมาในเวลาราวยี่สิบนาทีเมื่อปี 1956 โดยไม่มีดินสอหรือกระดาษ นั่งอยู่ที่ระเบียงคาเฟ่ในอัมสเตอร์ดัมระหว่างออกไปช็อปปิ้งกับคู่หมั้น (oral history, Charles Babbage Institute, 2001)

When subproblems repeat, *dynamic programming* says to remember rather than recompute. Richard Bellman developed it at RAND, where he worked from 1948 to 1965, and the name is a deliberate disguise: RAND answered to a Secretary of Defense who, in Bellman's telling, "had a pathological fear and hatred of the word research," so he chose two words nobody could object to (Dreyfus, "Richard Bellman on the Birth of Dynamic Programming," *Operations Research* 50, no. 1, 2002). The technique is more honest than its name — it is simply the trade of memory for time.

> เมื่อโจทย์ย่อยเกิดซ้ำ dynamic programming บอกให้จำเอาไว้แทนที่จะคำนวณใหม่ Richard Bellman พัฒนามันขึ้นที่ RAND ซึ่งเขาทำงานอยู่ตั้งแต่ปี 1948 ถึง 1965 และชื่อนี้คือการอำพรางที่ตั้งใจ — RAND ขึ้นตรงกับรัฐมนตรีกลาโหมที่ตามคำบอกเล่าของ Bellman "กลัวและเกลียดคำว่า research อย่างผิดปกติ" เขาจึงเลือกใช้สองคำที่ไม่มีใครคัดค้านได้แทน (Dreyfus, "Richard Bellman on the Birth of Dynamic Programming," Operations Research 50, no. 1, 2002) เทคนิคนี้ซื่อตรงกว่าชื่อของมันเสียอีก — มันคือการแลกหน่วยความจำเอาเวลากลับมาเท่านั้นเอง

Strangest of the four is *randomization*, which wins by refusing to be predictable. Tony Hoare devised Quicksort in 1959–60 as a student in Moscow, working on machine translation of Russian, and published it in 1962 (*The Computer Journal* 5, no. 1: 10–16). Its worst case is quadratic and an adversary who knows your rule can force it every time; choose the pivot at random and that adversary is disarmed, because now they would have to guess your dice.

> แปลกที่สุดในสี่กลยุทธ์นี้คือ การสุ่ม (randomization) ซึ่งชนะด้วยการปฏิเสธที่จะให้ทายทางได้ Tony Hoare คิดค้น Quicksort ขึ้นระหว่างปี 1959–60 ตอนเป็นนักศึกษาอยู่ที่มอสโก ทำงานด้านการแปลภาษารัสเซียด้วยเครื่อง แล้วตีพิมพ์มันในปี 1962 (The Computer Journal 5, no. 1: 10–16) กรณีแย่สุด (worst case) ของมันเป็นแบบยกกำลังสอง (quadratic) และฝ่ายตรงข้าม (adversary) ที่รู้กฎของคุณสามารถบังคับให้เกิดกรณีนั้นได้ทุกครั้ง เลือกจุดหมุน (pivot) แบบสุ่มแล้วฝ่ายตรงข้ามคนนั้นก็หมดทางสู้ เพราะตอนนี้พวกเขาต้องมาเดาลูกเต๋าของคุณแทน

A data structure is a payment plan. This is the idea that makes the second half of the field cohere. An unsorted array is cheap to add to and expensive to search: you pay at query time, every time. Sorting it is prepayment — one large charge now in exchange for logarithmic lookups forever after. A hash table pays at insertion for near-instant equality lookups but cannot answer "what comes next"; a balanced tree charges a little more per operation and answers ordered questions the hash table cannot. Databases call the same decision an index, and the trade is identical: writes get slower so that reads get faster (chapter 08). Nothing here is free; you are choosing only the currency. And the currency need not be time at all — precompute a lookup table and you have bought speed with memory, which is the space–time trade in its purest form.

> โครงสร้างข้อมูล (data structure) คือแผนการจ่ายเงินแบบหนึ่ง นี่คือไอเดียที่ทำให้ครึ่งหลังของสาขานี้เชื่อมกันเป็นเนื้อเดียว อาร์เรย์ที่ยังไม่เรียงลำดับเพิ่มข้อมูลเข้าไปได้ถูก แต่ค้นหา (search) แพง — คุณต้องจ่ายตอนคิวรี (query) ทุกครั้งไป แต่ถ้าเรียงลำดับ (sorting) มันไว้ก่อน นั่นคือการจ่ายล่วงหน้า จ่ายก้อนใหญ่ครั้งเดียวตอนนี้แลกกับการค้นหาที่เร็วแบบลอการิทึมไปตลอดหลังจากนั้น hash table จ่ายตอนใส่ข้อมูลเข้าไปเพื่อแลกกับการค้นหาความเท่ากันที่เร็วเกือบทันที แต่ตอบคำถามว่า "ตัวถัดไปคืออะไร" ไม่ได้ ต้นไม้สมดุล (balanced tree) คิดราคาต่อการกระทำแพงกว่านิดหน่อย แต่ตอบคำถามเชิงลำดับที่ hash table ตอบไม่ได้ ฐานข้อมูลเรียกการตัดสินใจแบบเดียวกันนี้ว่าดัชนี (index) และการแลกได้แลกเสียก็เหมือนกันเป๊ะ — เขียนช้าลงเพื่อให้อ่านเร็วขึ้น (บทที่ 08) ไม่มีอะไรในนี้ฟรี คุณแค่เลือกสกุลเงินที่จะจ่าย และสกุลเงินนั้นก็ไม่จำเป็นต้องเป็นเวลาเสมอไป — คำนวณตารางค้นหาไว้ล่วงหน้า แล้วคุณก็ซื้อความเร็วมาด้วยหน่วยความจำ นี่คือการแลกได้แลกเสียระหว่างพื้นที่กับเวลาในรูปแบบที่บริสุทธิ์ที่สุด

Some costs are floors, not failures of imagination. Any algorithm that sorts by comparing elements is, in effect, a decision tree: each comparison branches two ways, and the tree must have a distinct leaf for every one of the n! possible orderings. A binary tree with n! leaves has depth at least $\log_2(n!)$, which grows like n log n. So no comparison sort can beat $\Omega(n \log n)$ — not one that is cleverer, only one that stops comparing. Counting sort es-

capes precisely because it looks at the values themselves rather than at pairwise comparisons. Notice that this is chapter 01's move — counting what a procedure can possibly distinguish — aimed at cost rather than at possibility.

> ต้นทุนบางอย่างคือขอบล่าง (lower bound) ไม่ใช่ความล้มเหลวทางจินตนาการ อัลกอริทึมใดก็ตามที่เรียงลำดับด้วยการเปรียบเทียบสมาชิก (comparison sort) โดยเนื้อแท้แล้วก็คือต้นไม้การตัดสินใจ (decision tree) — การเปรียบเทียบแต่ละครั้งแตกออกเป็นสองทาง และต้นไม้นั้นต้องมีใบที่ต่างกันสำหรับทุกหนึ่งในความเป็นไปได้ n! แบบของการเรียงลำดับ ต้นไม้แบบไบนารี (binary tree) ที่มี n! ใบ ต้องมีความลึกอย่างน้อย $\log_2(n!)$ ซึ่งโตแบบ n log n ฉะนั้นไม่มี comparison sort ตัวไหนเอาชนะ Ω(n log n) ได้ — ไม่ใช่เพราะยังไม่มีตัวที่ฉลาดกว่านี้ แต่เพราะมีแต่ตัวที่เลิกเปรียบเทียบเท่านั้นที่จะทำได้ counting sort หลุดพ้นจากขอบเขตนี้ได้ก็เพราะมันมองที่ค่าจริงๆ แทนที่จะมองการเปรียบเทียบเป็นคู่ๆ สังเกตว่านี่คือท่าเดียวกับบทที่ 01 — การนับว่ากระบวนการหนึ่งแยกแยะอะไรได้บ้าง — เพียงแต่เล็งไปที่ต้นทุนแทนที่จะเล็งไปที่ความเป็นไปได้

And the model leaks. Big-O hides constants, and sometimes the constants are the whole story: Lipton and Regan named the class of *galactic algorithms* — asymptotically superior, never used, because the input size at which they win exceeds anything that exists (2010). Big-O also hides the machine. Two loops with identical asymptotic cost can differ tenfold because one walks memory in order and the other jumps, which is chapter 04's floor leaking upward. And worst case is not typical case, as chapter 01's SAT solvers demonstrate daily. Asymptotic cost is the price on the label; what you pay at the till still depends on the machine.

> และแบบจำลองนี้ก็รั่ว (leak) เหมือนกัน Big-O ซ่อนค่าคงที่ไว้ และบางครั้งค่าคงที่นั่นแหละคือเรื่องทั้งหมด — Lipton และ Regan ตั้งชื่อกลุ่มที่เรียกว่า อัลกอริทึมกาแล็กติก (galactic algorithms) ให้กับอัลกอริทึมที่เหนือกว่าในเชิงเส้นกำกับ (asymptotic) แต่ไม่เคยถูกใช้จริง เพราะขนาดอินพุตที่มันจะเริ่มชนะนั้นใหญ่เกินกว่าอะไรที่มีอยู่จริง (2010) Big-O ยังซ่อนตัวเครื่องไว้ด้วย ลูปสองลูปที่มีต้นทุนแบบ asymptotic เท่ากันเป๊ะ อาจต่างกันสิบเท่าได้ เพราะลูปหนึ่งเดินหน่วยความจำเรียง

ตามลำดับ ส่วนอีกลูปกระโดดไปมา ซึ่งเป็นการรั่วขึ้นมาจากชั้นของบทที่ 04 และกรณีแย่สุดก็ไม่ใช่กรณีทั่วไป เหมือนที่ตัวแก้ปัญหา SAT (SAT solvers) ในบทที่ 01 แสดงให้เห็นทุกวัน ต้นทุนแบบ asymptotic คือราคาบนป้าย สิ่งที่คุณจ่ายจริงตอนหน้าเคาน์เตอร์ยังขึ้นอยู่กับตัวเครื่องอยู่ดี

Connections · เชื่อมโยง

Below: 01, which decides what is possible before this floor asks what it costs. Beside it: 03, which supplies the counting that turns $\log_2(n!)$ into a genuine information-theoretic bound. Above: 04, where the memory hierarchy decides which constant factors are real; 08, where B-trees are a data structure shaped by the cost of touching a disk; 12, where the training of a model is dominated by exactly the matrix multiplication Strassen attacked.

ชั้นล่าง: บทที่ 01 ซึ่งตัดสินว่าอะไรเป็นไปได้ ก่อนที่ชั้นนี้จะถามว่ามันมีต้นทุนเท่าไร ชั้นข้างเคียง: บทที่ 03 ซึ่งให้การนับที่เปลี่ยน $\log_2(n!)$ ให้กลายเป็นขอบเขตเชิงทฤษฎีสารสนเทศ (information-theoretic bound) ตัวจริง ชั้นบน: บทที่ 04 ที่ลำดับชั้นหน่วยความจำ (memory hierarchy) เป็นตัวตัดสินว่าค่าคงที่ตัวไหนมีผลจริง บทที่ 08 ที่ B-tree เป็นโครงสร้างข้อมูลที่ถูกกำหนดรูปร่างโดยต้นทุนของการแตะดิสก์ บทที่ 12 ที่การเทรนโมเดลถูกครอบงำโดยการคูณเมทริกซ์ตัวเดียวกับที่ Strassen เข้าโจมตี

Open questions · คำถามที่ยังเปิดอยู่

Is the true exponent of matrix multiplication 2? Sixty years of work has moved it from 3 to about 2.371 and stalled far above the obvious lower bound. More broadly: is worst-case analysis still the right instrument, when the algorithms that matter commercially are judged on inputs that are any-

thing but adversarial? And is the catalogue of strategies above complete — or is there a fifth idea, as basic as divide-and-conquer, that nobody has had yet?

> เลขชี้กำลังที่แท้จริงของการคูณเมทริกซ์คือ 2 หรือเปล่า หกสิบปีของความพยายามเลื่อนมันจาก 3 มาอยู่ที่ราว 2.371 แล้วก็ค้างอยู่ห่างจากขอบล่างที่ชัดเจนอยู่มาก มองกว้างขึ้นไปอีก — การวิเคราะห์กรณีแย่สุด (worst-case analysis) ยังเป็นเครื่องมือที่ถูกต้องอยู่ไหม ในเมื่ออัลกอริทึมที่สำคัญในเชิงพาณิชย์ถูกตัดสินบนอินพุตที่ไม่ได้มีลักษณะเป็นปฏิปักษ์เลย และรายการกลยุทธ์ข้างต้นครบถ้วนแล้วหรือยัง — หรือยังมีไอเดียที่ห้าที่พื้นฐานพอๆ กับ divide-and-conquer ที่ยังไม่มีใครคิดถึงมันเลย

Go deeper · อ่านต่อ

— Cormen, Leiserson, Rivest & Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 — the reference; §8.1 has the sorting lower bound in full (MIT Press).

> Cormen, Leiserson, Rivest & Stein, Introduction to Algorithms, 4th ed., MIT Press, 2022 — เล่มอ้างอิงหลัก §8.1 มีขอบล่างของการเรียงลำดับแบบเต็มรูปแบบ (MIT Press).

— Edsger Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik* 1 (1959): 269–271 — three pages, two algorithms, no wasted sentence (PDF).

> Edsger Dijkstra, "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik 1 (1959): 269–271 — สามหน้า สองอัลกอริทึม ไม่มีประโยคไหนฟุ่มเฟือย (PDF).

— Volker Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik* 13 (1969): 354–356 — three pages that started a fifty-year race (Springer).

> Volker Strassen, "Gaussian elimination is not optimal," Numerische Mathematik 13 (1969): 354–356 — สามหน้าที่จุดชนวนการแข่งขันที่ยาวนานห้าสิบปี (Springer).

CHAPTER 03

Information & Coding

สารสนเทศและการเข้ารหัส

This chapter is the bedrock floor beneath every message the tower moves — it fixes what a signal is, and how much of one a channel can carry.

บทนี้คือชั้นรากฐานที่อยู่ใต้ทุกข้อความที่หอคอยลำเลียงไป — มันกำหนดว่าสัญญาณ (signal) คืออะไร และช่องสัญญาณ (channel) หนึ่งรับได้มากแค่ไหน

How much surprise can a channel carry?

ช่องสัญญาณหนึ่งรับความประหลาดใจได้มากแค่ไหน

Before 1948, every engineer working on a noisy line believed the same thing, and it was wrong. Noise corrupts messages; to fight it you shout louder or you repeat yourself. Either way you buy accuracy with power or with time, and the exchange rate looked merciless: to drive the error rate near zero you had to slow the message near to a stop. Reliability and speed were a seesaw, and that was simply the nature of physical channels.

> ก่อนปี 1948 วิศวกรทุกคนที่ทำงานกับสายสัญญาณที่มีสัญญาณรบกวน (noise) เชื่อเรื่องเดียวกัน และมันผิด สัญญาณรบกวนทำให้ข้อความเสียหาย จะสู้กับมันก็ต้องตะโกนดังขึ้นหรือพูดซ้ำ ไม่ว่าทางไหนคุณก็ต้องซื้อความแม่นยำด้วยพลังงานหรือด้วยเวลา และอัตราแลกเปลี่ยนดูโหดร้ายมาก — จะกดอัตราความผิดพลาดให้เข้าใกล้ศูนย์ คุณต้องทำให้ข้อความช้าลงจนเกือบหยุดนิ่ง ความน่าเชื่อถือ (reliability) กับความเร็วเป็นเหมือนไม้กระดกที่ถ่วงกันอยู่ และนั่นก็คือธรรมชาติของช่องสัญญาณทางกายภาพ

Then Claude Shannon, at Bell Labs, published a paper in two parts, in the July and October 1948 issues of the *Bell System Technical Journal*, and the seesaw broke (Shannon, 1948). Every channel, he proved, has a number at-

tached to it — its capacity. Send at any rate *below* that number and you can push the probability of error as close to zero as you like, without slowing down at all. Send above it and nothing saves you: not more power, not more repetition, not any coding scheme anyone will ever invent. Not a seesaw. A speed limit, with a free lunch underneath it.

> แล้ว Claude Shannon ที่ Bell Labs ก็ตีพิมพ์เปเปอร์สองภาคในฉบับเดือนกรกฎาคมและตุลาคมปี 1948 ของ Bell System Technical Journal และไม้กระดกตัวนั้นก็หักลง (Shannon, 1948) เขาพิสูจน์ว่าทุกช่องสัญญาณมีตัวเลขหนึ่งติดอยู่กับมันเสมอ — ความจุช่องสัญญาณ (channel capacity) ของมัน ส่งด้วยอัตราใดก็ตามที่ ต่ำกว่าตัวเลขนั้น คุณจะกดความน่าจะเป็นของความผิดพลาดให้ใกล้ศูนย์แค่ไหนก็ได้ โดยไม่ต้องช้าลงเลยแม้แต่น้อย ส่งเกินตัวเลขนั้นไป ไม่มีอะไรช่วยคุณได้ ไม่ว่าจะเพิ่มพลังงาน เพิ่มการพูดซ้ำ หรือคิดค้นวิธีเข้ารหัส (encoding) แบบไหนก็ตามที่จะมีใครคิดขึ้นในอนาคต ไม่ใช่ไม้กระดกอีกต่อไป แต่เป็นขีดจำกัดความเร็ว ที่มีของฟรีซ่อนอยู่ข้างใต้

To get there, Shannon first had to say what a message *is*.

> กว่าจะไปถึงจุดนั้น Shannon ต้องบอกก่อนว่าข้อความคืออะไร

Information is surprise, not meaning. His second paragraph throws away the thing everyone else cared about: "These semantic aspects of communication are irrelevant to the engineering problem." What matters is only that a message is a *selection* from a set of possible messages, and the engineering job is to reproduce the selection at the far end. That reframing gives the unit: "If the base 2 is used the resulting units may be called binary digits, or more briefly *bits*, a word suggested by J. W. Tukey." One fair coin flip is one bit; one fair die is about 2.6. A message you could have predicted carries none, however long it is, and a message you could not carries a great deal, however short. Entropy — Shannon's H — is just the average surprise

per symbol from a source. Harry Nyquist and Ralph Hartley had circled this in the same journal in 1924 and 1928; Shannon closed the circle by making it a measure.

> สารสนเทศ (information) คือความประหลาดใจ ไม่ใช่ความหมาย ย่อหน้าที่สองของเขาทิ้งสิ่งที่ทุกคนอื่นให้ความสำคัญไปเลย — "แง่มุมเชิงความหมาย (semantic) เหล่านี้ของการสื่อสารไม่เกี่ยวข้องกับโจทย์ทางวิศวกรรม" สิ่งที่สำคัญมีเพียงว่าข้อความหนึ่งคือการเลือกจากชุดข้อความที่เป็นไปได้ทั้งหมด และงานทางวิศวกรรมคือการสร้างการเลือกนั้นซ้ำที่ปลายทาง การตีกรอบใหม่นี้ให้หน่วยวัดออกมา — "ถ้าใช้ฐาน 2 หน่วยที่ได้จะเรียกว่า binary digit หรือเรียกสั้นๆ ว่า bit คำที่ J. W. Tukey เป็นคนเสนอ" การโยนเหรียญที่เที่ยงตรงหนึ่งครั้งมีค่าหนึ่งบิต (bit) ลูกเต๋าที่เที่ยงตรงหนึ่งลูกมีค่าราว 2.6 บิต ข้อความที่คุณทายได้อยู่แล้วไม่มีสารสนเทศเลยไม่ว่าจะยาวแค่ไหน ส่วนข้อความที่คุณทายไม่ได้มีสารสนเทศมากไม่ว่าจะสั้นแค่ไหน เอนโทรปี (entropy) — ตัว H ของ Shannon — ก็คือค่าเฉลี่ยของความประหลาดใจต่อสัญลักษณ์หนึ่งตัวจากแหล่งข้อมูลหนึ่งแหล่ง Harry Nyquist และ Ralph Hartley เคยวนอยู่รอบไอเดียนี้ในวารสารเดียวกันเมื่อปี 1924 และ 1928 Shannon ปิดวงนั้นด้วยการทำให้มันกลายเป็นหน่วยวัดที่แท้จริง

Entropy is the floor of compression. Once surprise has a number, the number binds. Shannon's noiseless coding result (Theorem 9 in the paper) says a source of entropy H cannot be encoded in fewer than H bits per symbol on average, and can be encoded in arbitrarily close to H. English is nowhere near the 4.7 bits per character that a 26-letter alphabet would allow, because letters predict each other — q is followed by u, and you already knew how this sentence would end. That gap between the alphabet's capacity and the language's entropy is exactly what every compressed archive eats. Compression is therefore not a trick performed on data; it is the re-

moval of predictability. Which explains two things everyone has noticed: a compressed file does not compress again, and truly random data does not compress at all.

> เอนโทรปีคือพื้นชั้นล่างสุดของการบีบอัด (compression) พอความประหลาดใจมีตัวเลขกำกับแล้ว ตัวเลขนั้นก็ผูกมัด ผลลัพธ์เรื่องการเข้ารหัสแบบไร้สัญญาณรบกวนของ Shannon (Theorem 9 ในเปเปอร์) บอกว่าแหล่งข้อมูลที่มีเอนโทรปี H ไม่สามารถเข้ารหัสให้ต่ำกว่า H บิตต่อสัญลักษณ์โดยเฉลี่ยได้ และสามารถเข้ารหัสให้ใกล้เคียง H ได้มากเท่าที่ต้องการ ภาษาอังกฤษยังห่างไกลจาก 4.7 บิตต่อตัวอักษรที่ตัวอักษร 26 ตัวจะอนุญาตให้มีได้มาก เพราะตัวอักษรทำนายกันเองได้ — q ตามด้วย u เสมอ และคุณก็รู้อยู่แล้วว่าประโยคนี้จะจบลงยังไง ช่องว่างระหว่างความจุของตัวอักษรกับเอนโทรปีของภาษานั้นแหละคือสิ่งที่ไฟล์บีบอัดทุกไฟล์กินเข้าไป ฉะนั้นการบีบอัดจึงไม่ใช่กลเม็ดที่ทำกับข้อมูล แต่คือการเอาความที่ทายได้ออกไป ซึ่งอธิบายสองเรื่องที่ทุกคนเคยสังเกตเห็น — ไฟล์ที่บีบอัดแล้วบีบซ้ำอีกไม่ได้ และข้อมูลที่สุ่มจริงๆ บีบอัดไม่ได้เลย

Capacity is a speed limit, and redundancy is how you drive at it. Shannon's noisy-channel theorem (Theorem 11) is the free lunch stated formally, and Theorem 17 puts a number on it for the commonest case: "The capacity of a channel of band W perturbed by white thermal noise of power N when the average transmitter power is P" is $C = W \log(1 + P/N)$ bits per second (BSTJ 27, no. 4). Note what this licenses. Repeating a message three times is redundancy spent stupidly — it triples your cost for a modest gain. Shannon proved that redundancy spent *cleverly* buys near-perfect reliability at almost no cost in rate. He did not, however, say how. His proof shows that good codes exist by showing that a randomly chosen code is probably good — an existence proof that handed engineers a construction project lasting fifty years.

> ความจุคือขีดจำกัดความเร็ว และความซ้ำซ้อน (redundancy) คือวิธีที่คุณขับไปให้ถึงมัน ทฤษฎีบทช่องสัญญาณที่มีสัญญาณรบกวนของ Shannon (Theorem 11) คือของฟรีที่ถูกกล่าวไว้อย่างเป็นทางการ และ Theorem 17 ใส่ตัวเลขให้กับกรณีที่พบบ่อยที่สุด — "ความจุของช่องสัญญาณย่านความถี่ W ที่ถูกรบกวนโดยสัญญาณ

> รบกวนความร้อนสีขาวกำลัง N เมื่อกำลังส่งเฉลี่ยคือ P" คือ C = W log(1 + P/N) บิตต่อวินาที (BSTJ 27, no. 4) สังเกตว่าสิ่งนี้เปิดทางให้อะไรบ้าง การพูดข้อความซ้ำสามครั้งคือการใช้ความซ้ำซ้อนแบบโง่ๆ — มันเพิ่มต้นทุนเป็นสามเท่าเพื่อแลกกับกำไรเพียงน้อยนิด Shannon พิสูจน์ว่าความซ้ำซ้อนที่ใช้อย่างฉลาดซื้อความน่าเชื่อถือที่เกือบสมบูรณ์แบบมาได้โดยแทบไม่เสียต้นทุนด้านอัตราเลย แต่เขาไม่ได้บอกว่าทำยังไง บทพิสูจน์ของเขาแสดงว่ารหัสที่ดีมีอยู่จริง ด้วยการแสดงว่ารหัสที่เลือกแบบสุ่มนั้นน่าจะดี — เป็นบทพิสูจน์เชิงมีอยู่จริง (existence proof) ที่ยื่นโครงการก่อสร้างให้วิศวกรทำต่อไปอีกห้าสิบปี

The construction project. Richard Hamming had a personal stake. Working weekends on Bell Labs' relay computer in 1947, with no operator present, he would submit a job on Friday and return Monday to find that the machine had detected a card-reading error, dumped his work, and moved on. Twice in a row. "Dammit," he recalled thinking, "if a machine can find out that there is an error, why can't it locate where it is and change the setting of the relay from one to zero or zero to one?" (IEEE Computer Society). The answer became the Hamming codes (*BSTJ* 29, no. 2, April 1950: 147–160) — parity bits arranged so that the pattern of failed checks spells out the position of the flipped bit in binary. Detection had become correction.

> โครงการก่อสร้างนั้น Richard Hamming มีเรื่องส่วนตัวเข้ามาเกี่ยวข้องด้วย ตอนทำงานวันเสาร์อาทิตย์กับเครื่องคำนวณแบบรีเลย์ของ Bell Labs ในปี 1947 โดยไม่มีเจ้าหน้าที่ควบคุมอยู่เลย เขาส่งงานเข้าไปวันศุกร์แล้วกลับมาวันจันทร์ก็พบว่าเครื่องตรวจเจอข้อผิดพลาดในการอ่านบัตร ทิ้งงานของเขาไป แล้วเดินหน้าต่อ สองครั้งติดกัน "แม่งเอ๊ย" เขาเล่าว่าคิดแบบนั้น "ถ้าเครื่องหาเจอว่ามีข้อผิดพลาดได้ ทำไมมันถึงหาตำแหน่งที่ผิดแล้วเปลี่ยนค่ารีเลย์จากหนึ่งเป็นศูนย์หรือศูนย์เป็นหนึ่งเองไม่ได้ล่ะ" (IEEE Computer Society) คำตอบกลายมาเป็น Hamming code (BSTJ 29, no. 2, April 1950: 147–160) — บิตพาริตี (parity bit) ที่จัดเรียงไว้ให้รูปแบบของการตรวจสอบที่ล้มเหลวสะกดตำแหน่งของบิตที่พลิกกลับออกมาเป็นเลขฐานสอง การตรวจจับ (detection) กลายเป็นการแก้ไข (correction) ไปแล้ว

Then the codes got good. Reed and Solomon's 1960 construction over finite fields (*Journal of the SIAM* 8: 300–304) survives bursts of damage rather than isolated flips, which is what deep space demanded: JPL launched Voyager 2 toward Uranus knowing its Reed–Solomon decoders did not yet exist, "gambl[ing] that by the time Voyager II would reach Uranus in 1986, decoding algorithms and equipment would be both available and perfected. They were correct!" (NASA TM 102162). Robert Gallager had meanwhile invented low-density parity-check codes in his 1960 MIT thesis, published them in 1962 (*IRE Transactions on Information Theory* 8: 21–28), and watched them vanish — too expensive to decode on the hardware of the day. They were rediscovered by MacKay and Neal in 1995–96, alongside the turbo codes of Berrou, Glavieux and Thitimajshima (*Proc. IEEE ICC*, 1993), and the wall Shannon had drawn was finally reached: by 2001 an irregular LDPC code came within 0.0045 decibels of capacity (Chung, Forney, Richardson & Urbanke, *IEEE Communications Letters* 5, 2001). Your phone uses both descendants — LDPC for 5G data channels, and Erdal Arıkan's polar codes, published in 2009 and standardized within a decade, for the control channels.

> แล้วรหัสก็เริ่มดีขึ้นเรื่อยๆ การสร้างของ Reed และ Solomon ในปี 1960 บนฟิลด์จำกัด (finite field) (Journal of the SIAM 8: 300–304) ทนต่อความเสียหายที่มาเป็นช่วง (burst) ได้ดีกว่าการพลิกบิตแบบเดี่ยวๆ ซึ่งเป็นสิ่งที่อวกาศห้วงลึกต้องการพอดี — JPL ปล่อยยาน Voyager 2 มุ่งสู่ดาวยูเรนัสทั้งที่ตัวถอดรหัส Reed–Solomon ของมันยังไม่มีอยู่จริง โดย "พนันว่าเมื่อถึงเวลาที่ Voyager II ไปถึงดาวยูเรนัสในปี 1986 อัลกอริทึมและอุปกรณ์ถอดรหัสจะพร้อมใช้งานและสมบูรณ์แล้ว ซึ่งพวกเขาคิดถูก!" (NASA TM 102162) ในระหว่างนั้น Robert Gallager คิดค้น low-density parity-check code (LDPC) ขึ้นในวิทยานิพนธ์ปริญญาเอกที่ MIT ปี 1960 ตีพิมพ์มันในปี 1962 (IRE Transactions on Information Theory 8: 21–28) แล้วก็เฝ้าดูมันเลือนหายไป — เพราะแพงเกินกว่าจะถอดรหัสได้บนฮาร์ดแวร์ยุคนั้น มันถูกค้นพบใหม่โดย MacKay และ Neal ในปี 1995–96 พร้อมกับ turbo code ของ Berrou, Glavieux และ Thitimajshima (Proc. IEEE ICC, 1993) และกำแพงที่ Shannon วางไว้ก็ถูกไปถึงในที่สุด — ภายในปี 2001 LDPC แบบ irregular เข้าใกล้ความจุได้ในระยะห่างเพียง 0.0045 เดซิเบล (Chung, Forney, Richardson &

> Urbanke, IEEE Communications Letters 5, 2001) โทรศัพท์ของคุณใช้ทายาททั้งสองสาย — LDPC สำหรับช่องข้อมูล 5G และ polar code ของ Erdal Arıkan ซึ่งตีพิมพ์ในปี 2009 และถูกกำหนดเป็นมาตรฐานภายในสิบปี สำหรับช่องควบคุม

And the floor's silence is its leak. Everything above is indifferent to what a message says. Shannon's disclaimer was a precondition for the theory's power, not an oversight — but it means this floor can price the transmission of a diagnosis and of a spam message identically, and has nothing to say about which was worth sending. Every floor above pays for that silence in its own currency: cryptography (11) must add who sent it, databases (08) must add what it means, and machine learning (12) spends its whole existence trying to recover meaning from statistics — while quietly using Shannon's entropy as its loss function.

> และความเงียบของชั้นนี้ก็คือจุดที่มันรั่ว (leak) ทุกอย่างข้างต้นไม่สนใจเลยว่าข้อความพูดว่าอะไร คำปฏิเสธความรับผิดชอบของ Shannon เป็นเงื่อนไขที่จำเป็นต่อพลังของทฤษฎีนี้ ไม่ใช่การมองข้าม — แต่มันหมายความว่าชั้นนี้ตั้งราคาการส่งผลวินิจฉัยโรคกับการส่งข้อความสแปมเท่ากันได้ และไม่มีอะไรจะพูดเลยว่าอันไหนคุ้มค่าที่จะส่งกว่ากัน ทุกชั้นที่อยู่เหนือขึ้นไปต้องจ่ายค่าความเงียบนี้ด้วยสกุลเงินของตัวเอง — วิทยาการรหัสลับ (cryptography, บทที่ 11) ต้องเติมว่าใครเป็นคนส่ง ฐานข้อมูล (บทที่ 08) ต้องเติมว่ามันหมายความว่าอะไร และ machine learning (บทที่ 12) ใช้ชีวิตทั้งหมดของมันพยายามกู้ความหมายกลับคืนมาจากสถิติ — พร้อมกับแอบใช้เอนโทรปีของ Shannon เป็น loss function ของมันไปด้วย

Connections · เชื่อมโยง

Beside this floor: 02, whose $\Omega(n \log n)$ sorting bound is this chapter's counting argument in disguise — $\log_2(n!)$ is the number of bits needed to name one ordering. Above: 09, where TCP builds reliability out of an unreliable channel — the same bargain, made at a higher floor; 08, where compression

and checksums guard stored data; 11, where a cipher's strength is counted in bits; 12, where a model that predicts the next token well is, by the argument above, a compressor.

> ชั้นข้างเคียง: บทที่ 02 ซึ่งขอบล่าง Ω(n log n) ของการเรียงลำดับก็คือข้อโต้แย้งเรื่องการนับของบทนี้แหละที่แฝงตัวมา — $\log_2(n!)$ คือจำนวนบิตที่ต้องใช้ในการระบุลำดับหนึ่งแบบ ชั้นบน: บทที่ 09 ที่ TCP สร้างความน่าเชื่อถือขึ้นจากช่องสัญญาณที่ไม่น่าเชื่อถือ — ข้อตกลงเดียวกัน แต่ทำในชั้นที่สูงกว่า บทที่ 08 ที่การบีบอัดและ checksum คุ้มครองข้อมูลที่จัดเก็บไว้ บทที่ 11 ที่ความแข็งแกร่งของ cipher ถูกนับเป็นบิต บทที่ 12 ที่โมเดลที่ทายโทเคนถัดไปได้ดี ตามข้อโต้แย้งข้างต้นก็คือตัวบีบอัดชนิดหนึ่งนั่นเอง

Open questions · คำถามที่ยังเปิดอยู่

Shannon deliberately excluded meaning — and no accepted theory of semantic information has replaced what he set aside, though the question is now live again because language models make it practical. Is compression the same thing as understanding, or merely correlated with it? And on the physical side: how close to capacity is close enough to stop, given that the last fraction of a decibel now costs more energy in decoding than it saves in transmission?

> Shannon ตัดความหมาย (meaning) ออกไปโดยตั้งใจ — และยังไม่มีทฤษฎีสารสนเทศเชิงความหมาย (semantic information) ที่ได้รับการยอมรับมาแทนที่สิ่งที่เขาตัดทิ้งไป แม้คำถามนี้จะกลับมามีชีวิตอีกครั้งเพราะโมเดลภาษาทำให้มันใช้ได้จริงในทางปฏิบัติ การบีบอัดคือสิ่งเดียวกับความเข้าใจหรือแค่มีความสัมพันธ์กันเฉยๆ และในแง่กายภาพ — ใกล้ความจุแค่ไหนถึงจะเรียกว่าใกล้พอที่จะหยุด ในเมื่อเดซิเบลเศษเสี้ยวสุดท้ายตอนนี้กินพลังงานในการถอดรหัสมากกว่าที่มันประหยัดได้ตอนส่งสัญญาณ

Go deeper · อ่านต่อ

- Claude Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal* 27 (1948): 379–423, 623–656 — still the clearest exposition of its own ideas (PDF).

> Claude Shannon, "A Mathematical Theory of Communication," Bell System Technical Journal 27 (1948): 379–423, 623–656 — ยังคงเป็นการอธิบายไอเดียของตัวเองที่ชัดเจนที่สุด (PDF).

- David MacKay, *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003 — free from the author's site, and the book that connects this floor to chapter 12 (inference.org.uk).

> David MacKay, Information Theory, Inference, and Learning Algorithms, Cambridge University Press, 2003 — ฟรีจากเว็บไซต์ของผู้เขียนเอง และเป็นหนังสือที่เชื่อมชั้นนี้เข้ากับบทที่ 12 (inference.org.uk).

- Richard Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal* 29, no. 2 (April 1950): 147–160 — the first working answer to Shannon's existence proof (scan).

> Richard Hamming, "Error Detecting and Error Correcting Codes," Bell System Technical Journal 29, no. 2 (April 1950): 147–160 — คำตอบที่ใช้งานได้จริงชิ้นแรกต่อบทพิสูจน์เชิงมีอยู่จริงของ Shannon (scan).

CHAPTER 04

Computer Architecture

สถาปัตยกรรมคอมพิวเตอร์

This chapter is the floor where physics becomes logic — the lowest floor this book visits, and the one every floor above it stands on.

บทนี้คือชั้นที่ฟิสิกส์กลายเป็นตรรกะ — ชั้นล่างสุดที่หนังสือเล่มนี้ไปเยือน และเป็นชั้นที่ทุกชั้นเหนือขึ้นไปยืนอยู่บน

How do you get arithmetic out of electricity?

ทำอย่างไรจึงดึงเลขคณิตออกมาจากไฟฟ้าได้

A single Nvidia Blackwell B200 processor holds 208 billion transistors. A transistor does exactly one thing: it conducts, or it does not. It cannot add. It has no notion of a number, or of quantity. Yet somewhere between that switch and the arithmetic running underneath this sentence, meaning appears — and it appears in no component. This chapter is about where it actually is.

> โปรเซสเซอร์ Nvidia Blackwell B200 ตัวเดียวมีทรานซิสเตอร์ (transistor) 208 billion ตัว ทรานซิสเตอร์ทำได้แค่อย่างเดียว คือนำไฟฟ้า หรือไม่นำ มันบวกเลขไม่เป็น ไม่มีความคิดเรื่องตัวเลขหรือปริมาณอยู่ในตัวเลยสักนิด แต่ระหว่างสวิตช์ตัวนั้นกับเลขคณิตที่กำลังทำงานอยู่เบื้องหลังประโยคนี้ ความหมายกลับปรากฏขึ้นมา — และมันไม่ได้อยู่ในชิ้นส่วนไหนชิ้นเดียว บทนี้ว่าด้วยเรื่องที่ว่าจริงๆ แล้วมันอยู่ตรงไหนกันแน่

The answer is composition, and it is older than the computer. In the master's thesis he submitted to MIT in 1937, Claude Shannon showed that the two-valued algebra George Boole had built to formalize logic described, exactly, the behavior of circuits made of switches (MIT's copy). Two

switches in series are AND; in parallel, OR. That correspondence is the hinge of the entire tower: a circuit can be *designed* as an argument rather than tinkered into working. After it, the climb is short and mechanical. Five gates — two XOR, two AND, one OR — make a full adder, a circuit that adds two bits plus a carry. Chain sixty-four of them and the machine adds sixty-four-bit integers. Nothing was introduced along the way except arrangement. *Arithmetic is not in the electricity. It is in the shape.*

> คำตอบคือการประกอบกัน (composition) และมันเก่าแก่กว่าคอมพิวเตอร์เสียอีก ในวิทยานิพนธ์ปริญญาโทที่ Claude Shannon ยื่นให้ MIT ในปี 1937 เขาแสดงให้เห็นว่าพีชคณิตสองค่าที่ George Boole สร้างขึ้นเพื่อทำให้ตรรกะเป็นรูปนัยนั้น อธิบายพฤติกรรมของวงจรที่สร้างจากสวิตช์ได้ตรงเป๊ะ (ต้นฉบับของ MIT) สวิตช์สองตัวต่ออนุกรมกันคือ AND ต่อขนานกันคือ OR ความสอดคล้องนี้คือบานพับของหอคอย (tower) ทั้งหลัง วงจรสามารถ ออกแบบ ขึ้นมาในฐานะข้อโต้แย้งหนึ่ง แทนที่จะลองผิดลองถูกจนใช้งานได้ หลังจากนั้นการปีนก็สั้นและเป็นกลไกล้วนๆ เกตตรรกะ (logic gate) ห้าตัว — XOR สองตัว AND สองตัว OR หนึ่งตัว — ประกอบกันเป็นวงจรบวกแบบเต็ม (full adder) หนึ่งตัว ซึ่งบวกบิตสองตัวบวกตัวทดหนึ่งตัว ต่อวงจรแบบนี้เข้าด้วยกัน 64 ตัว เครื่องก็บวกจำนวนเต็ม 64 บิตได้ ไม่มีอะไรถูกเติมเข้ามาระหว่างทางเลยนอกจากการจัดเรียง เลขคณิตไม่ได้อยู่ในไฟฟ้า มันอยู่ในรูปทรง

The second idea arrived in a document nobody finished. In June 1945, Herman Goldstine circulated twenty-four copies of a 101-page draft by John von Neumann, the *First Draft of a Report on the EDVAC* (scan). Its central move — the stored program — was to keep a machine's instructions in the same memory as its data. This sounds like a filing decision. It is the reason the upper tower exists: if a program is just numbers in memory, then a program can read, write, and generate other programs — which is what every tool in chapter 06 does. It carried von Neumann's name alone, and caused sixty years of argument: J. Presper Eckert and John Mauchly held that the idea came out of group meetings at the Moore School, and the report's wide

circulation later counted as prior art that helped invalidate the ENIAC patent in 1973. The lesson survives the quarrel: the machine's most consequential feature was a decision about what to *not* keep separate.

> ไอเดียที่สองมาจากเอกสารที่ไม่มีใครเขียนจบ ในเดือนมิถุนายน 1945 Herman Goldstine แจกจ่ายสำเนา 24 ชุดของร่างเอกสาร 101 หน้าที่ John von Neumann เขียน ชื่อ First Draft of a Report on the EDVAC (สแกนต้นฉบับ) แกนกลางของมันคือโปรแกรมที่เก็บในหน่วยความจำ (stored program) — เก็บคำสั่งเครื่องไว้ในหน่วยความจำก้อนเดียวกับข้อมูล ฟังดูเหมือนแค่การตัดสินใจเรื่องจัดเก็บเอกสาร แต่มันคือเหตุผลที่หอคอยชั้นบนทั้งหมดมีอยู่ได้ — ถ้าโปรแกรมเป็นแค่ตัวเลขในหน่วยความจำ โปรแกรมก็อ่าน เขียน และสร้างโปรแกรมอื่นได้ ซึ่งคือสิ่งที่เครื่องมือทุกชิ้นในบทที่ 06 ทำ เอกสารนี้มีชื่อ von Neumann อยู่คนเดียว และก่อให้เกิดการโต้เถียงนาน 60 ปี J. Presper Eckert และ John Mauchly ยืนยันว่าไอเดียนี้มาจากการประชุมกลุ่มที่ Moore School และการแจกจ่ายเอกสารในวงกว้างในเวลาต่อมาก็ถูกนับเป็นหลักฐานสิ่งที่มีมาก่อน (prior art) ที่ช่วยให้สิทธิบัตร ENIAC เป็นโมฆะในปี 1973 บทเรียนที่รอดจากการทะเลาะกันคือ คุณสมบัติที่สำคัญที่สุดของเครื่องคือการตัดสินใจว่าจะ ไม่ แยกอะไรออกจากกัน

The third idea is the one that gives this floor its name. On 7 April 1964 IBM announced System/360, a family of machines of differing size, speed, and price built to run the same programs. Gene Amdahl, Gerrit Blaauw, and Frederick Brooks defined architecture as "the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation" (Amdahl, Blaauw & Brooks, 1964). That distinction is the floor's surface. An instruction set is a *contract*: the hardware promises what each instruction means, and promises nothing about how it will keep the promise. Behind an unchanged contract, engineers have replaced nearly everything — pipelining instructions so several are in flight at once, executing them out of order, guessing at branches and

rolling back wrong guesses. The programmer's side of the contract never mentions any of it. This is why software outlives the hardware it was written for.

> ไอเดียที่สามคือไอเดียที่ให้ชื่อชั้นนี้ วันที่ 7 เมษายน 1964 IBM ประกาศเปิดตัว System/360 ตระกูลเครื่องที่ขนาด ความเร็ว และราคาต่างกัน แต่สร้างมาให้รันโปรแกรมชุดเดียวกันได้ Gene Amdahl, Gerrit Blaauw และ Frederick Brooks นิยาม architecture ว่าคือ "คุณลักษณะของระบบตามที่โปรแกรมเมอร์มองเห็น กล่าวคือโครงสร้างเชิงแนวคิดและพฤติกรรมเชิงหน้าที่ ต่างจากการจัดการ data flow และการควบคุม การออกแบบเชิงตรรกะ และการนำไปปฏิบัติจริงทางกายภาพ" (Amdahl, Blaauw & Brooks, 1964) ความแตกต่างนั้นคือผิว (surface) ของชั้นนี้ ชุดคำสั่ง (instruction set) หนึ่งชุดคือ สัญญา — ฮาร์ดแวร์สัญญาว่าคำสั่งเครื่องแต่ละตัวหมายถึงอะไร และไม่สัญญาอะไรเลยว่าจะรักษาสัญญานั้นด้วยวิธีไหน อยู่เบื้องหลังสัญญาที่ไม่เคยเปลี่ยนนั้น วิศวกรเปลี่ยนแทบทุกอย่าง — ทำ pipelining คำสั่งเครื่องให้หลายตัวอยู่ระหว่างดำเนินการพร้อมกัน สั่งให้ทำงานนอกลำดับ เดาทางแยกของโปรแกรมแล้วย้อนกลับเมื่อเดาผิด ฝั่งโปรแกรมเมอร์ของสัญญาไม่เคยพูดถึงเรื่องพวกนี้เลยสักคำ นี่คือเหตุผลที่ซอฟต์แวร์อยู่ได้นานกว่าฮาร์ดแวร์ที่มันถูกเขียนขึ้นมาด้วยซ้ำ

But the contract has fine print, and the fine print is where this floor leaks. In 1995 William Wulf and Sally McKee published a two-page note called "Hitting the Memory Wall: Implications of the Obvious". Their arithmetic used numbers everyone already had: memory was getting about 7 percent faster a year, processors about 80. Compounded, that gap projected an average memory access costing roughly 99 processor cycles by 2010 — a machine spending most of its life waiting. What has held the wall back is not a theorem but an empirical regularity: **locality**. Programs touch the same data again soon, and data near what they just touched. So architects built a hierarchy of small fast memories in front of large slow ones and bet on that habit — the most exploited empirical regularity in computing, and it usually wins. When it loses, two loops identical in logic — same operations, same results — can differ enormously in time, for reasons the instruction set does

not mention and the source code does not show. Knowing that such a gap is a *memory* problem, three floors down, rather than an algorithm problem, is a large fraction of what practical expertise in computing means.

> แต่สัญญามีตัวหนังสือตัวเล็ก และตัวหนังสือตัวเล็กนั้นคือจุดที่ชั้นนี้รั่ว ในปี 1995 William Wulf และ Sally McKee ตีพิมพ์บันทึกสั้นสองหน้าชื่อ "Hitting the Memory Wall: Implications of the Obvious" เลขคณิตของพวกเขาใช้ตัวเลขที่ทุกคนมีอยู่แล้ว หน่วยความจำเร็วขึ้นราวปีละ 7 เปอร์เซ็นต์ ส่วนโปรเซสเซอร์เร็วขึ้นราว 80 เปอร์เซ็นต์ พอทบต้นกันไปเรื่อยๆ ช่องว่างนี้ทำนายว่าภายในปี 2010 การเข้าถึงหน่วยความจำโดยเฉลี่ยจะกิน cycle ของโปรเซสเซอร์ไปราว 99 cycle — เครื่องที่ใช้ชีวิตส่วนใหญ่ไปกับการรอ สิ่งที่กันกำแพงนี้ไว้ไม่ใช่ทฤษฎีบทอะไร แต่คือความสม่ำเสมอเชิงประจักษ์อย่างหนึ่ง คือ **locality** โปรแกรมมักแตะข้อมูลเดิมซ้ำในเวลาไม่นาน และแตะข้อมูลที่อยู่ใกล้กับสิ่งที่เพิ่งแตะไป สถาปนิกจึงสร้างลำดับชั้นหน่วยความจำ (memory hierarchy) ที่มีหน่วยความจำเล็กเร็วอยู่หน้าหน่วยความจำใหญ่ช้า แล้วพนันไปกับนิสัยนั้น — ความสม่ำเสมอเชิงประจักษ์ที่ถูกใช้ประโยชน์มากที่สุดในวงการคอมพิวเตอร์ และมันมักจะชนะ เมื่อมันแพ้ ลูปสองลูปที่ตรรกะเหมือนกันเป๊ะ — ปฏิบัติการเดียวกัน ผลลัพธ์เดียวกัน — อาจใช้เวลาต่างกันมหาศาล ด้วยเหตุผลที่ instruction set ไม่พูดถึง และซอร์สโค้ดก็ไม่แสดงให้เห็น การรู้ว่าช่องว่างแบบนั้นเป็นปัญหาเรื่อง หน่วยความจำ ที่อยู่ลึกลงไปสามชั้น ไม่ใช่ปัญหาเรื่องอัลกอริทึม คือความเชี่ยวชาญเชิงปฏิบัติส่วนใหญ่ในวงการคอมพิวเตอร์เลยทีเดียว

The last idea is that this floor stopped being free. Gordon Moore's 1965 magazine article observed that the component count on a chip was doubling every year, a rate he revised to every two years in 1975; the famous "eighteen months" appears in neither. What made shrinking transistors *feel* free was a second rule from Robert Dennard and colleagues in 1974: scale dimensions and voltage together and power density stays constant, so more transistors cost no more heat (Dennard et al., 1974). Around 2005 that stopped working — voltages could not keep falling without leakage — and the whole tower felt it, because for four decades every floor above had been getting faster while doing nothing. Architects responded by spending transistors differently: first on more cores, then on chips that do less but do it better. Google's first Tensor Processing Unit, described at ISCA in 2017,

dropped nearly everything a general processor offers and reported, on its intended workload, 25 to 29 times the operations per watt of the contemporary Nvidia K80 GPU and 41 to 83 times a contemporary Intel CPU — the spread depending on how power is accounted (Jouppi et al., 2017). John Hennessy and David Patterson, who received the 2017 Turing Award for the quantitative approach that made such comparisons possible, called the result a new golden age for computer architecture. Note what changed: speed used to arrive from below as a gift; now it is negotiated.

> ไอเดียสุดท้ายคือชั้นนี้เลิกฟรีแล้ว บทความในนิตยสารปี 1965 ของ Gordon Moore สังเกตว่าจำนวนชิ้นส่วนบนชิปเพิ่มเป็นสองเท่าทุกปี อัตราที่เขาแก้ไขเป็นทุกสองปีใน ปี 1975 ส่วนคำที่โด่งดังอย่าง "สิบแปดเดือน (eighteen months)" นั้นไม่ปรากฏอยู่ ในทั้งสองฉบับเลย สิ่งที่ทำให้การย่อทรานซิสเตอร์ให้เล็กลง รู้สึก เหมือนฟรี คือกฎอีก ข้อจาก Robert Dennard และเพื่อนร่วมงานในปี 1974 — ย่อขนาดและแรงดันไฟไป พร้อมกัน แล้วความหนาแน่นของกำลังไฟจะคงที่ ทรานซิสเตอร์ที่เพิ่มขึ้นจึงไม่ทำให้ ความร้อนเพิ่มขึ้นตาม (Dennard et al., 1974) ราวปี 2005 กฎนี้หยุดใช้ได้ — แรงดัน ไฟลดต่อไปไม่ได้อีกโดยไม่เกิดการรั่วไหลของกระแส — และหอคอยทั้งหลังก็รู้สึกถึง มัน เพราะตลอดสี่ทศวรรษที่ผ่านมา ทุกชั้นเหนือขึ้นไปเร็วขึ้นเรื่อยๆ โดยไม่ต้องทำ อะไรเลย สถาปนิกตอบโต้ด้วยการใช้ทรานซิสเตอร์ไปในทางอื่น ก่อนอื่นคือเพิ่ม จำนวนคอร์ (core) แล้วจึงไปที่ชิปซึ่งทำงานได้น้อยลงแต่ทำได้ดีขึ้น Tensor Processing Unit ตัวแรกของ Google ที่บรรยายไว้ในงาน ISCA ปี 2017 ตัดแทบทุก อย่างที่โปรเซสเซอร์ทั่วไปมีทิ้งไป และรายงานว่า บนงานที่ตั้งใจให้มันทำ มันให้ จำนวนปฏิบัติการต่อวัตต์มากกว่า Nvidia K80 GPU ร่วมสมัย 25 ถึง 29 เท่า และ มากกว่า Intel CPU ร่วมสมัย 41 ถึง 83 เท่า — ช่วงที่ต่างกันขึ้นอยู่กับวิธีนับกำลังไฟ (Jouppi et al., 2017) John Hennessy และ David Patterson ผู้ได้รับรางวัล Turing Award ปี 2017 จากแนวทางเชิงปริมาณที่ทำให้การเปรียบเทียบแบบนี้เป็นไปได้ เรียกผลลัพธ์นี้ว่ายุคทองใหม่ของสถาปัตยกรรมคอมพิวเตอร์ สังเกตสิ่งที่เปลี่ยนไป — ความเร็วเคยมาจากชั้นล่างในฐานะของขวัญ ตอนนี้มันต้องต่อรองเอา

The people are unusually entangled with the ideas. Amdahl architected System/360 and then left IBM in 1970 to found a company building machines plug-compatible with the very contract he had drafted — the architecture

outlived its author's employment, which is the strongest proof it was a real interface. Dennard invented the one-transistor memory cell in 1966 (patent granted June 1968) and then wrote the scaling rule that let processors outrun it: one man is responsible both for the memory that is too slow and for the reason it matters.

> ผู้คนพัวพันกับไอเดียเหล่านี้อย่างผิดปกติ Amdahl เป็นสถาปนิกของ System/360 แล้วลาออกจาก IBM ในปี 1970 ไปตั้งบริษัทสร้างเครื่องที่เข้ากันได้โดยตรง (plug-compatible) กับสัญญาเดียวกับที่ตัวเองร่างไว้ — architecture อยู่ได้นานกว่าการเป็นพนักงานของผู้เขียนมันเสียอีก ซึ่งเป็นบทพิสูจน์ที่หนักแน่นที่สุดว่ามันคืออินเทอร์เฟซที่มีอยู่จริง Dennard คิดค้นเซลล์หน่วยความจำแบบทรานซิสเตอร์เดียวในปี 1966 (สิทธิบัตรได้รับอนุมัติมิถุนายน 1968) แล้วก็เขียนกฎการย่อขนาดที่ทำให้โปรเซสเซอร์วิ่งแซงหน้ามันไปเอง — คนคนเดียวรับผิดชอบทั้งหน่วยความจำที่ช้าเกินไป และเหตุผลที่มันสำคัญ

Connections · เชื่อมโยง

Below this chapter, in this book, there is nothing — 01 through 03 are about what can be computed and what information is, questions that hold for any machine or none. Above it, 05 takes this single machine and makes it behave like many, using the protection and timer hardware this floor provides; 06 turns the instruction-set contract into something a person can write in. Chapter 02's cost model quietly assumes memory that answers in constant time, which this floor does not supply; read the two against each other and the disagreement is instructive. Chapter 12 is where specialization now points, and chapter 14 asks what the energy wall does to all of it.

> ใต้บทนี้ในหนังสือเล่มนี้ไม่มีอะไรอีกแล้ว — บทที่ 01 ถึง 03 ว่าด้วยสิ่งที่คำนวณได้และสารสนเทศคืออะไร คำถามที่เป็นจริงไม่ว่าจะเป็นเครื่องไหนหรือไม่มีเครื่องเลยก็ตาม เหนือขึ้นไป บทที่ 05 เอาเครื่องเดียวนี้มาทำให้มันทำตัวเหมือนมีหลายเครื่องโดยใช้ฮาร์ดแวร์ป้องกันและตัวจับเวลาที่ชั้นนี้ยื่นให้ บทที่ 06 เปลี่ยนสัญญา instruction set ให้กลายเป็นสิ่งที่คนเขียนได้ โมเดลต้นทุนของบทที่ 02 แอบสมมติ

> ว่าหน่วยความจำตอบสนองด้วยเวลาคงที่ ซึ่งชั้นนี้ไม่ได้ให้ได้ อ่านทั้งสองบทเทียบกันแล้วความขัดแย้งนั้นให้บทเรียนได้ บทที่ 12 คือจุดที่การเฉพาะทางกำลังมุ่งไปตอนนี้ และบทที่ 14 ถามว่ากำแพงพลังงาน (energy wall) จะทำอะไรกับมันทั้งหมดนี้

Open questions · คำถามที่ยังเปิดอยู่

Is a fixed instruction set still worth its cost, when the workloads that matter change faster than chips can be designed? If memory is the bottleneck, should computation move into memory instead of data moving to the processor? Speculative execution bought decades of speed and then turned out to leak secrets across the security boundary chapter 11 depends on; can a machine be fast and honest about what it is doing? And after specialization, what is left for a *general-purpose* computer to be?

> instruction set ที่ตายตัวยังคุ้มกับต้นทุนของมันอยู่ไหม ในเมื่องานที่สำคัญเปลี่ยนเร็วกว่าที่ชิปจะถูกออกแบบทัน ถ้าหน่วยความจำคือคอขวด การคำนวณควรย้ายเข้าไปอยู่ในหน่วยความจำแทนที่จะให้ข้อมูลย้ายไปหาโปรเซสเซอร์หรือเปล่า speculative execution ซื้อความเร็วมาได้หลายสิบปี แล้วก็กลายเป็นว่ามันรั่วความลับข้ามพรมแดนความปลอดภัยที่บทที่ 11 พึ่งพาอยู่ — เครื่องจะเร็วและซื่อสัตย์เรื่องที่มันกำลังทำอยู่ไปพร้อมกันได้ไหม แล้วหลังจากการเฉพาะทางแล้ว อะไรจะเหลือให้คอมพิวเตอร์ อเนกประสงค์ เป็นได้อีก

Go deeper · อ่านต่อ

— John L. Hennessy & David A. Patterson, *Computer Architecture: A Quantitative Approach*, 7th ed. (with Christos Kozyrakis), Morgan Kaufmann, 2025 — the book that made architecture an experimental science. Publisher.

> John L. Hennessy & David A. Patterson, Computer Architecture: A Quantitative Approach, 7th ed. (with Christos Kozyrakis), Morgan Kaufmann, 2025 — หนังสือที่ทำให้ architecture กลายเป็นวิทยาศาสตร์เชิงทดลอง สำนักพิมพ์

— Gene Amdahl, Gerrit Blaauw & Frederick Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development* 8, no. 2 (1964): 87–101 — the paper that defined the word.

> Gene Amdahl, Gerrit Blaauw & Frederick Brooks, "Architecture of the IBM System/360," IBM Journal of Research and Development 8, no. 2 (1964): 87–101 — เปเปอร์ที่นิยามคำนี้ขึ้นมา

— William A. Wulf & Sally A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH Computer Architecture News* 23, no. 1 (1995): 20–24 — two pages, thirty years of consequences.

> William A. Wulf & Sally A. McKee, "Hitting the Memory Wall: Implications of the Obvious," ACM SIGARCH Computer Architecture News 23, no. 1 (1995): 20–24 — สองหน้า สามสิบปีแห่งผลที่ตามมา

CHAPTER 05

Operating Systems

ระบบปฏิบัติการ

This chapter is the floor directly above the machine — where one processor is made to behave like a room full of them, and where the machine stops being something a program owns.

บทนี้คือชั้นที่อยู่เหนือตัวเครื่องขึ้นมาโดยตรง — ที่ซึ่งโปรเซสเซอร์ตัวเดียวถูกทำให้ทำตัวเหมือนมีเต็มห้อง และที่ซึ่งเครื่องเลิกเป็นสิ่งที่โปรแกรมใดโปรแกรมหนึ่งเป็นเจ้าของ

How does one machine pretend to be many, safely?

เครื่องเดียวแสร้งทำเป็นหลายเครื่องได้อย่างปลอดภัยอย่างไร

Between June 1985 and January 1987, six patients treated with a radiation machine called the Therac-25 received massive overdoses; at least three died. The machine was not broken in any way an engineer could see. Its predecessor had carried hardware interlocks — physical devices that made the dangerous configuration impossible — and on the Therac-25 they had been removed, because software would check instead. Nancy Leveson and Clark Turner found that the software checked, but that two of its concurrent tasks shared variables with no synchronization, so an operator typing quickly could change the treatment mode after the check had read it and before the hardware had finished moving (Leveson & Turner, *IEEE Computer*, July 1993). They also record what the machine was running on: "The system is not built using a standard operating system or executive. Rather, the real-time executive was written especially for the Therac-25" — by a single programmer, in assembly language, over several years.

> ระหว่างเดือนมิถุนายน 1985 ถึงมกราคม 1987 ผู้ป่วยหกคนที่รักษาด้วยเครื่องฉายรังสีชื่อ Therac-25 ได้รับปริมาณรังสีเกินขนาดอย่างรุนแรง อย่างน้อยสามคนเสียชีวิต ตัวเครื่องไม่ได้เสียในแบบที่วิศวกรจะมองเห็นได้เลย เครื่องรุ่นก่อนหน้ามีกลไก

> ล็อกฮาร์ดแวร์ (hardware interlock) — อุปกรณ์ทางกายภาพที่ทำให้การตั้งค่าที่อันตรายเกิดขึ้นไม่ได้เลย — แต่ใน Therac-25 อุปกรณ์เหล่านั้นถูกถอดออก เพราะให้ซอฟต์แวร์ตรวจสอบแทน Nancy Leveson และ Clark Turner พบว่าซอฟต์แวร์ตรวจสอบจริง แต่งาน (task) ที่ทำงานพร้อมกันสองงานใช้ตัวแปรร่วมกันโดยไม่มีการซิงโครไนซ์ ทำให้ผู้ปฏิบัติงานที่พิมพ์เร็วสามารถเปลี่ยนโหมดการรักษาได้หลังจากที่การตรวจสอบอ่านค่าไปแล้ว แต่ก่อนที่ฮาร์ดแวร์จะเคลื่อนที่เสร็จ (Leveson & Turner, IEEE Computer, July 1993) พวกเขายังบันทึกด้วยว่าเครื่องรันอยู่บนอะไร — "ระบบนี้ไม่ได้สร้างขึ้นด้วยระบบปฏิบัติการหรือ executive มาตรฐาน แต่ real-time executive ถูกเขียนขึ้นมาเฉพาะสำหรับ Therac-25" — โดยโปรแกรมเมอร์คนเดียว เขียนด้วยภาษาแอสเซมบลี ตลอดเวลาหลายปี

That last sentence is the reason this chapter exists. An operating system is not a convenience layer. It is the accumulated answer to three problems lethal to solve again from scratch: how one machine is shared, how concurrent things stay coherent, and how anything survives being switched off. Remzi and Andrea Arpaci-Dusseau call them virtualization, concurrency, and persistence (*Operating Systems: Three Easy Pieces*). The unifying idea is that *an operating system is a machine that lies, carefully*: every service it offers is a falsehood about the floor below, maintained well enough to be relied on.

> ประโยคสุดท้ายนั้นคือเหตุผลที่บทนี้มีอยู่ ระบบปฏิบัติการ (operating system) ไม่ใช่แค่ชั้นอำนวยความสะดวก มันคือคำตอบสะสมของปัญหาสามข้อที่อันตรายถึงตายถ้าต้องแก้ใหม่จากศูนย์ทุกครั้ง — เครื่องเดียวถูกแบ่งปันกันอย่างไร สิ่งที่ทำงานพร้อมกันคงความสอดคล้องกันได้อย่างไร และอะไรจะอยู่รอดหลังจากถูกปิดเครื่อง Remzi และ Andrea Arpaci-Dusseau เรียกสามข้อนี้ว่า การทำเสมือน (virtualization) การทำงานพร้อมกัน (concurrency) และความคงอยู่ของข้อมูล (persistence) (Operating Systems: Three Easy Pieces) ไอเดียที่รวมทั้งหมดไว้ด้วยกันคือ ระบบปฏิบัติการเป็นเครื่องที่โกหกอย่างระมัดระวัง — บริการทุกอย่างที่มันเสนอคือความเท็จเกี่ยวกับชั้นที่อยู่ข้างล่าง ซึ่งถูกรักษาไว้ดีพอที่จะพึ่งพาได้

The first lie is arithmetic. Your laptop has a handful of processors and is running hundreds of programs, each written as though it had a machine to itself. The operating system multiplexes: it runs one program for a few milliseconds, saves every register, loads another's, and lets a hardware timer interrupt it so that no program can decline to give the machine back. The larger lie is about memory. The Atlas machine at Manchester, running in 1962, introduced a "one-level storage system": programs addressed a store far larger than physical memory, and hardware quietly moved pages between core and drum (Kilburn, Edwards, Lanigan & Sumner, 1962). Every process since has been handed the same fiction — virtual memory, as every descendant calls it: a private address space, starting at zero, that appears to be all its own. Two programs can both hold "address 4096" and never meet.

> คำโกหกแรกคือเรื่องเลขคณิต แล็ปท็อปของคุณมีโปรเซสเซอร์อยู่ไม่กี่ตัว แต่กำลังรันโปรแกรมอยู่นับร้อย แต่ละตัวถูกเขียนขึ้นราวกับว่ามันมีเครื่องทั้งเครื่องเป็นของตัวเอง ระบบปฏิบัติการทำ multiplex — มันรันโปรแกรมหนึ่งไปสองสามมิลลิวินาที บันทึกค่าในทุกรีจิสเตอร์ โหลดค่าของอีกโปรแกรมหนึ่งเข้ามา แล้วปล่อยให้ตัวจับเวลาฮาร์ดแวร์ขัดจังหวะมัน เพื่อไม่ให้โปรแกรมไหนปฏิเสธที่จะคืนเครื่องได้ คำโกหกที่ใหญ่กว่าคือเรื่องหน่วยความจำ เครื่อง Atlas ที่ Manchester ซึ่งรันอยู่ในปี 1962 เปิดตัว "ระบบเก็บข้อมูลชั้นเดียว" (one-level storage system) — โปรแกรมอ้างที่อยู่ในพื้นที่เก็บข้อมูลที่ใหญ่กว่าหน่วยความจำจริงมาก และฮาร์ดแวร์ก็ย้าย page ระหว่าง core กับ drum อย่างเงียบๆ (Kilburn, Edwards, Lanigan & Sumner, 1962) ทุกโพรเซส (process) นับแต่นั้นมาได้รับเรื่องแต่งเดียวกันนี้ — หน่วยความจำเสมือน (virtual memory) อย่างที่ลูกหลานของมันทุกตัวเรียกกัน คือพื้นที่ที่อยู่ส่วนตัวที่เริ่มจากศูนย์ ซึ่งดูเหมือนเป็นของมันเองทั้งหมด โปรแกรมสองตัวถืออยู่ "address 4096" เดียวกันได้โดยไม่มีวันเจอกันเลย

A lie is only safe if it cannot be inspected from inside; here the operating system depends on the floor below. The kernel is not privileged because it is cleverer than your program; it is privileged because the processor refuses certain instructions unless a mode bit is set. Multics defined eight such rings of protection; x86 inherited the idea with four, of which mainstream systems use two — kernel and user (Multics protection). Chapter 04's in-

struction set does not know what a process is; it knows only that some instructions are forbidden in the outer mode, and that attempting one traps. Everything about isolation, security, and multi-tenancy is built from that single hardware gesture.

> คำโกหกจะปลอดภัยก็ต่อเมื่อมันถูกตรวจสอบจากข้างในไม่ได้ ตรงนี้เองที่ระบบปฏิบัติการต้องพึ่งชั้นที่อยู่ข้างล่าง เคอร์เนล (kernel) ไม่ได้มีสิทธิพิเศษเพราะมันฉลาดกว่าโปรแกรมของคุณ มันมีสิทธิพิเศษเพราะโปรเซสเซอร์ปฏิเสธที่จะรันคำสั่งเครื่องบางตัว เว้นแต่จะมีการตั้งค่า mode bit ไว้ Multics นิยามวงแหวนการป้องกัน (ring) แบบนี้ไว้แปดชั้น x86 รับไอเดียนี้มาด้วยสี่ชั้น ซึ่งระบบกระแสหลักใช้จริงแค่สองชั้น — เคอร์เนลกับผู้ใช้ (Multics protection) instruction set ของบทที่ 04 ไม่รู้ด้วยซ้ำว่าโพรเซส (process) คืออะไร มันรู้แค่ว่าคำสั่งเครื่องบางตัวถูกห้ามในโหมดนอก และถ้าพยายามรันมันก็จะเกิด trap ทุกเรื่องเกี่ยวกับ isolation ความปลอดภัย และ multi-tenancy ถูกสร้างขึ้นมาจากท่าทีเดียวของฮาร์ดแวร์ท่านี้เท่านั้น

The second problem, concurrency, is the one that killed the Therac patients. Once two things run at once and touch the same data, correctness stops being a property of either one alone. Edsger Dijkstra gave the field its basic tool in 1965, the semaphore, in notes titled "Cooperating Sequential Processes", and used it to build the layered THE system (*CACM*, May 1968). But mutual exclusion trades one failure for another: a process holding one lock while waiting for a second can wait forever. Coffman, Elphick and Shoshani named the four conditions that must all hold for deadlock — mutual exclusion, hold-and-wait, no preemption, and circular wait — in *ACM Computing Surveys* in 1971 (System Deadlocks). Break any one and deadlock is impossible; that is still how it is handled. Notice what the operating system offers: not a removal of the difficulty but primitives correct enough to build on. The honest reading of the Therac-25 is that an engineer who declines them is re-deriving 1965 alone, under deadline.

> ปัญหาที่สอง การทำงานพร้อมกัน (concurrency) คือสิ่งที่ฆ่าผู้ป่วย Therac พอสองสิ่งทำงานพร้อมกันแล้วแตะข้อมูลก้อนเดียวกัน ความถูกต้องก็เลิกเป็นคุณสมบัติของฝ่ายใดฝ่ายหนึ่งเพียงลำพัง Edsger Dijkstra มอบเครื่องมือพื้นฐานให้วงการนี้ในปี 1965 คือ semaphore ในบันทึกชื่อ "Cooperating Sequential Processes" แล้วใช้

มันสร้างระบบ THE ที่แบ่งเป็นชั้น (CACM, May 1968) แต่ mutual exclusion แลกความล้มเหลวแบบหนึ่งไปกับอีกแบบหนึ่ง — โพรเซสที่ถือล็อก (lock) หนึ่งตัวอยู่ระหว่างรอล็อกตัวที่สอง อาจต้องรอไปตลอดกาล Coffman, Elphick และ Shoshani ระบุเงื่อนไขสี่ข้อที่ต้องเป็นจริงพร้อมกันทั้งหมดจึงจะเกิด deadlock — mutual exclusion, hold-and-wait, no preemption และ circular wait — ใน ACM Computing Surveys ปี 1971 (System Deadlocks) ตัดเงื่อนไขไหนออกไปข้อหนึ่ง deadlock ก็เป็นไปไม่ได้ นั่นยังคือวิธีจัดการกับมันจนถึงทุกวันนี้ สังเกตว่าระบบปฏิบัติการเสนออะไรให้ — ไม่ใช่การกำจัดความยากออกไป แต่เป็น primitive ที่ถูกต้องพอจะให้ต่อยอดได้ การอ่าน Therac-25 อย่างตรงไปตรงมาคือ วิศวกรที่ปฏิเสธเครื่องมือเหล่านี้ กำลังนั่งคิดค้นปี 1965 ขึ้นมาใหม่คนเดียว ภายใต้ deadline

The third lie is that storage remembers. It does — but power can be cut between any two instructions, including the two that would have left the disk in a sensible state. The operating system's answer is `fsync`, which returns only when data has reached the device, and file systems that journal their intentions, so a half-finished operation can be replayed or undone. The abstraction leaks badly: `fsync` on a new file does not make the file's *directory entry* durable — you must sync the parent directory too. When Pillai and colleagues checked how eleven widely used applications behaved across crashes, they found sixty distinct crash vulnerabilities (*All File Systems Are Not Created Equal*, OSDI 2014). These were not amateur programs. The surface says "saved," and only the floor below knows whether that is true.

คำโกหกที่สามคือ ที่เก็บข้อมูลจดจำได้ มันจำได้จริง — แต่ไฟสามารถถูกตัดระหว่างคำสั่งเครื่องสองตัวใดๆ ก็ได้ รวมถึงสองตัวที่ถ้าทำเสร็จแล้วจะทำให้ดิสก์อยู่ในสถานะ (state) ที่สมเหตุสมผล คำตอบของระบบปฏิบัติการคือ `fsync` ซึ่งจะ return ก็ต่อเมื่อข้อมูลไปถึงอุปกรณ์แล้วเท่านั้น และระบบไฟล์ (file system) ที่จด journal ความตั้งใจของตัวเองไว้ เพื่อให้ operation ที่ทำค้างไว้ครึ่งเดียวถูกเล่นซ้ำหรือย้อนกลับได้ นามธรรม (abstraction) ตัวนี้รั่วอย่างหนัก — `fsync` บนไฟล์ใหม่ไม่ได้ทำให้ directory entry ของไฟล์นั้นคงทนถาวรไปด้วย คุณต้อง sync ไดเรกทอรีแม่ด้วย เมื่อ Pillai และเพื่อนร่วมงานตรวจสอบว่าแอปพลิเคชันที่ใช้กันแพร่หลาย 11 ตัวมี

> พฤติกรรมอย่างไรเมื่อเครื่อง crash พวกเขาพบช่องโหว่ตอน crash ที่ต่างกันถึง 60 จุด (All File Systems Are Not Created Equal, OSDI 2014) โปรแกรมพวกนี้ไม่ใช่ของมือสมัครเล่นเลย ผิว (surface) บอกว่า "บันทึกแล้ว" มีแต่ชั้นที่อยู่ข้างล่างเท่านั้นที่รู้ว่านั่นเป็นจริงหรือเปล่า

One design rule holds the whole thing together: separate *mechanism* from *policy*. The kernel supplies the machinery to do a thing and leaves the decision of what to do to whoever is above it. Hydra stated it as a design principle in 1974 (Wulf et al., CACM 17, no. 6), and Butler Lampson's "Hints for Computer System Design" made it common sense by 1983. The reason is simple: policy changes far faster than mechanism, and a kernel that bakes in a scheduling preference or a security model must be rewritten when opinion moves. The interface that results is astonishingly narrow for what it carries: the x86-64 system call table in the current Linux source lists 385 entries (kernel source, checked 1 August 2026). Everything you have ever run passed through that door.

> กฎการออกแบบข้อหนึ่งยึดทุกอย่างไว้ด้วยกัน คือแยก กลไก (mechanism) ออกจาก นโยบาย (policy) เคอร์เนลจัดหากลไกให้ทำสิ่งหนึ่ง แล้วปล่อยให้การตัดสินใจว่าจะทำอะไรเป็นหน้าที่ของสิ่งที่อยู่เหนือมันขึ้นไป Hydra ระบุเรื่องนี้ไว้เป็นหลักการออกแบบตั้งแต่ปี 1974 (Wulf et al., CACM 17, no. 6) และ "Hints for Computer System Design" ของ Butler Lampson ทำให้มันกลายเป็นสามัญสำนึกภายในปี 1983 เหตุผลนั้นง่ายมาก — นโยบายเปลี่ยนเร็วกว่ากลไกมาก และเคอร์เนลที่ฝังความชอบเรื่อง scheduling หรือแบบจำลองความปลอดภัยไว้ตายตัว ก็ต้องถูกเขียนใหม่ทุกครั้งที่ความเห็นเปลี่ยนไป อินเทอร์เฟซที่ได้ออกมาแคบอย่างน่าประหลาดใจเมื่อเทียบกับสิ่งที่มันแบกรับอยู่ — ตาราง system call ของ x86-64 ใน Linux source ปัจจุบันมีอยู่ 385 รายการ (kernel source, checked 1 August 2026) ทุกอย่างที่คุณเคยรันผ่านประตูบานนั้นมาแล้วทั้งนั้น

The people here inherited each other directly. Fernando Corbató demonstrated the Compatible Time-Sharing System at MIT in 1961 and then led Multics, whose ambition was computing sold like electricity — a utility, always on, shared by strangers. Bell Labs pulled out in 1969, and Dennis

Ritchie's account of what followed is unglamorous: the Labs refused to buy a replacement machine, so "it did not take long, therefore, for Thompson to find a little-used PDP-7" (Ritchie, *The Evolution of the Unix Time-sharing System*). Ken Thompson wrote the first Unix on it in about a month, in the summer of 1969, while his wife was away — "essentially one person for a month, it was just myself," he said later (oral history, 1989). Multics tried to specify everything; Unix was small because it had no budget to be large, and the small one spread. Thompson and Ritchie received the Turing Award in 1983, Corbató in 1990.

> คนในเรื่องนี้สืบทอดต่อกันมาโดยตรง Fernando Corbató สาธิต Compatible Time-Sharing System ที่ MIT ในปี 1961 แล้วนำทีม Multics ซึ่งมีความทะเยอทะยานคือทำให้การคำนวณถูกขายเหมือนไฟฟ้า — เป็นสาธารณูปโภคที่เปิดอยู่ตลอดเวลาแบ่งปันกันระหว่างคนแปลกหน้า Bell Labs ถอนตัวออกในปี 1969 และเรื่องเล่าของ Dennis Ritchie เกี่ยวกับสิ่งที่เกิดขึ้นต่อมานั้นไม่หรูหราเลย — ทาง Labs ปฏิเสธที่จะซื้อเครื่องทดแทน ดังนั้น "จึงไม่ต้องใช้เวลานานนักที่ Thompson จะหา PDP-7 เครื่องหนึ่งที่แทบไม่มีใครใช้เจอ" ("it did not take long, therefore, for Thompson to find a little-used PDP-7") (Ritchie, The Evolution of the Unix Time-sharing System) Ken Thompson เขียน Unix ตัวแรกลงเครื่องนั้นภายในเวลาประมาณหนึ่งเดือน ในฤดูร้อนปี 1969 ขณะที่ภรรยาของเขาไม่อยู่บ้าน — "โดยพื้นฐานแล้วคือคนคนเดียวทำเป็นเวลาหนึ่งเดือน มีแค่ผมคนเดียวเท่านั้น" ("essentially one person for a month, it was just myself") เขาพูดถึงเรื่องนี้ในภายหลัง (oral history, 1989) Multics พยายามระบุทุกอย่างไว้ล่วงหน้า ส่วน Unix มีขนาดเล็กเพราะไม่มีงบให้ใหญ่ได้ และตัวที่เล็กกว่านั้นเองที่แพร่กระจายออกไป Thompson กับ Ritchie ได้รับรางวัล Turing Award ในปี 1983 ส่วน Corbató ได้รับในปี 1990

Connections · เชื่อมโยง

Below is 04, which supplies everything that makes the illusion enforceable: privileged mode, the timer interrupt, and address-translation hardware. Above it, 06 compiles programs against the process abstraction described here. Chapter 07 carries the same instinct upward — hide the decision that will change, publish only the interface — and chapter 08 is largely the story

of one application refusing to trust this floor's persistence guarantees and rebuilding them itself. Chapter 10 asks what happens to all of this when the "one machine" is a thousand of them, and chapter 11 treats the kernel boundary as the line an attacker is trying to cross.

> ชั้นล่างคือบทที่ 04 ซึ่งจัดหาทุกอย่างที่ทำให้ภาพลวงตานี้บังคับใช้ได้จริง — โหมดสิทธิพิเศษ การขัดจังหวะของตัวจับเวลา และฮาร์ดแวร์แปลที่อยู่ เหนือขึ้นไป บทที่ 06 คอมไพล์โปรแกรมโดยอิงกับ abstraction ของโพรเซสที่อธิบายไว้ในบทนี้ บทที่ 07 สานสัญชาตญาณเดียวกันนี้ขึ้นไปอีก — ซ่อนการตัดสินใจที่จะเปลี่ยนแปลงเผยแพร่แค่อินเทอร์เฟซ — และบทที่ 08 ส่วนใหญ่คือเรื่องราวของแอปพลิเคชันหนึ่งตัวที่ปฏิเสธจะเชื่อการรับประกันเรื่องความคงอยู่ของข้อมูล (persistence) ของชั้นนี้แล้วสร้างมันขึ้นมาใหม่เอง บทที่ 10 ถามว่าจะเกิดอะไรขึ้นกับทั้งหมดนี้ เมื่อ "เครื่องเดียว" กลายเป็นเครื่องนับพัน และบทที่ 11 มองพรมแดนของเคอร์เนลว่าเป็นเส้นที่ผู้โจมตีพยายามข้ามผ่าน

Open questions · คำถามที่ยังเปิดอยู่

Is the kernel still the right place for the boundary? Microkernels, unikernels, and in-kernel virtual machines each move the line, and none has settled the argument. The process abstraction was designed for a machine whose interesting hardware was the CPU; today much of the work happens on accelerators the kernel barely schedules. And why, after sixty years, is concurrency still where careful engineers are most likely to be wrong?

> เคอร์เนลยังคือที่ที่ถูกต้องสำหรับวางพรมแดนนี้อยู่ไหม microkernel, unikernel และ virtual machine ที่อยู่ในเคอร์เนล ต่างก็ขยับเส้นนี้กันไปคนละทาง และไม่มีตัวไหนยุติข้อถกเถียงนี้ได้ abstraction ของโพรเซสถูกออกแบบมาสำหรับเครื่องที่ฮาร์ดแวร์สำคัญคือ CPU แต่ทุกวันนี้งานส่วนใหญ่เกิดขึ้นบนตัวเร่ง (accelerator) ที่เคอร์เนลแทบไม่ได้จัดคิวให้เลย แล้วทำไมหลังจากผ่านมาหกสิบปี การทำงานพร้อมกัน (concurrency) ยังคงเป็นจุดที่วิศวกรที่รอบคอบมีโอกาสทำผิดพลาดมากที่สุดอยู่ดี

Go deeper · อ่านต่อ

- Remzi H. Arpaci-Dusseau & Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, version 1.10, 2023 — the clearest treatment of the three pieces, free online.

> Remzi H. Arpaci-Dusseau & Andrea C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, version 1.10, 2023 — บทความที่อธิบายสามชิ้นส่วนนี้ได้ชัดเจนที่สุด อ่านฟรีออนไลน์

- Dennis M. Ritchie & Ken Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* 17, no. 7 (July 1974): 365–375 — the design that most systems still imitate.

> Dennis M. Ritchie & Ken Thompson, "The UNIX Time-Sharing System," Communications of the ACM 17, no. 7 (July 1974): 365–375 — การออกแบบที่ระบบส่วนใหญ่ยังเลียนแบบอยู่จนถึงทุกวันนี้

- Nancy G. Leveson & Clark S. Turner, "An Investigation of the Therac-25 Accidents," *IEEE Computer* 26, no. 7 (July 1993): 18–41 — the case every systems engineer should read once.

> Nancy G. Leveson & Clark S. Turner, "An Investigation of the Therac-25 Accidents," IEEE Computer 26, no. 7 (July 1993): 18–41 — เคสที่วิศวกรระบบทุกคนควรอ่านสักครั้ง

CHAPTER 06

Languages & Compilers

ภาษาและคอมไพเลอร์

This chapter is the floor where instructions stop being addressed to a machine and start being addressed to people — and where a program is written to be read.

บทนี้คือชั้นที่คำสั่งเลิกพูดกับเครื่องจักร แล้วเริ่มพูดกับคน — และเป็นที่ซึ่งโปรแกรมถูกเขียนขึ้นมาเพื่อให้คนอ่าน

What makes a text mean something to a machine?

อะไรทำให้ข้อความมีความหมายต่อเครื่องจักร?

When John Backus's group at IBM began work on FORTRAN, the job was estimated at six months. In his own history of the project, Backus records that the six-month figure "remained the interval-to-completion until it actually was completed over two years later, in April 1957." The delay is not the interesting part. What everyone expected the result to be is. Backus recalled that "it is difficult to convey ... the strength of the skepticism about 'automatic programming'" and about "its ability to produce efficient programs"; his team's stated goal was only to "produce programs almost as efficient as hand coded ones" (Backus, *The History of FORTRAN I, II, and III*). *Almost.* The premise of the field was that a machine could not read a human notation and produce good code, and that premise was not stupid. Nothing about meaning looks mechanical.

> ตอนที่ทีมของ John Backus ที่ IBM เริ่มทำงานกับ FORTRAN งานนี้ถูกประเมินไว้ว่าใช้เวลาหกเดือน ในบันทึกประวัติโครงการของตัวเอง Backus เล่าว่าตัวเลขหกเดือนนั้น "ยังคงเป็นระยะเวลาที่คาดไว้จนกว่างานจะเสร็จจริง ซึ่งเกิดขึ้นอีกกว่าสองปีให้หลัง ในเดือนเมษายน 1957" (remained the interval-to-completion until it actually was completed over two years later, in April 1957) ความล่าช้าไม่ใช่

> ส่วนที่น่าสนใจ สิ่งที่ทุกคนคาดหวังว่าผลลัพธ์จะเป็นต่างหากที่น่าสนใจ Backus เล่าว่า "ยากที่จะสื่อ ... ถึงความแรงของความคลางแคลงใจต่อ 'automatic programming'" (it is difficult to convey ... the strength of the skepticism about 'automatic programming') และต่อ "ความสามารถของมันในการผลิตโปรแกรมที่มีประสิทธิภาพ" (its ability to produce efficient programs) เป้าหมายที่ทีมของเขาประกาศไว้มีแค่ "ผลิตโปรแกรมที่มีประสิทธิภาพเกือบเท่าโค้ดที่เขียนด้วยมือ" (produce programs almost as efficient as hand coded ones) (Backus, The History of FORTRAN I, II, and III) เกือบ. ข้อสมมติของวงการตอนนั้นคือเครื่องจักรไม่มีทางอ่านสัญกรณ์ของมนุษย์แล้วผลิตโค้ดที่ดีออกมาได้ และข้อสมมตินั้นก็ไม่ได้โง่เขลาอะไร ไม่มีอะไรในเรื่องความหมายที่ดูเป็นกลไกเลย

The first move that made it mechanical was to cut meaning in two. **Syntax** is the question of whether a text is well formed; **semantics** is the question of what it does. They are not even the same *kind* of question. Backus described the syntax of a proposed international algebraic language in 1959 using a notation of rules and alternatives; Peter Naur adapted it for the ALGOL 60 report, and Donald Knuth's 1964 letter to *CACM* proposed the name it still carries, Backus–Naur Form (Knuth, 1964). With a formal grammar, "is this a valid program" stops being a matter of judgment and becomes a decision a program can make. ALGOL 60 was the demonstration, and Tony Hoare's verdict is the most quoted line in language design: "Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors" — from his 1973 keynote "Hints on Programming Language Design", not, as is usually claimed, his Turing lecture.

> การเคลื่อนไหวแรกที่ทำให้เรื่องนี้กลายเป็นกลไกได้ คือการตัดความหมายออกเป็นสองส่วน **ไวยากรณ์ (syntax)** คือคำถามว่าข้อความมีรูปที่ถูกต้องหรือไม่ **ความหมาย (semantics)** คือคำถามว่ามันทำอะไร ทั้งสองไม่ใช่คำถามแบบเดียวกัน ด้วยซ้ำ Backus บรรยาย syntax ของภาษาพีชคณิตสากลที่มีการเสนอไว้ในปี 1959 ด้วยสัญกรณ์ของกฎและทางเลือก Peter Naur นำมันไปดัดแปลงใช้ในรายงาน ALGOL 60 และจดหมายของ Donald Knuth ถึง CACM ในปี 1964 เสนอชื่อที่ยังใช้กันอยู่จนถึงทุกวันนี้ คือ Backus–Naur Form (Knuth, 1964) เมื่อมีไวยากรณ์รูปนัย

> แล้ว คำถามว่า "โปรแกรมนี้ถูกต้องหรือไม่" ก็เลิกเป็นเรื่องดุลพินิจ กลายเป็นการตัดสินใจที่โปรแกรมทำเองได้ ALGOL 60 คือบทพิสูจน์ของแนวคิดนี้ และคำตัดสินของ Tony Hoare ก็เป็นประโยคที่ถูกอ้างถึงมากที่สุดในวงการออกแบบภาษา คือ "นี่คือภาษาที่ล้ำยุคของมันมากจนไม่ใช่แค่ก้าวหน้ากว่ารุ่นก่อนหน้า แต่ยังก้าวหน้ากว่ารุ่นต่อๆ มาเกือบทั้งหมดด้วย" (Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors) — จากปาฐกถาปี 1973 เรื่อง "Hints on Programming Language Design" ไม่ใช่จากปาฐกถา Turing อย่างที่มักถูกอ้างผิดๆ

Semantics resisted the same treatment, and the compiler's answer was not to solve meaning but to descend it — one small, exact step at a time. A compiler is a tower inside the tower. Lexical analysis turns characters into words. Parsing turns words into a tree that reflects the grammar. Semantic analysis annotates that tree with what the names refer to and whether the operations are permitted. Then the tree becomes an intermediate representation belonging to no particular language and no particular machine; optimization rewrites it into a cheaper one with the same meaning; and code generation emits the instruction set of chapter 04. Each stage exists because it hands the next one a representation in which the next one's job is easy — which is this book's whole thesis, performed in miniature inside a single program. The intermediate representation earns its keep with arithmetic: without it, supporting n languages on m machines takes $n \times m$ compilers; with it, $n + m$.

> ความหมายไม่ยอมให้ถูกจัดการด้วยวิธีเดียวกัน คำตอบของคอมไพเลอร์จึงไม่ใช่การไขความหมายให้กระจ่าง แต่คือการลงบันไดทีละขั้นเล็กๆ ที่แม่นยำ คอมไพเลอร์คือหอคอยซ้อนอยู่ในหอคอย การวิเคราะห์คำศัพท์ (lexical analysis) แปลงตัวอักษรให้เป็นคำ parsing แปลงคำให้เป็นต้นไม้ที่สะท้อนไวยากรณ์ การวิเคราะห์ความหมาย (semantic analysis) เติมลงบนต้นไม้นั้นว่าแต่ละชื่ออ้างถึงอะไร และการดำเนินการต่างๆ ได้รับอนุญาตหรือไม่ จากนั้นต้นไม้จะกลายเป็นตัวแทนกลาง (intermediate representation) ที่ไม่ผูกกับภาษาใดภาษาหนึ่งหรือเครื่องใดเครื่องหนึ่ง การเพิ่มประสิทธิภาพ (optimization) เขียนมันใหม่ให้ต้นทุนถูกลงโดยความหมายเดิม และการสร้างโค้ด (code generation) ปล่อยชุดคำสั่งเครื่องของบทที่ 04 ออกมา แต่ละ

> ขั้นมีอยู่เพราะมันส่งต่อตัวแทนที่ทำให้งานของขั้นถัดไปง่ายขึ้น — ซึ่งคือวิทยานิพนธ์ทั้งเล่มของหนังสือนี้ ที่แสดงย่อไว้ในโปรแกรมเดียว ตัวแทนกลางคุ้มค่าตัวมันเองด้วยเลขคณิตล้วนๆ ถ้าไม่มีมัน การรองรับ n ภาษาบน m เครื่อง ต้องใช้คอมไพเลอร์ n × m ตัว แต่ถ้ามีมัน ใช้แค่ n + m ตัว

What languages built on that machinery offer is a set of ways to name things — functions, types, modules, objects — each of which exists to let a reader ignore something. The most compressed illustration is LISP. In April 1960 John McCarthy published a paper defining computation over symbolic expressions, including a function called `eval` that described what it means to evaluate any expression in the language (*Recursive Functions of Symbolic Expressions*, *CACM* 3, no. 4). It was mathematics, meant for reading. Then Steve Russell hand-compiled `eval` into IBM 704 machine code (McCarthy, *History of Lisp*), and the definition of the language became an interpreter for the language. That is chapter 04's stored-program idea coming due: because a program is data, a description of a language is a program that runs it.

> สิ่งที่ภาษาที่สร้างขึ้นบนกลไกนั้นมอบให้ คือชุดวิธีตั้งชื่อให้กับสิ่งต่างๆ — ฟังก์ชัน, ชนิดข้อมูล (type), มอดูล (module), อ็อบเจกต์ (object) — แต่ละอย่างมีไว้เพื่อให้ผู้อ่านเพิกเฉยต่อบางสิ่งได้ ตัวอย่างที่กระชับที่สุดคือ LISP ในเดือนเมษายน 1960 John McCarthy ตีพิมพ์เปเปอร์ที่นิยามการคำนวณบนนิพจน์เชิงสัญลักษณ์ (symbolic expressions) รวมถึงฟังก์ชันชื่อ `eval` ที่อธิบายว่าการประเมินนิพจน์ใดๆ ในภาษานี้หมายความว่าอย่างไร (Recursive Functions of Symbolic Expressions, CACM 3, no. 4) มันคือคณิตศาสตร์ ที่ตั้งใจให้อ่าน แล้ว Steve Russell ก็คอมไพล์ `eval` ด้วยมือลงเป็นโค้ดเครื่อง IBM 704 (McCarthy, History of Lisp) และนิยามของภาษาก็กลายเป็นอินเทอร์พรีเตอร์ (interpreter) ของภาษานั้นเอง นั่นคือไอเดียโปรแกรมที่เก็บในหน่วยความจำ (stored program) จากบทที่ 04 ที่มาถึงจุดจ่ายคืน เพราะโปรแกรมคือข้อมูล คำอธิบายของภาษาจึงเป็นโปรแกรมที่รันภาษานั้นเองได้

Types are the sharpest of the naming mechanisms, because a type is a promise a machine can check. Declaring that a function takes an integer and returns a string is a claim about every possible execution, verified

before any execution happens — the only part of programming where you get a guarantee rather than a test result. The guarantee is narrow but real: when Gao, Bird and Barr took public JavaScript bugs, checked out the code just before each fix, added type annotations by hand, and asked whether a type checker would have complained, both Flow and TypeScript caught 15 percent of them (*To Type or Not to Type*, ICSE 2017). Read that carefully: it is not "15 percent of bugs are type errors," but 15 percent of bugs that had already survived testing, review, and release. Push the idea far enough and it stops being about bugs. The Curry–Howard correspondence, observed by Haskell Curry in the 1930s and stated by William Howard in a 1969 manuscript titled "The Formulae-as-Types Notion of Construction," says that types *are* propositions and programs *are* proofs of them. A type checker is a small automatic theorem prover most programmers use daily without noticing.

> Type คือกลไกตั้งชื่อที่คมที่สุด เพราะ type คือคำสัญญาที่เครื่องจักรตรวจสอบได้ การประกาศว่าฟังก์ชันหนึ่งรับจำนวนเต็มเข้ามาแล้วคืนค่าเป็นสตริงออกไป คือข้อกล่าวอ้างเกี่ยวกับการรันทุกครั้งที่เป็นไปได้ ซึ่งถูกตรวจสอบก่อนที่การรันครั้งใดจะเกิดขึ้นจริงด้วยซ้ำ — เป็นส่วนเดียวของการเขียนโปรแกรมที่คุณได้การรับประกัน (guarantee) แทนที่จะได้แค่ผลจากการเทสต์ การรับประกันนี้แคบแต่จริง เมื่อ Gao, Bird และ Barr เอาบั๊กจริงของ JavaScript ที่เปิดเผยต่อสาธารณะมาศึกษา เช็คเอาต์โค้ดช่วงก่อนแก้บั๊กแต่ละตัว ใส่ type annotation ด้วยมือ แล้วถามว่าตัวตรวจชนิด (type checker) จะโวยวายไหม ผลคือทั้ง Flow และ TypeScript จับบั๊กได้ 15 เปอร์เซ็นต์ (To Type or Not to Type, ICSE 2017) อ่านให้ละเอียด นี่ไม่ใช่ "15 เปอร์เซ็นต์ของบั๊กเป็นข้อผิดพลาดเรื่อง type" แต่คือ 15 เปอร์เซ็นต์ของบั๊กที่รอดจากการเทสต์ การรีวิว และการปล่อยใช้งานจริงมาแล้ว ดันไอเดียนี้ไปไกลพอ มันก็เลิกเป็นเรื่องบั๊กอีกต่อไป Curry–Howard correspondence ที่ Haskell Curry สังเกตเห็นในทศวรรษ 1930 และ William Howard ระบุไว้ในต้นฉบับปี 1969 ชื่อ "The Formulae-as-Types Notion of Construction" บอกว่า type คือ ข้อเสนอ (proposition) และโปรแกรม คือ บทพิสูจน์ของมัน type checker ก็คือเครื่องพิสูจน์ทฤษฎีบทอัตโนมัติตัวจิ๋วที่โปรแกรมเมอร์ส่วนใหญ่ใช้ทุกวันโดยไม่ทันสังเกต

Which raises the question of why there are so many. The HOPL database, maintained since 1995, catalogued 8,945 of them as of August 2026. The proliferation is not waste. A language is a bet about which mistakes are worth preventing and which repetitions are worth naming, and those bets are genuinely different for an operating system kernel, a statistical model, and a web page. Alan Perlis put the useful test in his 1982 *Epigrams on Programming*: "A language that doesn't affect the way you think about programming, is not worth knowing." What has not survived is the 1957 fear. Compilers routinely beat competent hand-written assembly, and superoptimizers now beat compilers: STOKE, searching randomly from unoptimized code for equivalent rewrites, matched or outperformed the best output of production compilers (Schkufza, Sharma & Aiken, ASPLOS 2013). Backus's team asked for *almost* as good, and lost that bet in the most agreeable direction.

> ซึ่งนำไปสู่คำถามว่าทำไมถึงมีภาษามากมายขนาดนี้ ฐานข้อมูล HOPL ที่ดูแลมาตั้งแต่ปี 1995 นับได้ 8,945 ภาษา ณ เดือนสิงหาคม 2026 การแตกแขนงนี้ไม่ใช่ความสูญเปล่า ภาษาหนึ่งคือการเดิมพันว่าความผิดพลาดแบบไหนคุ้มที่จะป้องกัน และการทำซ้ำแบบไหนคุ้มที่จะตั้งชื่อให้ ซึ่งการเดิมพันนั้นต่างกันจริงๆ ระหว่างเคอร์เนลของระบบปฏิบัติการ โมเดลเชิงสถิติ และหน้าเว็บ Alan Perlis วางบททดสอบที่ใช้งานได้จริงไว้ใน Epigrams on Programming ปี 1982 ว่า "ภาษาที่ไม่กระทบวิธีคิดของคุณเกี่ยวกับการเขียนโปรแกรม ก็ไม่คุ้มที่จะรู้จัก" (A language that doesn't affect the way you think about programming, is not worth knowing) สิ่งที่ไม่รอดมาคือความกลัวของปี 1957 คอมไพเลอร์เอาชนะแอสเซมบลีที่เขียนด้วยมืออย่างเก่งกาจได้เป็นเรื่องปกติไปแล้ว และตอนนี้ superoptimizer ก็เอาชนะคอมไพเลอร์ได้อีกที: STOKE ค้นหาแบบสุ่มจากโค้ดที่ยังไม่ถูกเพิ่มประสิทธิภาพ เพื่อหารูปแบบเขียนใหม่ที่ความหมายเท่ากัน แล้วให้ผลเทียบเท่าหรือดีกว่าผลลัพธ์ที่ดีที่สุดของคอมไพเลอร์ระดับโปรดักชัน (Schkufza, Sharma & Aiken, ASPLOS 2013) ทีมของ Backus ขอแค่ให้ดี เกือบ เท่าโค้ดมือ แล้วก็แพ้การเดิมพันนั้นไปในทิศทางที่น่ายินดีที่สุด

The people on this floor had a habit of turning on their own work. Backus received the Turing Award in 1977 and used the lecture to attack the model FORTRAN entrenched, titling it "Can Programming Be Liberated from the von Neumann Style?" — published in *CACM* the following August. Grace Hopper's A-0 system of 1951–52 is often called the first compiler; it functioned closer to a loader, and the more consequential work was FLOW-MATIC, whose English-like verbs passed almost intact into COBOL and put readable text into commercial programming for good. She retired from the US Navy in 1986 as a rear admiral, in the habit of handing visitors a length of wire just under a foot long — the distance light travels in a nanosecond — so that they would stop treating wasted time as free.

> คนที่อยู่ชั้นนี้มีนิสัยชอบหันมาโจมตีงานของตัวเอง Backus ได้รับรางวัล Turing Award ในปี 1977 แล้วใช้ปาฐกถานั้นโจมตีโมเดลที่ FORTRAN ปักหลักไว้ ตั้งชื่อปาฐกถาว่า "Can Programming Be Liberated from the von Neumann Style?" — ตีพิมพ์ใน CACM เดือนสิงหาคมปีถัดมา ระบบ A-0 ของ Grace Hopper ในปี 1951–52 มักถูกเรียกว่าเป็นคอมไพเลอร์ตัวแรก แต่จริงๆ มันทำงานคล้ายตัวโหลด (loader) มากกว่า งานที่ส่งผลจริงกว่าคือ FLOW-MATIC ซึ่งคำกริยาที่คล้ายภาษาอังกฤษของมันถูกส่งต่อเข้าไปใน COBOL แทบจะทั้งดุ้น และทำให้ข้อความที่คนอ่านออกกลายเป็นส่วนหนึ่งของการเขียนโปรแกรมเชิงพาณิชย์ไปตลอดกาล เธอเกษียณจากกองทัพเรือสหรัฐฯ ในปี 1986 ด้วยยศพลเรือตรี และมีนิสัยชอบยื่นเส้นลวดยาวเกือบหนึ่งฟุตให้แขกที่มาเยี่ยม — ระยะทางที่แสงเดินทางได้ในหนึ่งนาโนวินาที — เพื่อให้พวกเขาเลิกคิดว่าเวลาที่เสียไปนั้นไม่มีต้นทุน

Connections · เชื่อมโยง

Below this chapter, 04 supplies the instruction set the back end emits into, and 05 the process abstraction a compiled program is loaded into. Chapter 01 sets the ceiling: because the halting problem is undecidable, no compiler can decide in general whether an optimization is safe in every case, which is why real optimizers are conservative by construction. Above, chapter 07

asks which of these naming mechanisms keep large systems changeable over years, and chapter 12 raises the possibility that the next notation people write in will not be a programming language at all.

> ชั้นล่างของบทนี้ บทที่ 04 มอบชุดคำสั่งเครื่องที่ back end ปล่อยโค้ดออกไปหา และบทที่ 05 มอบ abstraction ของ process ที่โปรแกรมซึ่งคอมไพล์แล้วถูกโหลดเข้าไปอยู่ บทที่ 01 กำหนดเพดานไว้ เพราะปัญหาการหยุด (halting problem) ตัดสินไม่ได้ คอมไพเลอร์จึงไม่มีทางตัดสินได้ในกรณีทั่วไปว่าการเพิ่มประสิทธิภาพหนึ่งๆ ปลอดภัยในทุกกรณีหรือไม่ นี่คือเหตุผลที่ optimizer ตัวจริงถูกออกแบบมาให้ระมัดระวังไว้ก่อนเสมอ ชั้นบน บทที่ 07 ถามว่ากลไกตั้งชื่อเหล่านี้ตัวไหนที่ทำให้ระบบใหญ่ๆ เปลี่ยนแปลงได้ตลอดหลายปี และบทที่ 12 ยกความเป็นไปได้ว่าสัญกรณ์ถัดไปที่คนจะใช้เขียนอาจไม่ใช่ภาษาโปรแกรมเลยด้วยซ้ำ

Open questions · คำถามที่ยังเปิดอยู่

If a model can produce working code from an English description, what is a programming language for — specification, verification, or only review? Types demonstrably catch real defects, so why does a large part of the industry still decline them, and is the honest answer about evidence or about cost? And what would a language look like that was designed for programs to read rather than write?

> ถ้าโมเดลสามารถผลิตโค้ดที่ทำงานได้จากคำอธิบายภาษาอังกฤษ ภาษาโปรแกรมมีไว้ทำอะไรกันแน่ — เพื่อทำสเปก (specification) การพิสูจน์ความถูกต้อง (verification) หรือแค่การรีวิว? Type พิสูจน์ได้ชัดว่าจับข้อบกพร่องจริงได้ แล้วทำไมอุตสาหกรรมส่วนใหญ่ถึงยังปฏิเสธมันอยู่ คำตอบที่ตรงไปตรงมาคือเรื่องหลักฐานหรือเรื่องต้นทุนกันแน่? และภาษาที่ถูกออกแบบมาให้โปรแกรมอ่านมากกว่าให้คนเขียน จะมีหน้าตาแบบไหน?

Go deeper · อ่านต่อ

- Alfred V. Aho, Monica S. Lam, Ravi Sethi & Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed., Addison-Wesley, 2006 — the "dragon book," still the standard treatment of the pipeline.

> Alfred V. Aho, Monica S. Lam, Ravi Sethi & Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed., Addison-Wesley, 2006 — หรือที่รู้จักกันในชื่อ "dragon book" ยังคงเป็นตำราอ้างอิงมาตรฐานสำหรับ pipeline ของคอมไพเลอร์

— John Backus, "The History of FORTRAN I, II, and III," *ACM SIGPLAN Notices* 13, no. 8 (1978) — a founder's account of proving his field's skeptics wrong.

> John Backus, "The History of FORTRAN I, II, and III," ACM SIGPLAN Notices 13, no. 8 (1978) — เรื่องเล่าของผู้ก่อตั้งวงการที่พิสูจน์ให้คนที่ไม่เชื่อเห็นว่าตัวเองคิดผิด

— John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Communications of the ACM* 3, no. 4 (1960): 184–195 — a language defined, not implemented, and implemented anyway.

> John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," Communications of the ACM 3, no. 4 (1960): 184–195 — ภาษาที่ถูกนิยามไว้ ไม่ได้ถูกทำให้ใช้งานได้จริง แต่สุดท้ายก็มีคนทำให้ใช้งานได้จริงอยู่ดี

CHAPTER 07

Software Engineering

วิศวกรรมซอฟต์แวร์

This chapter is the floor where a program stops being a text that runs and becomes an object that has to survive being changed.

บทนี้คือชั้นที่โปรแกรมเลิกเป็นแค่ข้อความที่รันได้ แล้วกลายเป็นวัตถุที่ต้องอยู่รอดผ่านการถูกเปลี่ยนแปลง

Why does software rot when it has no moving parts?

ทำไมซอฟต์แวร์ถึงผุ (rot) ทั้งที่ไม่มีชิ้นส่วนที่เคลื่อนไหวเลย?

On 1 August 2012, the trading firm Knight Capital pushed new code to the eight servers that routed its orders to the New York Stock Exchange. It arrived on seven. The eighth still carried a routine called Power Peg, which the firm had stopped using in 2003 and never deleted, and the new code had reused an old feature flag that — on that one machine — switched the dead routine back on. Over roughly forty-five minutes Knight sent millions of unintended orders into the market and lost $460 million, more than the company was worth. The Securities and Exchange Commission's order records one further detail: the dormant code had last been disturbed in 2005, when a function it depended on was moved elsewhere and nobody retested the combination (SEC Release 34-70694, 2013). Knight paid a $12 million penalty and did not survive the year as an independent firm.

> วันที่ 1 สิงหาคม 2012 บริษัทเทรดหุ้น Knight Capital ปล่อยโค้ดใหม่ลงเซิร์ฟเวอร์แปดเครื่องที่ทำหน้าที่ส่งคำสั่งซื้อขายไปยังตลาดหุ้นนิวยอร์ก โค้ดไปถึงเจ็ดเครื่อง เครื่องที่แปดยังมีรูทีนชื่อ Power Peg ค้างอยู่ ซึ่งบริษัทเลิกใช้มันไปตั้งแต่ปี 2003 แล้วไม่เคยลบทิ้ง และโค้ดใหม่ก็บังเอิญใช้ feature flag เก่าตัวเดิมซ้ำ ซึ่ง — บนเครื่องนั้นเครื่องเดียว — กลับไปเปิดรูทีนที่ตายไปแล้วให้ทำงานอีกครั้ง ตลอดราว 45

> นาที Knight ส่งคำสั่งซื้อขายที่ไม่ได้ตั้งใจนับล้านรายการเข้าตลาด และขาดทุนไป 460 ล้านดอลลาร์ มากกว่ามูลค่าทั้งบริษัทเสียอีก คำสั่งของ Securities and Exchange Commission บันทึกรายละเอียดเพิ่มอีกจุดหนึ่งไว้ว่า โค้ดที่หลับใหลตัวนี้ถูกแตะครั้งล่าสุดในปี 2005 ตอนที่ฟังก์ชันที่มันพึ่งพาอยู่ถูกย้ายไปที่อื่น แล้วไม่มีใครเทสต์ซ้ำว่าสองส่วนนี้ยังเข้ากันได้ไหม (SEC Release 34-70694, 2013) Knight จ่ายค่าปรับ 12 ล้านดอลลาร์ และอยู่ไม่ถึงสิ้นปีในฐานะบริษัทอิสระ

Not one line of Power Peg had changed in nine years. That is the whole difficulty of this floor. Software has no bearings to wear, no metal to fatigue, no parts to fall out of tolerance; a program left alone computes tomorrow exactly what it computed today. What moves is everything around it — the hardware beneath, the libraries beside it, the regulations above it, the people who understood it. Meir Lehman gave this its name in the 1970s as the first of his laws of software evolution: a program embedded in the real world must be continually adapted or it becomes progressively less satisfactory. His second law is the sting in the tail — as such a program evolves, its complexity increases unless deliberate work is done to hold it down (Lehman, *Proc. IEEE* 68(9), 1980). Rot is not decay of the artifact. It is drift of the ground.

> ไม่มีบรรทัดไหนของ Power Peg ที่เปลี่ยนไปเลยตลอดเก้าปี นั่นแหละคือความยากทั้งหมดของชั้นนี้ ซอฟต์แวร์ไม่มีตลับลูกปืนให้สึก ไม่มีโลหะให้ล้า ไม่มีชิ้นส่วนที่หลุดจากค่าคลาดเคลื่อนที่ยอมรับได้ โปรแกรมที่ถูกทิ้งไว้เฉยๆ จะคำนวณพรุ่งนี้ออกมาเหมือนกับที่มันคำนวณวันนี้เป๊ะ สิ่งที่เคลื่อนไหวคือทุกอย่างรอบตัวมันต่างหาก — ฮาร์ดแวร์ข้างล่าง ไลบรารีข้างๆ กฎระเบียบข้างบน คนที่เคยเข้าใจมัน Meir Lehman ตั้งชื่อปรากฏการณ์นี้ไว้ในทศวรรษ 1970 เป็นกฎข้อแรกในกฎวิวัฒนาการซอฟต์แวร์ของเขา — โปรแกรมที่ฝังตัวอยู่ในโลกจริงต้องถูกปรับตัวต่อเนื่อง ไม่งั้นมันจะค่อยๆ ใช้การได้แย่ลงเรื่อยๆ กฎข้อสองของเขาคือหนามที่ซ่อนอยู่ท้ายเรื่อง — เมื่อโปรแกรมแบบนี้วิวัฒนาการไป ความซับซ้อนของมันจะเพิ่มขึ้นเรื่อยๆ เว้นแต่จะมีคนตั้งใจลงมือกดมันไว้ (Lehman, Proc. IEEE 68(9), 1980) ผุ ไม่ใช่การเสื่อมสภาพของตัววัตถุ แต่คือการเคลื่อนของพื้นที่มันยืนอยู่

Which is why this discipline is not the study of writing programs. Google's engineers put the distinction in one line: *"software engineering is programming integrated over time"* (Winters, Manshreck & Wright, 2020). Programming produces something that works. Engineering produces something that can be made to work again, by someone else, after the ground has moved. A script you run once and throw away needs none of this floor. A payroll system running in 2040 needs all of it.

> นี่จึงเป็นเหตุผลว่าทำไมศาสตร์นี้ไม่ใช่การศึกษาเรื่องการเขียนโปรแกรม วิศวกรของ Google สรุปความต่างนี้ไว้ในประโยคเดียวว่า "วิศวกรรมซอฟต์แวร์คือการเขียนโปรแกรมที่ถูกอินทิเกรตไปตามเวลา" (software engineering is programming integrated over time) (Winters, Manshreck & Wright, 2020) การเขียนโปรแกรมผลิตสิ่งที่ทำงานได้ วิศวกรรมผลิตสิ่งที่ทำให้กลับมาทำงานได้อีกครั้ง โดยคนอื่น หลังจากพื้นที่มันยืนอยู่เคลื่อนไปแล้ว สคริปต์ที่รันครั้งเดียวแล้วทิ้งไม่ต้องการอะไรจากชั้นนี้เลย แต่ระบบจ่ายเงินเดือนที่ยังรันอยู่ในปี 2040 ต้องการทุกอย่างจากมัน

The first thing that floor must confront is that not all of the difficulty can be removed. Fred Brooks, who had managed IBM's OS/360 — the largest programming project of its era, and a famous struggle — split the work in two: the *essential* tasks, "the fashioning of the complex conceptual structures that compose the abstract software entity," and the *accidental* tasks, the labor of representing those structures in a language and getting them onto a machine (*No Silver Bullet*, 1986). Better tools attack the accidental half, and they have done so brilliantly — nobody now fights the compiler for a week. But the essential half is the specification of what the thing must actually do, tangled with everything else it must do, and no notation makes a tangled requirement simple. Brooks's prediction was that no single advance would deliver an order-of-magnitude improvement within a decade, because the part that remained was the part that was never about tooling.

> สิ่งแรกที่ชั้นนี้ต้องเผชิญคือความยากบางส่วนไม่มีทางกำจัดออกไปได้ Fred Brooks ผู้เคยบริหารโครงการ OS/360 ของ IBM — โครงการเขียนโปรแกรมที่ใหญ่ที่สุดในยุคนั้น และเป็นการดิ้นรนที่โด่งดัง — แบ่งงานออกเป็นสองส่วน คืองาน เนื้อแท้

> (essential) ซึ่งเป็น "การปั้นแต่งโครงสร้างเชิงมโนทัศน์ที่ซับซ้อนซึ่งประกอบกันขึ้นเป็นตัวตนซอฟต์แวร์เชิงนามธรรม" (the fashioning of the complex conceptual structures that compose the abstract software entity) และงาน บังเอิญ (accidental) ซึ่งคือแรงงานในการนำโครงสร้างเหล่านั้นไปแทนด้วยภาษาหนึ่งแล้วเอาขึ้นเครื่องจริงให้ได้ (No Silver Bullet, 1986) เครื่องมือที่ดีขึ้นโจมตีความซับซ้อนโดยบังเอิญ (accidental complexity) และมันก็ทำได้ยอดเยี่ยมมาก — ทุกวันนี้ไม่มีใครต้องสู้กับคอมไพเลอร์เป็นอาทิตย์อีกแล้ว แต่ความซับซ้อนโดยเนื้อแท้ (essential complexity) คือสเปกของสิ่งที่งานนั้นต้องทำจริงๆ ซึ่งพันกันยุ่งเหยิงกับทุกอย่างอื่นที่มันต้องทำด้วย ไม่มีสัญกรณ์ไหนทำให้ข้อกำหนด (requirement) ที่พันกันยุ่งกลายเป็นเรื่องง่ายได้ คำทำนายของ Brooks คือจะไม่มีความก้าวหน้าจุดเดียวที่ทำให้ดีขึ้นสิบเท่าได้ภายในสิบปี เพราะส่วนที่เหลืออยู่ไม่เคยเป็นเรื่องของเครื่องมือตั้งแต่แรก

If complexity cannot be removed, it can be *confined*, and this is the field's one great structural idea. David Parnas asked what makes a decomposition into modules good, and rejected the obvious answer — that modules should mirror the steps of the process. His criterion instead: *"Every module … is characterized by its knowledge of a design decision which it hides from all others. Its interface … was chosen to reveal as little as possible about its inner workings"* (Parnas, CACM 15(12), 1972). A module is a bet about what will change. Hide the decision that will change, and change stays local; expose it, and change propagates. Knight Capital's flag was exactly this failure made visible — a name shared between new code and code no one remembered, a coupling that existed in the machine but in nobody's head.

> ถ้าความซับซ้อนกำจัดไม่ได้ มันก็ยัง กักไว้ ได้ และนี่คือไอเดียเชิงโครงสร้างที่ยิ่งใหญ่หนึ่งเดียวของศาสตร์นี้ David Parnas ตั้งคำถามว่าอะไรทำให้การแตกระบบเป็นมอดูล (module) ดี แล้วปฏิเสธคำตอบที่ดูชัดเจนที่สุด — ว่ามอดูลควรสะท้อนขั้นตอนของกระบวนการ เกณฑ์ของเขากลับเป็นว่า มอดูลแต่ละตัวถูกกำหนดโดยการรู้ถึงการตัดสินใจเชิงออกแบบหนึ่งอย่างที่มันซ่อนไว้จากมอดูลอื่นทั้งหมด อินเทอร์เฟซของมันถูกเลือกให้เปิดเผยการทำงานภายในให้น้อยที่สุดเท่าที่จะทำได้ (Parnas, CACM 15(12), 1972) มอดูลคือการเดิมพันว่าอะไรจะเปลี่ยน ซ่อนการตัดสินใจที่จะเปลี่ยน แล้วการเปลี่ยนแปลงก็อยู่เฉพาะที่ เปิดเผยมันออกมา แล้วการ

> เปลี่ยนแปลงก็จะลามไปทั่ว feature flag ของ Knight Capital คือความล้มเหลวแบบนี้เป๊ะๆ ที่ปรากฏให้เห็น — ชื่อหนึ่งที่ใช้ร่วมกันระหว่างโค้ดใหม่กับโค้ดที่ไม่มีใครจำได้ การเกาะเกี่ยว (coupling) ที่มีอยู่จริงในเครื่อง แต่ไม่มีอยู่ในหัวใคร

The instrument that makes change survivable is the test, and its power is narrower than it looks. Dijkstra's corollary is exact: *"Program testing can be used to show the presence of bugs, but never to show their absence!"* (EWD249). A test suite proves nothing about the system as a whole. What it does is convert a hope — *this probably still works* — into a specific, mechanically re-checkable claim, and a large system is changeable in proportion to how many such claims it carries. Version control does the same job for history rather than behavior: since Git in April 2005 (Torvalds), a project's past is a queryable artifact, and the question "why is this line here" has an answer that outlives the person who wrote it.

> เครื่องมือที่ทำให้การเปลี่ยนแปลงรอดได้คือเทสต์ (test) และพลังของมันแคบกว่าที่มันดูเยอะ ข้อสรุปของ Dijkstra ตรงเป๊ะ "การเทสต์โปรแกรมใช้แสดงการมีอยู่ของบั๊กได้ แต่ไม่มีทางใช้แสดงว่าบั๊กไม่มีอยู่ได้เลย!" (Program testing can be used to show the presence of bugs, but never to show their absence!) (EWD249) ชุดเทสต์ (test suite) ไม่ได้พิสูจน์อะไรเกี่ยวกับระบบทั้งหมดเลย สิ่งที่มันทำคือแปลงความหวัง — น่าจะยังทำงานได้อยู่ — ให้กลายเป็นข้อกล่าวอ้างที่เจาะจงและตรวจซ้ำได้ด้วยกลไก และระบบใหญ่จะเปลี่ยนแปลงได้มากแค่ไหนก็ขึ้นกับว่ามันแบกข้อกล่าวอ้างแบบนี้ไว้กี่ข้อ ระบบควบคุมเวอร์ชัน (version control) ทำหน้าที่แบบเดียวกันนี้ให้กับประวัติศาสตร์แทนที่จะเป็นพฤติกรรม — นับตั้งแต่ Git ในเดือนเมษายน 2005 (Torvalds) อดีตของโปรเจกต์หนึ่งก็กลายเป็นวัตถุที่คิวรีได้ และคำถามว่า "ทำไมบรรทัดนี้ถึงอยู่ตรงนี้" ก็มีคำตอบที่อยู่ยืนยาวกว่าคนที่เขียนมันขึ้นมา

The last principle is the one practitioners discover last. Melvin Conway, writing in *Datamation* in April 1968: *"Any organization that designs a system ... will produce a design whose structure is a copy of the organization's communication structure"* (Conway's law). Two teams that rarely speak will build two components with a thin, brittle, under-specified interface

between them, because that interface is a faithful cast of their conversation. Architecture is therefore partly a staffing decision, and reorganizations are silent design changes.

> หลักการสุดท้ายคือหลักที่คนทำงานจริงมักค้นพบเป็นลำดับท้ายสุด Melvin Conway เขียนไว้ใน Datamation เดือนเมษายน 1968 ว่า "องค์กรใดก็ตามที่ออกแบบระบบหนึ่ง ... จะผลิตงานออกแบบที่มีโครงสร้างเป็นสำเนาของโครงสร้างการสื่อสารในองค์กรนั้น" (Any organization that designs a system ... will produce a design whose structure is a copy of the organization's communication structure) (Conway's law) สองทีมที่แทบไม่คุยกันจะสร้างสองส่วนประกอบที่มีอินเทอร์เฟซบางเบา เปราะ และกำหนดสเปกไว้ไม่ครบระหว่างกัน เพราะอินเทอร์เฟซนั้นคือแบบหล่อที่ซื่อตรงต่อบทสนทนาของพวกเขาเอง สถาปัตยกรรมจึงเป็นการตัดสินใจเรื่องอัตรากำลังคนส่วนหนึ่งด้วย และการปรับโครงสร้างองค์กรก็คือการเปลี่ยนแปลงงานออกแบบแบบเงียบๆ

The field named itself in that same year, at a NATO conference at Garmisch in October 1968, where "software engineering" was chosen deliberately to provoke — an assertion that this work *ought* to rest on theory and discipline the way established engineering does, made at a moment when it visibly did not (Naur & Randell, eds., 1969). Nearly sixty years on, the aspiration is still doing more work than the achievement, and the honest members of the field say so.

> ศาสตร์นี้ตั้งชื่อให้ตัวเองในปีเดียวกันนั้นเอง ที่การประชุม NATO ณ Garmisch เดือนตุลาคม 1968 ซึ่งคำว่า "software engineering" ถูกเลือกใช้อย่างตั้งใจให้ยั่วยุ — เป็นการยืนยันว่างานนี้ ควร วางอยู่บนทฤษฎีและวินัยแบบเดียวกับวิศวกรรมที่ตั้งตัวมานานแล้ว ทั้งที่ในตอนนั้นเห็นได้ชัดว่ามันยังไม่เป็นแบบนั้นเลย (Naur & Randell, eds., 1969) เกือบหกสิบปีผ่านไป ความมุ่งหวังก็ยังคงทำงานหนักกว่าความสำเร็จจริง และคนในวงการที่ซื่อตรงก็ยอมรับเรื่องนี้

Connections · เชื่อมโยง

Below this floor sit chapters 05 and 06: the operating system and the language give you a program that runs correctly once, and everything here begins after that. Chapter 02 supplies correctness for a single algorithm; this chapter is what happens when ten thousand of them must be changed by people who never met. Above and beside it, chapters 08, 09 and 10 are the domains where these disciplines are tested hardest, because a database, a network stack, or a distributed system cannot be stopped and rewritten. Chapter 11 is the case where a leak is not merely expensive but adversarial.

> ชั้นล่างของชั้นนี้คือบทที่ 05 และ 06 ระบบปฏิบัติการกับภาษาโปรแกรมให้โปรแกรมที่รันถูกต้องได้หนึ่งครั้ง แล้วทุกอย่างในบทนี้ก็เริ่มต้นหลังจากจุดนั้น บทที่ 02 มอบความถูกต้องให้กับอัลกอริทึมเดี่ยวๆ หนึ่งตัว ส่วนบทนี้คือสิ่งที่เกิดขึ้นเมื่ออัลกอริทึมแบบนั้นหมื่นตัวต้องถูกเปลี่ยนแปลงโดยคนที่ไม่เคยเจอกันมาก่อน ชั้นบนและข้างๆ บทที่ 08, 09 และ 10 คือโดเมนที่ศาสตร์เหล่านี้ถูกทดสอบหนักที่สุด เพราะฐานข้อมูล เครือข่าย หรือระบบกระจาย ไม่สามารถหยุดแล้วเขียนใหม่ทั้งหมดได้ บทที่ 11 คือกรณีที่การรั่วไม่ใช่แค่มีต้นทุนแพง แต่ยังเป็นปฏิปักษ์อีกด้วย

Open questions · คำถามที่ยังเปิดอยู่

The field's own evidence base is thin. Everyone quotes a figure for how much of software's cost is maintenance rather than initial construction, and the figures range from sixty to over ninety percent with no shared definition or measurement window behind them — a discipline that measures its central claim that loosely should be uneasy. Deeper: does machine-generated code shift Brooks's ratio, or does it merely make the accidental half cheaper while leaving the essential tangle untouched — and if reading and reviewing code becomes the bottleneck, is that a new floor or the same

one? And if Conway's law holds, is it usable in reverse: can you get the architecture you want by first building the organization that would produce it?

> ฐานหลักฐานของศาสตร์นี้เองก็ยังบางมาก ทุกคนอ้างตัวเลขว่าต้นทุนซอฟต์แวร์กี่เปอร์เซ็นต์เป็นการบำรุงรักษา (maintenance) แทนที่จะเป็นการสร้างครั้งแรก แล้วตัวเลขก็แกว่งตั้งแต่หกสิบไปจนถึงเกินเก้าสิบเปอร์เซ็นต์ โดยไม่มีนิยามร่วมหรือหน้าต่างการวัดที่ชัดเจนอยู่เบื้องหลังเลย — ศาสตร์ที่วัดข้อกล่าวอ้างหลักของตัวเองหลวมขนาดนี้ควรจะรู้สึกไม่สบายใจ ลึกลงไปอีก โค้ดที่เครื่องผลิตขึ้นเองเปลี่ยนสัดส่วนของ Brooks จริงไหม หรือมันแค่ทำให้ครึ่งบังเอิญถูกลงโดยไม่แตะครึ่งเนื้อแท้ที่ยังพันกันยุ่งอยู่ — และถ้าการอ่านกับรีวิวโค้ดกลายเป็นคอขวดขึ้นมา นั่นคือชั้นใหม่หรือชั้นเดิม? และถ้ากฎของคอนเวย์ (Conway's law) เป็นจริง มันใช้ย้อนกลับได้ไหม — คุณสร้างสถาปัตยกรรมที่ต้องการได้ไหม ด้วยการสร้างองค์กรที่จะผลิตสถาปัตยกรรมแบบนั้นขึ้นมาก่อน?

Go deeper · อ่านต่อ

- Titus Winters, Tom Manshreck & Hyrum Wright, *Software Engineering at Google*, O'Reilly, 2020 — the clearest modern statement of the time-integrated view; free in full at abseil.io.

  > Titus Winters, Tom Manshreck & Hyrum Wright, Software Engineering at Google, O'Reilly, 2020 — คำอธิบายยุคใหม่ที่ชัดเจนที่สุดของมุมมองแบบอินทิเกรตไปตามเวลา อ่านฟรีฉบับเต็มได้ที่ abseil.io

- D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the* ACM 15, no. 12 (1972): 1053–1058 — full text.

  > D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM 15, no. 12 (1972): 1053–1058 — ฉบับเต็ม

- Frederick P. Brooks, Jr., "No Silver Bullet — Essence and Accident in Software Engineering," IFIP Congress, 1986 — UNC technical report.

> Frederick P. Brooks, Jr., "No Silver Bullet — Essence and Accident in Software Engineering," IFIP Congress, 1986 — รายงานทางเทคนิคของ UNC

CHAPTER 08

Databases

ฐานข้อมูล

This chapter is the floor that turns storage into memory — the layer between raw persistence below and every application above that needs to be told the truth.

บทนี้คือชั้นที่เปลี่ยนพื้นที่เก็บข้อมูล (storage) ให้กลายเป็นความจำ — ชั้นที่คั่นระหว่างความคงอยู่ของข้อมูล (persistence) แบบดิบด้านล่าง กับทุกแอปพลิเคชันด้านบนที่ต้องได้รับความจริง

How does civilization remember reliably?

อารยธรรมจดจำสิ่งต่างๆ ได้อย่างน่าเชื่อถือได้อย่างไร

Before 1970, asking a computer a question it had not been built to answer meant writing a new program, and that program had to know where the answer physically lived: which file, in which order, chained to which other record by which pointer. Reorganize the storage to make one report faster and every program that touched those records broke. Businesses were accumulating data far faster than they could accumulate programmers to chase it around the disk.

> ก่อนปี 1970 การถามคำถามกับคอมพิวเตอร์ที่มันไม่ได้ถูกสร้างมาให้ตอบ หมายถึงต้องเขียนโปรแกรมใหม่ และโปรแกรมนั้นต้องรู้ด้วยว่าคำตอบอยู่ที่ไหนในทางกายภาพ — ไฟล์ไหน เรียงลำดับแบบไหน ผูกกับ record อื่นด้วย pointer ตัวไหน จัดโครงสร้างที่เก็บข้อมูลใหม่เพื่อให้รายงานหนึ่งเร็วขึ้น ทุกโปรแกรมที่แตะ record เหล่านั้นก็พังตาม ธุรกิจสะสมข้อมูลเร็วกว่าที่จะหาโปรแกรมเมอร์มาไล่ตามมันไปทั่วดิสก์ได้ทัน

Then an IBM mathematician at the San Jose Research Laboratory published a paper whose first sentence was a declaration of independence: *"Future users of large data banks must be protected from having to know how the data*

is organized in the machine (the internal representation)" (Codd, CACM 13(6), 1970). Edgar Codd's proposal was to model data as *relations* — mathematical sets of tuples, with no order, no pointers, no physical trace of the disk in them at all. What a user sees is a table. What a table is made of is nobody's business but the system's.

> จากนั้นนักคณิตศาสตร์ของ IBM ที่ San Jose Research Laboratory ตีพิมพ์เปเปอร์ฉบับหนึ่งซึ่งประโยคแรกคือคำประกาศอิสรภาพ: "ผู้ใช้ธนาคารข้อมูลขนาดใหญ่ในอนาคตต้องได้รับการปกป้องจากการต้องรู้ว่าข้อมูลถูกจัดระเบียบอย่างไรในเครื่อง (internal representation)" ("Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation)") (Codd, CACM 13(6), 1970) ข้อเสนอของ Edgar Codd คือการสร้างแบบจำลองข้อมูลให้เป็น relations — เซตทางคณิตศาสตร์ของ tuple ที่ไม่มีลำดับ ไม่มี pointer ไม่มีร่องรอยทางกายภาพของดิสก์หลงเหลืออยู่เลย สิ่งที่ผู้ใช้เห็นคือตาราง ส่วนตารางประกอบขึ้นจากอะไรนั้นไม่ใช่เรื่องของใครนอกจากระบบเอง

That is the abstraction this floor sells, and the name for it is **data independence**. Everything else in the chapter exists to make it affordable.

> นั่นคือนามธรรม (abstraction) ที่ชั้นนี้ขาย และชื่อของมันคือ **data independence** ทุกอย่างที่เหลือในบทนี้มีอยู่เพื่อทำให้มันมีต้นทุนที่จ่ายไหว

It is expensive because a question phrased against an idealized table gives the machine no plan. Codd saw the consequence and stated it precisely: the power of such a query language *"lies in its descriptive ability (not its computing ability)*" — you describe the answer you want, not the route to it. Somebody must still supply the route. That somebody became the **query optimizer**, and the technique that made it work was published by Pat Selinger's group in 1979 out of IBM's System R prototype: enumerate the possible access paths, estimate the cost of each from statistics about the data, and pick the cheapest (Selinger et al., SIGMOD 1979). It is an unglamorous idea with an enormous consequence — the reason a modern analyst can write nine lines of SQL over a billion rows and get an answer, while the

equivalent hand-written traversal would take a week and be wrong. System R also produced the language: Donald Chamberlin and Raymond Boyce's SEQUEL, in 1974 (SIGFIDET workshop), later renamed SQL because another company already held the trademark on the word.

> มันมีต้นทุนสูง เพราะคำถามที่เขียนขึ้นเทียบกับตารางในอุดมคติไม่ได้บอกแผนการทำงานอะไรกับเครื่องเลย Codd มองเห็นผลที่ตามมาและพูดไว้ชัดเจนว่า พลังของภาษาคิวรี (query) แบบนี้ "อยู่ที่ความสามารถในการบรรยาย (ไม่ใช่ความสามารถในการคำนวณ)" — คุณบรรยายคำตอบที่ต้องการ ไม่ใช่เส้นทางไปหามัน แต่ต้องมีใครสักคนจัดหาเส้นทางนั้นให้อยู่ดี คนนั้นกลายมาเป็น **query optimizer** และเทคนิคที่ทำให้มันทำงานได้ถูกตีพิมพ์โดยกลุ่มของ Pat Selinger ในปี 1979 จากต้นแบบ System R ของ IBM: ไล่ดูเส้นทางการเข้าถึงข้อมูลที่เป็นไปได้ทั้งหมด ประเมินต้นทุนของแต่ละเส้นทางจากสถิติของข้อมูล แล้วเลือกเส้นทางที่ถูกที่สุด (Selinger et al., SIGMOD 1979) เป็นไอเดียที่ดูไม่หวือหวาแต่ส่งผลมหาศาล — เหตุผลที่นักวิเคราะห์ยุคนี้เขียน SQL แค่เก้าบรรทัดบนข้อมูลพันล้านแถวแล้วได้คำตอบ ในขณะที่การไล่ค้นด้วยมือแบบเทียบเท่ากันจะใช้เวลาเป็นสัปดาห์และยังผิดอีกด้วย System R ยังให้กำเนิดภาษาด้วย: SEQUEL ของ Donald Chamberlin กับ Raymond Boyce ในปี 1974 (SIGFIDET workshop) ซึ่งภายหลังเปลี่ยนชื่อเป็น SQL เพราะบริษัทอื่นถือเครื่องหมายการค้าของคำนั้นไปแล้ว

The second thing this floor hides is **failure**. Power dies mid-write; two clerks sell the same seat; a disk returns a page from last Tuesday. The abstraction offered is the *transaction*: a group of operations that either all happen or none do, that behaves as though no one else were running at the same time, and that once confirmed cannot be lost. Jim Gray set out these guarantees in 1981 (*The Transaction Concept*, VLDB); Theo Härder and Andreas Reuter gave them the acronym everyone now recites — atomicity, consistency, isolation, durability — in a 1983 survey that credits Gray for the substance (*ACM Computing Surveys* 15(4)). Note what a transaction really is:

not a mechanism for avoiding failure, but a *contract about what failure is allowed to look like from above*. The machinery underneath fails constantly. The surface it offers admits only two outcomes.

> สิ่งที่สองที่ชั้นนี้ซ่อนไว้คือ **ความล้มเหลว** (failure) ไฟดับกลางคันขณะเขียนข้อมูล เสมียนสองคนขายที่นั่งเดียวกัน ดิสก์คืนหน้าข้อมูลของวันอังคารที่แล้วกลับมา สิ่งที่เสนอให้แทนคือ ทรานแซกชัน (transaction): กลุ่มของการปฏิบัติการที่จะเกิดขึ้นทั้งหมดหรือไม่เกิดขึ้นเลย ทำงานราวกับไม่มีใครอื่นทำงานพร้อมกันอยู่ในเวลาเดียวกัน และเมื่อยืนยันแล้วจะไม่มีวันสูญหาย Jim Gray วางการรับประกันเหล่านี้ไว้ในปี 1981 (The Transaction Concept, VLDB) ส่วน Theo Härder กับ Andreas Reuter ตั้งชื่อย่อที่ทุกคนท่องกันจนถึงทุกวันนี้ — atomicity, consistency, isolation, durability — ไว้ในงานสำรวจปี 1983 ที่ให้เครดิตเนื้อหาทั้งหมดแก่ Gray (ACM Computing Surveys 15(4)) สังเกตว่าทรานแซกชันคืออะไรกันแน่: มันไม่ใช่กลไกสำหรับหลีกเลี่ยงความล้มเหลว แต่เป็น สัญญาว่าความล้มเหลวได้รับอนุญาตให้มีหน้าตาแบบไหนเมื่อมองจากด้านบน กลไกด้านล่างพังอยู่ตลอดเวลา ส่วนผิว (surface) ที่มันยื่นให้ยอมรับผลลัพธ์ได้แค่สองแบบเท่านั้น

The mechanism that delivers that contract is the log, and it is the least intuitive idea on this floor. Durability is not achieved by writing the data carefully. It is achieved by writing the *intention* first, sequentially, to a file that is only ever appended — and only then touching the data pages, whenever it is convenient. If the machine dies, recovery reads the log and replays history: redo what was committed, undo what was not. The canonical treatment is ARIES (Mohan et al., *ACM TODS* 17(1), 1992), and the pattern escaped the database long ago — filesystems journal, message queues are logs wearing a different hat, and replication in chapter 10 is one machine reading another machine's log. The database's real heart is not the table. It is the append-only record of what it was told to do.

> กลไกที่ส่งมอบสัญญานั้นคือ log และมันเป็นไอเดียที่เข้าใจยากที่สุดในชั้นนี้ ความทนทาน (durability) ไม่ได้เกิดจากการเขียนข้อมูลอย่างระมัดระวัง แต่เกิดจากการเขียน เจตนา ก่อน เป็นลำดับ ลงไฟล์ที่มีแต่การต่อท้าย (append) เท่านั้น แล้วค่อยไปแตะหน้าข้อมูลจริงเมื่อไหร่ก็ได้ที่สะดวก ถ้าเครื่องตาย การกู้คืนจะอ่าน log แล้วเล่น

> ประวัติศาสตร์ซ้ำ: ทำสิ่งที่ commit แล้วซ้ำอีกครั้ง (redo) และย้อนสิ่งที่ยังไม่ commit กลับ (undo) แนวทางมาตรฐานคือ ARIES (Mohan et al., ACM TODS 17(1), 1992) และแพตเทิร์นนี้หลุดออกไปจากฐานข้อมูลนานแล้ว — filesystem ทำ journal, message queue ก็คือ log ที่สวมหน้ากากคนละแบบ และการทำสำเนา (replication) ในบทที่ 10 ก็คือเครื่องหนึ่งอ่าน log ของอีกเครื่องหนึ่ง หัวใจจริงของฐานข้อมูลไม่ใช่ตาราง แต่คือบันทึกแบบต่อท้ายอย่างเดียว (append-only) ของสิ่งที่มันถูกสั่งให้ทำ

The one place the physical world is allowed to show through is the **index**, and the shape it takes is dictated by hardware two floors down. Rudolf Bayer and Edward McCreight's B-tree, from a Boeing report in 1970 and *Acta Informatica* in 1972 (Bayer & McCreight), is deliberately wide and shallow — hundreds of children per node, so that one node fills one disk page and a lookup among billions of rows costs three or four fetches instead of thirty. It is chapter 02's bargain, paid at industrial scale: build the structure now, in space and in write cost, to make a future question cheap. And it is where the abstraction leaks. Two logically identical queries can differ by four orders of magnitude in time because one has an index and the other does not — the surface says *what*, but the bill is denominated in *how*.

> จุดเดียวที่โลกทางกายภาพได้รับอนุญาตให้โผล่ผ่านมาคือ **ดัชนี (index)** และรูปทรงของมันถูกกำหนดโดยฮาร์ดแวร์ที่อยู่สองชั้นด้านล่าง B-tree ของ Rudolf Bayer กับ Edward McCreight จากรายงานของ Boeing ปี 1970 และ Acta Informatica ปี 1972 (Bayer & McCreight) ถูกออกแบบให้กว้างและตื้นโดยตั้งใจ — แต่ละโหนดมีลูกได้เป็นร้อย จนหนึ่งโหนดเต็มพอดีหนึ่งหน้าดิสก์ และการค้นหาท่ามกลางข้อมูลนับพันล้านแถวเสียแค่สามหรือสี่ครั้งในการดึงข้อมูล แทนที่จะเป็นสามสิบครั้ง มันคือข้อแลกเปลี่ยน (trade-off) ของบทที่ 02 ที่จ่ายในระดับอุตสาหกรรม: สร้างโครงสร้างไว้ตั้งแต่ตอนนี้ ทั้งในแง่พื้นที่และต้นทุนการเขียน เพื่อให้คำถามในอนาคตมีต้นทุนถูกลง และนี่คือจุดที่ abstraction รั่ว คิวรี (query) สองตัวที่เหมือนกันทุกประการในเชิงตรรกะ อาจใช้เวลาต่างกันถึงสี่ระดับขนาด (order of magnitude) เพราะตัวหนึ่งมี index ส่วนอีกตัวไม่มี — ผิว (surface) บอกแค่ อะไร แต่บิลคิดเป็น อย่างไร

The people are worth a moment. Codd's employer was slow to build what he described, having a large and profitable investment in the older hierarchical system it would replace (Wade & Chamberlin, *IEEE Annals*, 2012); the relational model reached customers partly through others, and Codd received the Turing Award in 1981 for a paper his own company had spent a decade arguing with (ACM).

> เรื่องของคนคุ้มค่าที่จะแวะดูสักครู่ นายจ้างของ Codd เองก็ช้าที่จะสร้างสิ่งที่เขาบรรยายไว้ เพราะมีเงินลงทุนก้อนใหญ่ที่ยังทำกำไรอยู่ในระบบ hierarchical รุ่นเก่าที่มันจะเข้ามาแทนที่ (Wade & Chamberlin, IEEE Annals, 2012) แบบจำลองเชิงสัมพันธ์ (relational model) ไปถึงมือลูกค้าส่วนหนึ่งผ่านทางคนอื่น และ Codd ได้รับรางวัล Turing Award ในปี 1981 จากเปเปอร์ที่บริษัทของเขาเองเถียงค้านมันมานานนับสิบปี (ACM)

Which brings the story to the loosening. Around 2006–2007 two systems built for a scale the relational contract could not reach were published — Google's Bigtable (OSDI 2006) and Amazon's Dynamo (SOSP 2007) — and in June 2009 a San Francisco meetup gave the movement a name it would spend a decade regretting: NoSQL. The trade was explicit: give up joins, give up strong transactions across machines, and buy horizontal scale with the proceeds. What happened next is the interesting part. Having sold the guarantees, the industry discovered how much application code it takes to fake them, and most of these systems have been growing transactions and query languages back ever since. Chapter 10 explains why the guarantees were expensive in the first place — and why they can never be free once the data lives on more than one machine.

> เรื่องราวมาถึงช่วงที่มันคลายตัวลง ราวปี 2006–2007 มีระบบสองตัวที่สร้างมาเพื่อรองรับสเกลที่สัญญาเชิงสัมพันธ์ไปไม่ถึงถูกตีพิมพ์ — Bigtable ของ Google (OSDI 2006) กับ Dynamo ของ Amazon (SOSP 2007) — และในเดือนมิถุนายน 2009 งาน meetup ที่ San Francisco ก็ตั้งชื่อให้ขบวนการนี้ ชื่อที่มันจะต้องมานั่งเสียใจไปอีกสิบปี: NoSQL การแลกเปลี่ยนนั้นชัดเจน: ยอม join ไม่ได้ ยอมไม่มีทรานแซกชัน (transaction) ที่แข็งแรงข้ามเครื่อง แล้วซื้อสเกลแนวนอนด้วยสิ่งที่ยอมแลกไป ส่วนที่น่าสนใจคือสิ่งที่เกิดขึ้นต่อมา พอขายการรับประกันเหล่านั้นไปแล้ว วงการก็ค้นพบ

> ว่าต้องเขียนโค้ดในแอปพลิเคชันมากแค่ไหนถึงจะปลอมมันขึ้นมาใหม่ได้ และระบบพวกนี้ส่วนใหญ่ก็ค่อยๆ งอกทรานแซกชันกับภาษาคิวรีกลับคืนมาตั้งแต่นั้น บทที่ 10 อธิบายว่าทำไมการรับประกันเหล่านี้ถึงมีต้นทุนแพงตั้งแต่แรก — และทำไมมันจะไม่มีวันฟรีได้เลย เมื่อข้อมูลอยู่บนเครื่องมากกว่าหนึ่งเครื่อง

Connections · เชื่อมโยง

Below: chapter 04 supplies the memory hierarchy that gives the B-tree its shape, and chapter 05 supplies the file, the process, and the crash-consistency problem in raw form — the database is what happens when one program takes those duties from the operating system and does them properly. Chapter 02 owns the data structures; chapter 03 explains why compressed columns and encoded pages work at all. Above: chapter 07 lives here permanently, since a schema is the hardest thing in any system to change; chapter 09 puts the client across a network; chapter 10 asks what survives of these guarantees when the machine is really many machines.

> ด้านล่าง: บทที่ 04 ให้ลำดับชั้นหน่วยความจำ (memory hierarchy) ที่กำหนดรูปทรงของ B-tree ส่วนบทที่ 05 ให้ไฟล์ โพรเซส (process) และปัญหา crash consistency มาในรูปดิบๆ — ฐานข้อมูลคือสิ่งที่เกิดขึ้นเมื่อโปรแกรมหนึ่งรับหน้าที่เหล่านั้นมาจากระบบปฏิบัติการแล้วทำมันอย่างถูกต้องจริงจัง บทที่ 02 เป็นเจ้าของโครงสร้างข้อมูล บทที่ 03 อธิบายว่าทำไมคอลัมน์ที่บีบอัดแล้วกับหน้าข้อมูลที่เข้ารหัสแล้วถึงทำงานได้ตั้งแต่แรก ด้านบน: บทที่ 07 อยู่ที่นี่ถาวร เพราะสคีมา (schema) คือสิ่งที่แก้ไขยากที่สุดในระบบใดๆ ก็ตาม บทที่ 09 พา client ข้ามเครือข่ายไป บทที่ 10 ถามว่าการรับประกันเหล่านี้เหลืออะไรอยู่บ้าง เมื่อเครื่องจริงๆ แล้วคือเครื่องหลายเครื่อง

Open questions · คำถามที่ยังเปิดอยู่

Is the relational model a permanent floor or a fifty-year local maximum? Its critics have twice declared it obsolete — for objects in the 1990s, for scale in the 2000s — and it has twice absorbed the challenger. Second: the optimizer is now the least predictable component in most systems, capable of choosing catastrophically on stale statistics; learned optimizers are an active research front, but a mis-estimating model is harder to debug than a

mis-estimating formula. Third, and least settled: the log is the one component every modern data system has converged on. If storage, queues, replication and stream processing are all logs underneath, is "the database" still the right unit of thought, or is it the log that is the floor and the database merely one view built over it?

> แบบจำลองเชิงสัมพันธ์ (relational model) เป็นชั้นถาวร หรือเป็นแค่จุดสูงสุดเฉพาะที่ (local maximum) ที่อยู่มาได้ห้าสิบปีกันแน่ ผู้วิจารณ์เคยประกาศว่ามันล้าสมัยมาแล้วสองครั้ง — เรื่อง object ในทศวรรษ 1990 เรื่องสเกลในทศวรรษ 2000 — และมันก็กลืนคู่แข่งเข้าไปได้ทั้งสองครั้ง ข้อสอง: query optimizer ตอนนี้เป็นส่วนประกอบที่คาดเดาไม่ได้มากที่สุดในระบบส่วนใหญ่ สามารถเลือกผิดพลาดร้ายแรงได้จากสถิติที่ล้าสมัย optimizer ที่เรียนรู้ได้ (learned optimizer) เป็นแนวรบวิจัยที่กำลังคึกคัก แต่โมเดลที่ประเมินผิดนั้นดีบั๊กยากกว่าสูตรที่ประเมินผิด ข้อสามและยังไม่ลงตัวที่สุด: log คือส่วนประกอบเดียวที่ระบบข้อมูลสมัยใหม่ทุกตัวมาบรรจบกัน ถ้า storage, queue, replication และ stream processing ล้วนเป็น log อยู่ข้างใต้ทั้งหมด แล้ว "ฐานข้อมูล" ยังเป็นหน่วยคิดที่ถูกต้องอยู่หรือเปล่า หรือว่า log ต่างหากที่เป็นชั้น ส่วนฐานข้อมูลเป็นแค่มุมมองหนึ่งที่สร้างซ้อนอยู่ข้างบนมัน

Go deeper · อ่านต่อ

— E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the* ACM 13, no. 6 (1970): 377–387 — twelve pages that reorganized an industry; ACM DL.

> E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM 13, no. 6 (1970): 377–387 — สิบสองหน้าที่จัดระเบียบทั้งอุตสาหกรรมใหม่; ACM DL.

— Peter Bailis, Joseph M. Hellerstein & Michael Stonebraker, eds., *Readings in Database Systems*, 5th ed., 2015 — the field's own curated canon, with editorial commentary; free at redbook.io.

> Peter Bailis, Joseph M. Hellerstein & Michael Stonebraker, eds., Readings in Database Systems, 5th ed., 2015 — คัมภีร์ที่วงการคัดสรรเอง พร้อมคำอธิบายจากบรรณาธิการ; อ่านฟรีที่ redbook.io.

— Abraham Silberschatz, Henry F. Korth & S. Sudarshan, *Database System Concepts*, 7th ed., McGraw-Hill, 2019 — the standard textbook if you want the mechanisms in full.

> Abraham Silberschatz, Henry F. Korth & S. Sudarshan, Database System Concepts, 7th ed., McGraw-Hill, 2019 — ตำราเรียนมาตรฐาน ถ้าต้องการกลไกแบบเต็มรูปแบบ.

CHAPTER 09

Networks

เครือข่าย

This chapter is the floor that dissolves the boundary of the single machine — everything above it can address something that is not itself.

บทนี้คือชั้นที่ละลายขอบเขตของเครื่องเดี่ยว — ทุกอย่างที่อยู่เหนือชั้นนี้สามารถอ้างถึงบางสิ่งที่ไม่ใช่ตัวมันเองได้

How do machines that share nothing agree to talk?

เครื่องที่ไม่มีอะไรร่วมกันเลยตกลงกันคุยกันได้อย่างไร

In October 1986, two sites four hundred yards apart stopped being able to talk to each other. Van Jacobson recorded the numbers: between Lawrence Berkeley Laboratory and the University of California at Berkeley — "separated by 400 yards and two IMP hops" — throughput fell from 32 Kbps to 40 bps (Jacobson & Karels, SIGCOMM '88). A factor of a thousand, across a distance a person could walk in four minutes. Nothing had broken. No cable was cut, no machine had failed, no attacker was present. The network had simply become so busy retransmitting the packets it was already failing to deliver that it left no room to deliver anything. It had eaten itself, and it would do so repeatedly for the next two years.

> ในเดือนตุลาคม 1986 สองสถานที่ที่ห่างกันสี่ร้อยหลาคุยกันไม่ได้อีกต่อไป Van Jacobson บันทึกตัวเลขไว้ว่า ระหว่าง Lawrence Berkeley Laboratory กับ University of California ที่ Berkeley — "ห่างกัน 400 หลา และสองช่วง IMP hop" ("separated by 400 yards and two IMP hops") — ปริมาณงานต่อเวลา (throughput) ตกจาก 32 Kbps เหลือ 40 bps (Jacobson & Karels, SIGCOMM '88) ต่างกันถึงพันเท่า ข้ามระยะทางที่คนเดินได้ในสี่นาที ไม่มีอะไรพัง ไม่มีสายถูกตัด ไม่มีเครื่องไหนล่ม ไม่มีผู้โจมตีอยู่ในเหตุการณ์ เครือข่ายแค่ยุ่งอยู่กับการส่งซ้ำ

(retransmission) แพ็กเก็ต (packet) ที่มันส่งไม่สำเร็จอยู่แล้วมากจนไม่เหลือที่ว่างให้ส่งอะไรได้อีกเลย มันกินตัวเองไปแล้ว และมันจะทำแบบนั้นซ้ำอีกไปอีกสองปีข้างหน้า

That episode is the best introduction to this floor, because it shows what kind of thing a network is. Not a wire. An agreement — with no authority to enforce it.

เหตุการณ์นั้นคือบทนำที่ดีที่สุดสำหรับชั้นนี้ เพราะมันแสดงให้เห็นว่าเครือข่าย (network) คือของแบบไหนกันแน่ ไม่ใช่สายไฟ แต่เป็นข้อตกลง — ที่ไม่มีอำนาจใดบังคับให้ทำตาม

The agreement starts with a decision made twice, independently, in the early 1960s. The telephone system worked by *circuit switching*: reserve a path end to end, hold it for the duration, and the conversation gets a private wire. Paul Baran at RAND, designing a communications system that could survive the loss of much of itself, proposed instead that messages be chopped into standard blocks that each find their own way and be reassembled at the far end (*On Distributed Communications*, RM-3420-PR, 1964). Donald Davies, at Britain's National Physical Laboratory, arrived at the same design for a different reason — computer traffic is bursty, and a reserved circuit sits idle most of the time — and gave the blocks the name that stuck: *packets*. Packet switching trades a guarantee for a statistic: no conversation is promised a path, all conversations share all paths, and they share well precisely because none talks continuously. The first message on the resulting ARPANET went from UCLA toward Stanford Research Institute on 29 October 1969; the operator typed the first two letters of "LOGIN" and the far end crashed (Kleinrock's account).

ข้อตกลงนี้เริ่มจากการตัดสินใจที่เกิดขึ้นสองครั้งแยกจากกัน ในช่วงต้นทศวรรษ 1960 ระบบโทรศัพท์ทำงานด้วย circuit switching: จองเส้นทางไว้ทั้งสาย ยึดครองมันไว้ตลอดการสนทนา แล้วบทสนทนาก็ได้สายส่วนตัวไปใช้ Paul Baran ที่ RAND ซึ่งกำลังออกแบบระบบสื่อสารที่รอดได้แม้ส่วนใหญ่ของตัวมันจะถูกทำลายไป กลับเสนอให้สับข้อความเป็นก้อนมาตรฐาน ให้แต่ละก้อนหาทางไปเองแล้วค่อยประกอบ

> กลับที่ปลายทาง (On Distributed Communications, RM-3420-PR, 1964) Donald Davies ที่ National Physical Laboratory ของอังกฤษ มาถึงการออกแบบเดียวกันด้วยเหตุผลคนละอย่าง — ทราฟฟิกของคอมพิวเตอร์มาเป็นห้วงๆ (bursty) และวงจรที่จองไว้ก็ว่างเปล่าอยู่เกือบตลอดเวลา — และเขาเป็นคนตั้งชื่อก้อนพวกนี้ที่ติดปากมาจนถึงทุกวันนี้ว่า แพ็กเก็ต (packet) packet switching แลกการรับประกันด้วยค่าสถิติแทน: ไม่มีบทสนทนาไหนถูกสัญญาว่าจะได้เส้นทางแน่นอน ทุกบทสนทนาใช้ทุกเส้นทางร่วมกัน และมันใช้ร่วมกันได้ดีก็เพราะไม่มีใครพูดต่อเนื่องตลอดเวลา ข้อความแรกบน ARPANET ที่เกิดขึ้นจากทั้งหมดนี้ ถูกส่งจาก UCLA ไปยัง Stanford Research Institute เมื่อวันที่ 29 ตุลาคม 1969 ผู้ควบคุมพิมพ์อักษรสองตัวแรกของ "LOGIN" แล้วปลายทางก็ล่ม (Kleinrock's account)

The second structural decision is **layering** — the tower recapitulated inside one chapter. Each layer takes a job from above, does one part of it, and hands the rest down, knowing nothing of the implementation beneath. The formal version is the seven-layer OSI Basic Reference Model, standardized in 1984 (ISO/IEC 7498-1); the version the world runs is smaller and messier, four layers written down after the fact in RFC 1122 (1989). Keep that embarrassment in mind: the model designed by committee is the one taught in courses, and the model grown by implementers is the one carrying this sentence to you.

> การตัดสินใจเชิงโครงสร้างข้อที่สองคือ **การแบ่งชั้น (layering)** — หอคอย (tower) ทั้งใบถูกย่อลงมาไว้ในบทเดียว แต่ละชั้นรับงานมาจากชั้นบน ทำแค่ส่วนของตัวเอง แล้วส่งที่เหลือลงไปข้างล่าง โดยไม่รู้อะไรเลยเกี่ยวกับการ implement ที่อยู่ข้างใต้ เวอร์ชันที่เป็นทางการคือ OSI Basic Reference Model เจ็ดชั้น ที่ถูกกำหนดเป็นมาตรฐานในปี 1984 (ISO/IEC 7498-1) ส่วนเวอร์ชันที่โลกจริงใช้งานอยู่นั้นเล็กกว่าและรกกว่า มีสี่ชั้น เขียนขึ้นภายหลังจากที่มันเกิดขึ้นจริงแล้วใน RFC 1122 (1989) จำความน่าอายนี้ไว้ให้ดี: โมเดลที่คณะกรรมการออกแบบคือโมเดลที่ถูกสอนในห้องเรียน ส่วนโมเดลที่งอกขึ้นมาจากมือของคนที่ implement จริงคือตัวที่กำลังพาประโยคนี้มาถึงคุณ

What makes the grown one work is the shape of its middle. The Internet Protocol offers almost nothing: it will attempt to deliver a packet toward an address, and it may drop it, duplicate it, delay it, or deliver it out of order without apology. Every network technology beneath can be different — fiber, radio, satellite, copper — and every application above can be different, and they meet at one deliberately impoverished layer. The principle behind that poverty was named by Jerome Saltzer, David Reed and David Clark: a function *"can completely and correctly be implemented only with the knowledge and help of the application standing at the end points … providing that questioned function as a feature of the communication system itself is not possible"* (end-to-end arguments, 1981/1984). Put the intelligence at the edges. A network that tried to guarantee delivery would still not spare an application from checking, and would tax every application that did not care.

> สิ่งที่ทำให้เวอร์ชันที่งอกขึ้นมานี้ทำงานได้คือรูปทรงของส่วนกลางของมัน Internet Protocol ให้อะไรแทบไม่ได้เลย: มันจะพยายามส่งแพ็กเก็ตไปยังที่อยู่หนึ่ง และมันอาจทำแพ็กเก็ตหาย ทำซ้ำ หน่วงเวลา หรือส่งถึงแบบผิดลำดับโดยไม่ขอโทษใคร เทคโนโลยีเครือข่ายทุกแบบที่อยู่ข้างล่างต่างกันได้หมด — ไฟเบอร์ วิทยุ ดาวเทียม สายทองแดง — และแอปพลิเคชันทุกตัวที่อยู่ข้างบนก็ต่างกันได้หมดเช่นกัน ทั้งสองฝั่งมาเจอกันที่ชั้นเดียวที่จนใจโดยตั้งใจ หลักการที่อยู่เบื้องหลังความจนใจนั้นถูกตั้งชื่อโดย Jerome Saltzer, David Reed และ David Clark ว่า ฟังก์ชันหนึ่งจะถูก implement ได้อย่างสมบูรณ์และถูกต้องก็ต่อเมื่อมีความรู้และความช่วยเหลือจากแอปพลิเคชันที่ยืนอยู่ที่จุดปลายทางทั้งสองฝั่งเท่านั้น การทำให้ฟังก์ชันที่ถูกตั้งคำถามนั้นเป็นฟีเจอร์ของระบบสื่อสารเองจึงเป็นไปไม่ได้ (end-to-end arguments, 1981/1984) วางความฉลาดไว้ที่ขอบ เครือข่ายที่พยายามรับประกันการส่งถึงก็ยังไม่ช่วยแอปพลิเคชันให้พ้นจากการต้องตรวจสอบเองอยู่ดี แถมยังเก็บภาษีจากทุกแอปพลิเคชันที่ไม่สนใจเรื่องนี้เลยด้วยซ้ำ

So reliability is built *on top of* unreliability, by the endpoints, and that is TCP. Vint Cerf and Robert Kahn's 1974 design (*IEEE Trans. Comm.* COM-22(5): 637–648) numbers every byte, acknowledges what arrives, and retransmits what does not, manufacturing an ordered, gap-free stream out of a medium

that promises none of those things. The ARPANET switched over to it wholesale on 1 January 1983, a date fixed two years in advance in RFC 801 — the last time the network was small enough to be changed on a single day.

> ฉะนั้นความน่าเชื่อถือจึงถูกสร้าง ซ้อนทับ ความไม่น่าเชื่อถือ โดยจุดปลายทางทั้งสองฝั่งเป็นคนทำ และนั่นคือ TCP การออกแบบของ Vint Cerf กับ Robert Kahn ในปี 1974 (IEEE Trans. Comm. COM-22(5): 637–648) ใส่หมายเลขให้ทุกไบต์ ยืนยันสิ่งที่มาถึง แล้วส่งซ้ำสิ่งที่ไม่มาถึง สร้างสตรีมที่เรียงลำดับและไม่มีช่องว่างขึ้นมาจากตัวกลางที่ไม่ได้สัญญาอะไรแบบนั้นไว้เลยสักอย่าง ARPANET เปลี่ยนมาใช้มันทั้งระบบในวันที่ 1 มกราคม 1983 วันที่ถูกกำหนดไว้ล่วงหน้าสองปีใน RFC 801 — ครั้งสุดท้ายที่เครือข่ายยังเล็กพอจะเปลี่ยนได้ภายในวันเดียว

Which returns us to 1986, because retransmission is exactly the behavior that ate the network. A sender that responds to congestion by trying harder makes congestion worse; ten thousand senders doing so collapse the commons. There is no central regulator to prevent it — no one owns the internet, and no one can be told to slow down. Jacobson's fix, which is still in the bones of every connection you make, was to have each sender infer the state of the whole from what it can see locally: treat a lost packet as evidence of congestion, cut your sending rate sharply, then probe upward again gently. Congestion control is voluntary restraint by parties who cannot see each other, and it holds because the alternative is that nobody gets anything. It is the most consequential piece of distributed politeness ever deployed.

> ซึ่งพาเรากลับไปที่ปี 1986 เพราะการส่งซ้ำ (retransmission) คือพฤติกรรมเดียวกับที่กินเครือข่ายไปเองนั่นแหละ ผู้ส่งที่ตอบสนองต่อความแออัด (congestion) ด้วยการพยายามส่งให้หนักขึ้น จะยิ่งทำให้ความแออัดแย่ลง ผู้ส่งหนึ่งหมื่นรายทำแบบนั้นพร้อมกันก็พังทรัพยากรส่วนรวมลงได้ ไม่มีหน่วยงานกลางไหนคอยห้ามได้ — ไม่มีใครเป็นเจ้าของอินเทอร์เน็ต และไม่มีใครถูกสั่งให้ช้าลงได้ ทางแก้ของ Jacobson ซึ่งยังฝังอยู่ในกระดูกของทุกการเชื่อมต่อที่คุณสร้างขึ้นทุกวันนี้ คือให้ผู้ส่งแต่ละรายอนุมานสถานะ (state) ของทั้งระบบจากสิ่งที่ตัวเองมองเห็นในพื้นที่: มองแพ็กเก็ตที่หายไปเป็นหลักฐานของความแออัด ตัดอัตราการส่งลงอย่างแรง แล้วค่อยๆ เพิ่มขึ้น

> ใหม่อย่างนุ่มนวล congestion control คือการยับยั้งชั่งใจโดยสมัครใจของฝ่ายต่างๆ ที่มองไม่เห็นกันและกัน และมันคงอยู่ได้ก็เพราะทางเลือกอื่นคือไม่มีใครได้อะไรเลย มันคือชิ้นส่วนของความสุภาพแบบกระจายศูนย์ที่ส่งผลสำคัญที่สุดเท่าที่เคยถูกนำไปใช้จริง

The same structure — no authority, only agreement — governs how things are found. The Domain Name System, specified by Paul Mockapetris in 1983 and refined in RFCs 1034 and 1035 (1987), turns names into addresses by delegation: each zone is trusted to answer for the names beneath it. Routing between networks runs on BGP (RFC 4271), in which each network announces which addresses it can reach and its neighbors believe it. On 24 February 2008, Pakistan Telecom, complying with an order to block YouTube domestically, announced a route to a slice of YouTube's addresses more specific than YouTube's own — and because more specific wins, and because its upstream provider passed the claim on, the world's traffic to YouTube went to Pakistan for roughly two hours (RIPE NCC). Nothing was hacked. The system worked as designed, and the design assumes honesty.

> โครงสร้างเดียวกันนี้ — ไม่มีอำนาจ มีแต่ข้อตกลง — เป็นตัวคุมว่าจะหาของเจอได้อย่างไร Domain Name System ที่ Paul Mockapetris เขียนสเปกไว้ในปี 1983 แล้วปรับปรุงใน RFCs 1034 และ 1035 (1987) แปลงชื่อให้เป็นที่อยู่ด้วยการมอบอำนาจ (delegation): แต่ละโซนได้รับความไว้ใจให้ตอบแทนชื่อที่อยู่ใต้มัน การจัดเส้นทาง (routing) ระหว่างเครือข่ายวิ่งอยู่บน BGP (RFC 4271) ซึ่งแต่ละเครือข่ายประกาศว่าตัวเองไปถึงที่อยู่ไหนได้บ้าง แล้วเพื่อนบ้านก็เชื่อคำประกาศนั้น เมื่อวันที่ 24 กุมภาพันธ์ 2008 Pakistan Telecom ทำตามคำสั่งให้บล็อก YouTube ภายในประเทศ ด้วยการประกาศเส้นทางไปยังที่อยู่ส่วนหนึ่งของ YouTube ที่จำเพาะเจาะจงกว่าของ YouTube เองเสียอีก — และเพราะเส้นทางที่จำเพาะกว่าชนะเสมอ แถมผู้ให้บริการต้นทางของมันก็ส่งต่อคำประกาศนั้นออกไป ทราฟฟิกทั้งโลกที่มุ่งหา YouTube จึงวิ่งไปที่ปากีสถานอยู่ราวสองชั่วโมง (RIPE NCC) ไม่มีอะไรถูกแฮ็กเลย ระบบทำงานตามที่ถูกออกแบบมาทุกอย่าง และการออกแบบนั้นตั้งอยู่บนสมมติฐานว่าทุกฝ่ายซื่อสัตย์

On top of all this sits one application among many, proposed at CERN in March 1989 as a way to keep track of documents (Berners-Lee) and released to the public domain on 30 April 1993. For most people the Web *is* the network — which is what a well-built floor looks like from above.

> อยู่บนสุดของทั้งหมดนี้คือแอปพลิเคชันหนึ่งในหลายๆ ตัว ที่ถูกเสนอขึ้นที่ CERN ในเดือนมีนาคม 1989 เพื่อใช้ติดตามเอกสาร (Berners-Lee) แล้วถูกปล่อยสู่สาธารณสมบัติในวันที่ 30 เมษายน 1993 สำหรับคนส่วนใหญ่ เว็บ (the Web) คือ เครือข่าย (network) เอง — ซึ่งนั่นแหละคือหน้าตาของชั้นที่ถูกสร้างมาอย่างดีเมื่อมองจากด้านบน

Connections · เชื่อมโยง

Below: chapter 03 sets the physical ceiling on any channel and supplies the error-detecting codes every layer uses; chapters 04 and 05 provide the machine and the operating system that actually move the bytes. Above: chapter 10 takes the machines connected here and asks what it means for them to agree on anything, a far harder question than making them talk; chapter 08 is often on the far side of one of these connections; chapter 11 is the one this floor keeps needing, since DNS and BGP were built for a network of colleagues.

> ด้านล่าง: บทที่ 03 กำหนดเพดานทางกายภาพของช่องสัญญาณ (channel) ใดๆ และให้รหัสตรวจจับข้อผิดพลาด (error-detecting code) ที่ทุกชั้นใช้ บทที่ 04 กับ 05 ให้เครื่องและระบบปฏิบัติการที่ลงมือขยับไบต์จริงๆ ด้านบน: บทที่ 10 เอาเครื่องที่เชื่อมต่อกันในบทนี้มาถามว่าการที่พวกมันจะตกลงอะไรกันสักอย่างหนึ่งแปลว่าอะไร ซึ่งเป็นคำถามที่ยากกว่าการทำให้พวกมันคุยกันได้มาก บทที่ 08 มักอยู่อีกฟากหนึ่งของการเชื่อมต่อแบบนี้เสมอ บทที่ 11 คือบทที่ชั้นนี้ต้องพึ่งอยู่ตลอด เพราะ DNS กับ BGP ถูกสร้างมาสำหรับเครือข่ายของคนที่รู้จักกันเป็นเพื่อนร่วมงาน

Open questions · คำถามที่ยังเปิดอยู่

The end-to-end argument is under steady pressure: firewalls, address translators and content delivery networks all put intelligence back in the middle, and QUIC moved transport out of the kernel to escape the middle's

grip. Is the narrow waist still the architecture, or a memory of one? Second: BGP's trust problem has had a technical answer for years — cryptographic validation of route origins — and deployment remains partial, which makes it less a research question than a study in why coordination fails when no one can be compelled. Third: congestion control now runs many algorithms at once, some far more aggressive than others, on a commons that assumed roughly equal restraint. What holds an agreement together when the parties no longer agree?

> end-to-end argument กำลังถูกกดดันอย่างต่อเนื่อง: firewall, address translator และ content delivery network ต่างพาความฉลาดกลับเข้าไปอยู่ตรงกลางกันหมด ส่วน QUIC ก็ย้าย transport ออกจาก kernel เพื่อหนีเงื้อมมือของตรงกลางนั้น เอวคอดแคบ (narrow waist) ยังเป็นสถาปัตยกรรมจริงอยู่ หรือเป็นแค่ความทรงจำถึงมันไปแล้วกันแน่ ข้อสอง: ปัญหาความไว้ใจของ BGP มีคำตอบทางเทคนิคมาหลายปีแล้ว — การตรวจสอบแหล่งที่มาของเส้นทางด้วยวิทยาการรหัสลับ (cryptographic validation) — แต่การใช้งานจริงยังทำได้แค่บางส่วน ซึ่งทำให้มันไม่ใช่คำถามวิจัยเท่ากับเป็นกรณีศึกษาว่าทำไมการประสานงานถึงล้มเหลว เมื่อไม่มีใครถูกบังคับได้ ข้อสาม: congestion control ตอนนี้รันหลายอัลกอริทึมพร้อมกัน บางตัวก็ก้าวร้าวกว่าตัวอื่นมาก บนทรัพยากรส่วนรวมที่ออกแบบมาโดยสมมติว่าทุกฝ่ายยับยั้งชั่งใจเท่าๆ กัน อะไรจะยึดข้อตกลงเอาไว้ได้ เมื่อฝ่ายต่างๆ ไม่เห็นตรงกันอีกต่อไป

Go deeper · อ่านต่อ

- J. H. Saltzer, D. P. Reed & D. D. Clark, "End-to-End Arguments in System Design," *ACM TOCS* 2, no. 4 (1984): 277–288 — the shortest route to understanding why the Internet is shaped as it is; MIT PDF.

  > J. H. Saltzer, D. P. Reed & D. D. Clark, "End-to-End Arguments in System Design," ACM TOCS 2, no. 4 (1984): 277–288 — เส้นทางสั้นที่สุดที่จะเข้าใจว่าทำไมอินเทอร์เน็ตถึงมีรูปทรงแบบที่เป็นอยู่; MIT PDF.

- Van Jacobson & Michael J. Karels, "Congestion Avoidance and Control," SIGCOMM '88 — PDF.

> Van Jacobson & Michael J. Karels, "Congestion Avoidance and Control," SIGCOMM '88 — PDF.

- Larry Peterson & Bruce Davie, *Computer Networks: A Systems Approach* — the systems-first textbook, kept current and free at systemsapproach.org. For the classic course treatment, Kurose & Ross, *Computer Networking: A Top-Down Approach*, 9th ed., 2025.

> Larry Peterson & Bruce Davie, Computer Networks: A Systems Approach — ตำราที่เริ่มจากมุมมองระบบ ปรับปรุงต่อเนื่องและอ่านฟรีที่ systemsapproach.org. ถ้าต้องการแนวทางแบบคอร์สคลาสสิก ดู Kurose & Ross, Computer Networking: A Top-Down Approach, 9th ed., 2025.

CHAPTER 10

Distributed Systems

ระบบกระจาย

This chapter is the floor where "the computer" stops being a noun — the machine is now many machines, and the abstraction of one is something you have to build.

บทนี้คือชั้นที่ "คอมพิวเตอร์" เลิกเป็นคำนามเดี่ยว — เครื่องกลายเป็นเครื่องหลายเครื่อง และนามธรรม (abstraction) ของการเป็น "หนึ่งเดียว" คือสิ่งที่ต้องสร้างขึ้นเอง

What breaks when there is no "now"?

อะไรพังเมื่อไม่มี "ตอนนี้" อยู่จริง

At 22:52 UTC on 21 October 2018, engineers replacing failing optical equipment briefly severed the link between GitHub's East Coast network hub and its primary East Coast data center. Connectivity was restored in forty-three seconds. In those seconds the system that keeps the databases healthy did exactly what it was built to do: unable to reach the East Coast primaries, it promoted replicas on the West Coast. Writes had by then been accepted on both coasts. Neither side was wrong, neither could be discarded, and reconciling them took twenty-four hours and eleven minutes (GitHub post-incident analysis).

> เวลา 22:52 UTC วันที่ 21 ตุลาคม 2018 วิศวกรที่กำลังเปลี่ยนอุปกรณ์ใยแก้วนำแสงที่เสียอยู่ ตัดการเชื่อมต่อระหว่างศูนย์เครือข่ายฝั่งตะวันออกของ GitHub กับดาต้าเซ็นเตอร์หลักฝั่งตะวันออกไปชั่วขณะ การเชื่อมต่อกลับมาใน 43 วินาที แต่ในช่วงเวลานั้น ระบบที่คอยดูแลให้ฐานข้อมูลแข็งแรงก็ทำสิ่งที่มันถูกสร้างมาให้ทำพอดี — เมื่อติดต่อ primary ฝั่งตะวันออกไม่ได้ มันก็เลื่อน replica ฝั่งตะวันตกขึ้นเป็น primary แทน ตอนนั้นมีการเขียนข้อมูลถูกรับเข้าไปแล้วทั้งสองฝั่ง ไม่มีฝั่งไหนผิด ไม่มีฝั่งไหนทิ้งได้ และการประสานให้ตรงกันใหม่ใช้เวลา 24 ชั่วโมง 11 นาที (GitHub post-incident analysis)

Nothing failed. That is the shape of every hard problem on this floor. The link came back, the machines were healthy, the failover logic was correct — and the system still could not say what had happened, because for forty-three seconds there was no shared answer to the question *what is true right now*.

> ไม่มีอะไรพังเลยสักอย่าง — และนั่นแหละคือรูปร่างของปัญหายากทุกข้อบนชั้นนี้ ลิงก์กลับมาแล้ว เครื่องทุกเครื่องยังแข็งแรงดี ตรรกะ failover ก็ทำงานถูกต้อง แต่ระบบกลับบอกไม่ได้ว่าเกิดอะไรขึ้น เพราะตลอด 43 วินาทีนั้นไม่มีคำตอบร่วมกันสักคำตอบเดียวสำหรับคำถามว่า ตอนนี้อะไรคือความจริง

Start with why that question has no easy answer. Two machines have two clocks, clocks drift, and no amount of care makes a timestamp from one comparable with a timestamp from the other. Leslie Lamport's response, in 1978, was to stop asking for time and ask only for *order*. His "happened before" relation says one event precedes another if they occur in sequence on the same machine, or if the first is the sending of a message and the second its receipt — plus whatever follows by chaining those together. Everything else is *concurrent*: genuinely unordered, so that any consistent story about which came first is as good as any other (Lamport, CACM 21(7), 1978). Physics reached this first: simultaneity is not absolute, and the only real ordering is the one causation can carry. In a distributed system, messages are the light cones.

> เริ่มจากว่าทำไมคำถามนี้ถึงไม่มีคำตอบง่ายๆ เครื่องสองเครื่องมีนาฬิกาสองเรือน นาฬิกาเหวี่ยงไม่ตรงกัน และต่อให้ระวังแค่ไหนก็ไม่มีทางทำให้ timestamp ของเครื่องหนึ่งเทียบกับอีกเครื่องได้ตรงๆ Leslie Lamport ตอบโจทย์นี้ในปี 1978 ด้วยการเลิกถามหา "เวลา" แล้วถามหาแค่ ลำดับ แทน ความสัมพันธ์ "happened-before" ของเขาบอกว่าเหตุการณ์หนึ่งเกิดก่อนอีกเหตุการณ์หนึ่งได้ก็ต่อเมื่อทั้งคู่เกิดตามลำดับบนเครื่องเดียวกัน หรือเหตุการณ์แรกคือการส่งข้อความและเหตุการณ์หลังคือการรับข้อความนั้น — บวกกับอะไรก็ตามที่ไล่ต่อกันเป็นสายจากสองแบบนี้ ที่เหลือทั้งหมดคือ concurrent: ไม่มีลำดับจริงๆ ดังนั้นเรื่องเล่าที่สอดคล้องกันเองแบบ

> ไหนก็ตามว่าอะไรเกิดก่อนย่อมดีเท่ากันหมด (Lamport, CACM 21(7), 1978) ฟิสิกส์เจอเรื่องนี้มาก่อนแล้ว — ความพร้อมกันไม่ใช่สิ่งสัมบูรณ์ ลำดับที่แท้จริงมีแค่ลำดับที่ความเป็นเหตุเป็นผลพาไปได้เท่านั้น ในระบบกระจาย ข้อความคือ light cone

The second difficulty is worse, because no formalism softens it. On one machine a component either works or has crashed. Across machines there is a third state indistinguishable from the other two: *no answer yet*. A silent peer may be dead, overloaded, or healthy behind a broken link, and no timeout separates these: a timeout says the answer has not arrived, never that it will not. Lamport's much-quoted definition captures what this does to the experience of building anything: *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"* (Lamport, 28 May 1987). Failure here is not an event. It is an ambiguity.

> ปัญหาที่สองหนักกว่านั้น เพราะไม่มีรูปแบบทางคณิตศาสตร์ไหนช่วยผ่อนมันได้ บนเครื่องเดียว คอมโพเนนต์หนึ่งจะทำงานอยู่หรือล่มไปแล้วเท่านั้น แต่ข้ามเครื่องกันไปมีสถานะที่สามที่แยกไม่ออกจากสองแบบแรก คือ ยังไม่มีคำตอบ เพื่อนร่วมเครือข่ายที่เงียบไปอาจตายแล้ว อาจโหลดเกิน หรืออาจแข็งแรงดีแต่อยู่หลังลิงก์ที่ขาด และไม่มี timeout ไหนแยกสามแบบนี้ออกจากกันได้ — timeout บอกได้แค่ว่าคำตอบยังไม่มาถึง ไม่เคยบอกว่ามันจะไม่มาถึงเลย นิยามที่ถูกอ้างถึงบ่อยที่สุดของ Lamport จับภาพว่าเรื่องนี้ทำอะไรกับประสบการณ์การสร้างของทุกอย่าง — "ระบบกระจายคือระบบที่ความล้มเหลวของคอมพิวเตอร์เครื่องหนึ่งซึ่งคุณไม่รู้ด้วยซ้ำว่ามีอยู่ สามารถทำให้คอมพิวเตอร์ของคุณเองใช้งานไม่ได้" (A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable) (Lamport, 28 May 1987) ความล้มเหลวตรงนี้ไม่ใช่เหตุการณ์ มันคือความกำกวม

Given ambiguity, can machines at least agree on one value — which replica is primary, which transaction committed, who holds the lock? This is the **consensus** problem, and it is the hardest thing here. Fischer, Lynch and Paterson proved in 1985 that in an asynchronous system, where messages can be delayed arbitrarily, no deterministic protocol can guarantee that all

correct processes decide if even one process may crash (JACM 32(2): 374–382). The proof is not about bad engineering. It is the impossibility of telling a dead process from a slow one, elevated into a theorem.

> เมื่อมีความกำกวมแบบนี้ เครื่องต่างๆ ยังพอจะตกลงกันได้เรื่องค่าเดียวไหม — replica ไหนคือ primary, transaction ไหน commit ไปแล้ว, ใครถือ lock อยู่? นี่คือปัญหา **ฉันทามติ (consensus)** และมันคือเรื่องยากที่สุดบนชั้นนี้ Fischer, Lynch และ Paterson พิสูจน์ไว้ในปี 1985 ว่าในระบบอะซิงโครนัส (asynchronous) ที่ข้อความอาจถูกหน่วงได้นานเท่าไรก็ได้ ไม่มีโพรโทคอลเชิงกำหนดแน่นอน (deterministic) ตัวไหนรับประกันได้ว่าโพรเซสที่ถูกต้องทุกตัวจะตัดสินใจได้ลงตัว ถ้ามีแม้แต่โพรเซสเดียวที่อาจล่ม (JACM 32(2): 374–382) บทพิสูจน์นี้ไม่ได้ว่าด้วยวิศวกรรมที่แย่ มันคือความเป็นไปไม่ได้ในการแยกโพรเซสที่ตายแล้วออกจากโพรเซสที่แค่ช้า ที่ถูกยกขึ้นเป็นทฤษฎีบท

Working systems live inside that result. Paxos, and Raft after it, never sacrifice *safety* — they will not decide two different values. What they give up is the promise to decide *promptly*: during a bad enough network they make no progress, and when it recovers, they do. Lamport submitted Paxos in 1990 as an archaeological account of a fictional Greek parliament, and it went unpublished for eight years while reviewers puzzled over the joke (ACM TOCS 16(2), 1998); he later wrote a four-page version, "Paxos Made Simple". Diego Ongaro and John Ousterhout's Raft made understandability an explicit design goal and won a best-paper award for it (USENIX ATC '14). GitHub's failover ran on Raft. Raft did not malfunction; it agreed, correctly and quickly, on the wrong thing.

> ระบบที่ใช้งานจริงอยู่ภายในผลลัพธ์นั้น Paxos และ Raft ที่ตามมาทีหลัง ไม่ยอมเสียความปลอดภัย เด็ดขาด — มันจะไม่ตัดสินใจไปที่ค่าสองค่าที่ต่างกัน สิ่งที่มันยอมเสียคือคำมั่นว่าจะตัดสินใจได้ ทันท่วงที: ระหว่างที่เครือข่ายแย่มากพอ มันจะไม่คืบหน้าเลย แล้วพอเครือข่ายกลับมาดี มันก็เดินหน้าต่อ Lamport ส่ง Paxos เข้าตีพิมพ์ในปี 1990 ในรูปเรื่องเล่าเชิงโบราณคดีเกี่ยวกับรัฐสภากรีกที่แต่งขึ้น และมันไม่ได้ตีพิมพ์อยู่แปดปีเพราะผู้ตรวจทานงงกับมุกตลกนั้น (ACM TOCS 16(2), 1998) ภายหลังเขาเขียนฉบับสี่หน้าชื่อ "Paxos Made Simple" Raft ของ Diego Ongaro และ John

> Ousterhout ตั้งเป้าการออกแบบตรงๆ ว่าต้องเข้าใจง่าย และได้รางวัล best paper จากมัน (USENIX ATC '14) ระบบ failover ของ GitHub วิ่งอยู่บน Raft — Raft ไม่ได้ทำงานผิดพลาด มันตกลงกันได้ ถูกต้องและรวดเร็ว บนเรื่องที่ผิด

Because consensus costs round trips, systems trade it away. Eric Brewer proposed in a July 2000 keynote that consistency, availability and partition tolerance cannot all be had (PODC keynote); Seth Gilbert and Nancy Lynch made the conjecture a theorem in 2002 (*SIGACT News* 33(2)). The popular reading — *pick two of three* — is wrong, and Brewer said so twelve years later: partitions are a fact of networks, not a design choice, so the only real decision is what to do *during* one, and it can be made differently for different operations (*IEEE Computer* 45(2), 2012). Refuse writes on both sides and stay consistent; accept them and stay available. GitHub's day of repair is the bill for the second option.

> เพราะการหาฉันทามติมีต้นทุนเป็น round trip ระบบต่างๆ จึงมักแลกมันทิ้งไป Eric Brewer เสนอไว้ในคีย์โน้ตเดือนกรกฎาคม 2000 ว่าความสอดคล้อง (consistency) ความพร้อมใช้งาน (availability) และการทนต่อการแยกขาดของเครือข่าย (partition tolerance) ไม่มีทางได้ครบทั้งสามอย่างพร้อมกัน (PODC keynote) Seth Gilbert และ Nancy Lynch ทำให้ข้อสันนิษฐานนี้กลายเป็นทฤษฎีบทในปี 2002 (SIGACT News 33(2)) การอ่านแบบที่คนทั่วไปเข้าใจ — เลือกได้สองในสาม — นั้นผิด และ Brewer เองก็พูดแบบนั้นสิบสองปีให้หลัง: การแยกขาดของเครือข่ายเป็นข้อเท็จจริงของเครือข่าย ไม่ใช่ทางเลือกในการออกแบบ ดังนั้นการตัดสินใจจริงมีอย่างเดียวคือจะทำอะไร ระหว่างที่ มันเกิดขึ้น และคำตอบนั้นเลือกต่างกันได้สำหรับแต่ละปฏิบัติการ (IEEE Computer 45(2), 2012) ปฏิเสธการเขียนทั้งสองฝั่งแล้วรักษาความสอดคล้องไว้ หรือรับการเขียนไว้แล้วรักษาความพร้อมใช้งานไว้ วันซ่อมของ GitHub คือบิลที่ต้องจ่ายสำหรับทางเลือกที่สอง

So consistency became a menu rather than a property. At the strict end is linearizability — every operation appears to take effect at a single instant between its start and its finish, which is to say the system successfully pretends to be one machine (Herlihy & Wing, *ACM TOPLAS* 12(3), 1990). At the loose end is eventual consistency: replicas may disagree now and will

converge if you stop writing (Vogels, CACM 52(1), 2009). Between them lies a long spectrum, and choosing a point on it is the central architectural decision of any system spanning a building, let alone a planet.

> ความสอดคล้องจึงกลายเป็นเมนูให้เลือก มากกว่าจะเป็นคุณสมบัติตายตัว ปลายด้านเข้มงวดที่สุดคือ linearizability — ทุกปฏิบัติการดูเหมือนเกิดขึ้นในจุดเดียวระหว่างจุดเริ่มกับจุดจบของมัน กล่าวคือระบบแกล้งเป็นเครื่องเดียวได้สำเร็จ (Herlihy & Wing, ACM TOPLAS 12(3), 1990) ปลายด้านหลวมที่สุดคือ eventual consistency: replica อาจไม่ตรงกันตอนนี้ แต่จะลู่เข้าหากันถ้าหยุดเขียนข้อมูล (Vogels, CACM 52(1), 2009) ระหว่างสองปลายนี้คือสเปกตรัมยาวเหยียด และการเลือกจุดหนึ่งบนสเปกตรัมนั้นคือการตัดสินใจเชิงสถาปัตยกรรมที่สำคัญที่สุดของระบบไหนก็ตามที่กว้างเกินตึกเดียว ยิ่งไม่ต้องพูดถึงกว้างเท่าดาวเคราะห์

One more move is available: buy "now" back with hardware. Google's Spanner equips its data centers with GPS receivers and atomic clocks, and its TrueTime interface reports not a timestamp but an *interval* — the uncertainty ε "varies from about 1 to 7 ms over each poll interval," roughly 4 ms most of the time (OSDI 2012). A transaction that must be globally ordered waits out the uncertainty before committing. It is the most literal illustration of the tower: a floor that could not assume a global clock bought one, in satellites, and sold the result upward as an ordinary database.

> ยังมีอีกท่าหนึ่งที่ทำได้ คือซื้อ "ตอนนี้" กลับมาด้วยฮาร์ดแวร์ Spanner ของ Google ติดตั้งเครื่องรับ GPS และนาฬิกาอะตอมไว้ในดาต้าเซ็นเตอร์ของตัวเอง และอินเทอร์เฟซ TrueTime ของมันรายงานไม่ใช่ timestamp แต่เป็น ช่วง (interval) — ความไม่แน่นอน ε "แกว่งอยู่ราว 1 ถึง 7 มิลลิวินาทีต่อรอบโพล" (varies from about 1 to 7 ms over each poll interval) คือราว 4 มิลลิวินาทีในเวลาส่วนใหญ่ (OSDI 2012) transaction ที่ต้องเรียงลำดับกันทั่วโลกจะรอให้ความไม่แน่นอนนี้ผ่านไปก่อนค่อย commit นี่คือภาพประกอบที่ตรงตัวที่สุดของหอคอย — ชั้นที่ตั้งสมมติฐานเรื่องนาฬิกากลางไม่ได้ ก็ซื้อมันมาเองด้วยดาวเทียม แล้วขายผลลัพธ์ขึ้นไปชั้นบนในรูปฐานข้อมูลธรรมดาๆ

The oldest warning here is still the most useful. Around 1994 Peter Deutsch listed the false assumptions newcomers make — 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 — and James Gosling later added the eighth, that the network is homogeneous (Gosling). Each is a place where the abstraction of "one machine" leaks, and every large outage of the last thirty years is a receipt for one of them.

> คำเตือนที่เก่าแก่ที่สุดในเรื่องนี้ก็ยังใช้ได้ผลที่สุดอยู่ดี ราวปี 1994 Peter Deutsch ไล่รายการความเชื่อผิดๆ ที่คนใหม่มักเชื่อ — เครือข่ายเชื่อถือได้ ความหน่วงเป็นศูนย์ แบนด์วิดท์ไม่จำกัด เครือข่ายปลอดภัย โทโพโลยีไม่เปลี่ยน มีผู้ดูแลระบบคนเดียว ต้นทุนการส่งข้อมูลเป็นศูนย์ — แล้ว James Gosling ก็เติมข้อที่แปดทีหลังว่า เครือข่ายเป็นเนื้อเดียวกันหมด (Gosling) แต่ละข้อคือจุดที่ abstraction ของ "เครื่องเดียว" รั่ว และเหตุขัดข้องครั้งใหญ่ทุกครั้งในสามสิบปีที่ผ่านมาคือใบเสร็จของข้อใดข้อหนึ่งในนี้

Connections · เชื่อมโยง

Below: chapter 09 supplies the messages and, crucially, their unreliability — this floor exists because that one cannot promise delivery. Chapter 05 owns concurrency within one machine, where locks work and failure is unambiguous; the gap between them is exactly what shared memory was buying. Chapter 08 supplies the guarantees traded away here. Above: chapter 07 pays the operational cost of every decision on this floor; chapter 11 replaces "machines may crash" with "machines may lie," a strictly harder problem; chapter 12 trains its models on clusters that are distributed systems in every respect above.

> ชั้นล่าง: บทที่ 09 ส่งข้อความมาให้ พร้อมกับความไม่น่าเชื่อถือของมันที่สำคัญกว่านั้น — ชั้นนี้มีอยู่ได้เพราะชั้นนั้นรับประกันการส่งถึงไม่ได้ บทที่ 05 ดูแลการทำงานพร้อมกันภายในเครื่องเดียว ที่ที่ lock ทำงานได้จริงและความล้มเหลวไม่กำกวม — ช่องว่างระหว่างสองบทนี้แหละคือสิ่งที่หน่วยความจำร่วม (shared memory) เคยซื้อไว้ให้ บทที่ 08 ส่งการรับประกัน (guarantee) ที่ถูกแลกทิ้งไปตรงนี้มาให้ ชั้นบน: บทที่ 07 จ่ายต้นทุนด้านปฏิบัติการของทุกการตัดสินใจบนชั้นนี้ บทที่ 11 เปลี่ยน "เครื่อง

> อาจล่ม" ให้กลายเป็น "เครื่องอาจโกหก" ซึ่งเป็นปัญหาที่ยากกว่าเดิมอย่างเคร่งครัด บทที่ 12 เทรนโมเดลของมันบนคลัสเตอร์ที่เป็นระบบกระจายในทุกแง่มุมข้างต้นนี้เป๊ะ

Open questions · คำถามที่ยังเปิดอยู่

Is consensus getting cheaper? Tight clock synchronization and fast data-center networks have narrowed the gap between the theory's worst case and the common one, and how much of the CAP trade is fundamental rather than an artifact of 2000-era networks is unsettled. Second: consistency models are precise but not intuitive, and most production bugs here come from engineers reasoning about a model they could define but not feel. Third, and largest: partial failure is the one thing no abstraction has ever fully hidden — every attempt to make a remote call look local has leaked. A flaw awaiting a fix, or the honest shape of the world showing through?

> ฉันทามติกำลังถูกกว่าเดิมไหม? การซิงก์นาฬิกาที่แม่นขึ้นและเครือข่ายดาต้าเซ็นเตอร์ที่เร็วขึ้นได้ทำให้ช่องว่างระหว่างกรณีแย่สุดตามทฤษฎีกับกรณีที่พบจริงแคบลง แต่ยังไม่มีข้อสรุปว่าการแลกแบบ CAP นั้นเป็นเรื่องพื้นฐานจริงแค่ไหน หรือเป็นแค่ร่องรอยของเครือข่ายยุค 2000 ข้อที่สอง: แบบจำลองความสอดคล้อง (consistency model) แม่นยำแต่ไม่ได้สัมผัสได้ตามสัญชาตญาณ และบั๊กส่วนใหญ่ในงานจริงตรงนี้มาจากวิศวกรที่ให้เหตุผลกับแบบจำลองที่นิยามได้แต่รู้สึกไม่ได้ ข้อที่สาม และใหญ่ที่สุด: ความล้มเหลวบางส่วน (partial failure) คือสิ่งเดียวที่ไม่มี abstraction ไหนเคยซ่อนได้มิดจริงๆ — ทุกความพยายามทำให้การเรียกข้ามเครื่องดูเหมือนเรียกในเครื่องเดียวกันล้วนรั่วมาแล้วทั้งนั้น นี่คือข้อบกพร่องที่รอการแก้ หรือคือรูปร่างที่แท้จริงของโลกที่โผล่ออกมาให้เห็นกันแน่?

Go deeper · อ่านต่อ

- Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM* 21, no. 7 (1978): 558–565 — eight pages under everything above; award citation and links.

> Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM 21, no. 7 (1978): 558–565 — แปดหน้าที่รองรับทุกอย่างข้างต้นอยู่ข้างใต้; award citation and links

- Fischer, Lynch & Paterson, "Impossibility of Distributed Consensus with One Faulty Process," JACM 32, no. 2 (1985): 374–382 — the boundary of the possible; award citation.

> Fischer, Lynch & Paterson, "Impossibility of Distributed Consensus with One Faulty Process," JACM 32, no. 2 (1985): 374–382 — เส้นแบ่งของสิ่งที่เป็นไปได้; award citation

- Maarten van Steen & Andrew S. Tanenbaum, *Distributed Systems*, 4th ed., 2023 — free PDF at distributed-systems.net. For the practitioner's view, Kleppmann & Riccomini, *Designing Data-Intensive Applications*, 2nd ed., O'Reilly, 2026.

> Maarten van Steen & Andrew S. Tanenbaum, Distributed Systems, 4th ed., 2023 — PDF ฟรีที่ distributed-systems.net ถ้าอยากได้มุมมองแบบคนทำงานจริง อ่าน Kleppmann & Riccomini, Designing Data-Intensive Applications, 2nd ed., O'Reilly, 2026

CHAPTER 11

Security & Cryptography

ความปลอดภัยและวิทยาการรหัสลับ

This chapter is not a floor of the tower but a stairwell cut through all of them — an attacker enters on whichever floor you left unguarded, and is under no obligation to use the door you built.

บทนี้ไม่ใช่ชั้นหนึ่งของหอคอย แต่คือบันไดที่เจาะทะลุผ่านทุกชั้น — ผู้โจมตีเข้ามาทางชั้นไหนก็ได้ที่คุณเผลอปล่อยให้ไม่มีคนเฝ้า และไม่มีข้อผูกมัดใดๆ ที่จะต้องใช้ประตูที่คุณสร้างไว้

How do strangers build trust over a wire?

คนแปลกหน้าสร้างความไว้ใจกันผ่านสายได้อย่างไร

In November 1976 the *IEEE Transactions on Information Theory* carried a paper opening with a sentence academics almost never allow themselves: "We stand today on the brink of a revolution in cryptography" (Diffie & Hellman, 1976). The problem it dissolved was thousands of years old, and it was not that codes kept getting broken. It was a delivery problem. Every cipher in history — Caesar's, Mary Stuart's, Enigma's — required that sender and receiver already share a secret key: a courier, a meeting, a locked bag. Scrambling the message was never the hard part. Getting the key to the far end untouched was, and on a planetary network of strangers who will never meet, it looked impossible.

> เดือนพฤศจิกายน 1976 วารสาร IEEE Transactions on Information Theory ตีพิมพ์เปเปอร์ที่เปิดเรื่องด้วยประโยคที่นักวิชาการแทบไม่มีใครกล้าเขียน — "วันนี้เรากำลังยืนอยู่บนขอบของการปฏิวัติทางวิทยาการรหัสลับ" (We stand today on the brink of a revolution in cryptography) (Diffie & Hellman, 1976) ปัญหาที่เปเปอร์นี้คลี่คลายมีอายุนับพันปี และไม่ใช่ว่ารหัสถูกถอดได้เรื่อยๆ แต่เป็นปัญหาการส่งมอบต่างหาก รหัสลับทุกแบบในประวัติศาสตร์ — ของ Caesar, ของ Mary Stuart,

> ของ Enigma — ต้องการให้ผู้ส่งกับผู้รับมีกุญแจลับ (key) ร่วมกันอยู่ก่อนแล้ว ไม่ว่าจะผ่านคนเดินสาร การนัดพบ หรือกระเป๋าที่ล็อกไว้ การสับเปลี่ยนข้อความให้อ่านไม่ออกไม่เคยเป็นส่วนที่ยาก ส่วนที่ยากคือการส่งกุญแจไปถึงปลายทางโดยไม่มีใครแตะต้อง และบนเครือข่ายระดับดาวเคราะห์ที่เต็มไปด้วยคนแปลกหน้าที่จะไม่มีวันเจอกัน เรื่องนี้ดูเป็นไปไม่ได้

It is not. Two people who have never met can shout across a room where everyone is listening and writing everything down, and finish sharing a number no listener can compute.

> แต่มันเป็นไปได้ คนสองคนที่ไม่เคยเจอกันมาก่อนสามารถตะโกนข้ามห้องที่ทุกคนกำลังฟังและจดทุกคำไว้ แล้วจบลงด้วยการมีตัวเลขร่วมกันตัวหนึ่งที่ไม่มีผู้ฟังคนไหนคำนวณได้เลย

The strangest part of that revolution is that it had already happened, in silence. In a January 1970 report, James Ellis of Britain's GCHQ argued that "non-secret encryption" was possible; Clifford Cocks built a working scheme in 1973 and Malcolm Williamson another in 1974. All of it was classified. GCHQ released the papers on 17 December 1997, the day before Cocks announced them in public — a month after Ellis had died on 25 November 1997, never permitted to say what he had done (GCHQ).

> ส่วนที่แปลกที่สุดของการปฏิวัติครั้งนี้คือมันเกิดขึ้นไปแล้วอย่างเงียบๆ ก่อนหน้านั้น ในรายงานเดือนมกราคม 1970 James Ellis แห่ง GCHQ ของอังกฤษ เสนอว่า "การเข้ารหัสแบบไม่ลับ" (non-secret encryption) เป็นไปได้ Clifford Cocks สร้างสคีมาที่ใช้งานได้จริงในปี 1973 และ Malcolm Williamson สร้างอีกแบบในปี 1974 ทั้งหมดนี้ถูกจัดเป็นความลับ GCHQ ปล่อยเอกสารออกมาวันที่ 17 ธันวาคม 1997 หนึ่งวันก่อนที่ Cocks จะประกาศต่อสาธารณะ — หนึ่งเดือนหลังจาก Ellis เสียชีวิตไปเมื่อวันที่ 25 พฤศจิกายน 1997 โดยไม่เคยได้รับอนุญาตให้พูดว่าตัวเองทำอะไรไว้บ้าง (GCHQ)

Assume the enemy has your blueprints. In 1883 Auguste Kerckhoffs published six requirements for a field cipher in the *Journal des sciences militaires*. Five are period furniture. The second has outlived them all: the system "must not require secrecy and can be stolen by the enemy without causing trouble" (Kerckhoffs, 1883). Claude Shannon restated the rule in the 1949 paper that made cryptography a mathematical science (*Communication Theory of Secrecy Systems*), and modern standards still say it plainly — security "should not depend on the secrecy of the implementation or its components" (NIST SP 800-123). A key can be replaced in seconds by one person; a hidden design baked into a million shipped devices cannot be replaced at all, needing only one captured device to become public. Concentrate the secrecy in the smallest, most replaceable thing you own. That thing is the key.

> สมมติว่าศัตรูมีพิมพ์เขียวของคุณอยู่ในมือแล้ว ในปี 1883 Auguste Kerckhoffs ตีพิมพ์ข้อกำหนดหกข้อสำหรับรหัสลับภาคสนามใน Journal des sciences militaires ห้าข้อเป็นของแต่งยุคนั้นที่ล้าสมัยไปแล้ว แต่ข้อที่สองอยู่รอดมาถึงทุกวันนี้ — ระบบ "ต้องไม่ต้องอาศัยความลับ และต่อให้ถูกศัตรูขโมยไปก็ไม่ก่อปัญหา" (must not require secrecy and can be stolen by the enemy without causing trouble) (Kerckhoffs, 1883) Claude Shannon นำกฎนี้มาเขียนใหม่ในเปเปอร์ปี 1949 ที่ทำให้วิทยาการรหัสลับกลายเป็นศาสตร์เชิงคณิตศาสตร์ (Communication Theory of Secrecy Systems) และมาตรฐานยุคปัจจุบันก็ยังพูดตรงๆ แบบเดียวกัน — ความปลอดภัย "ไม่ควรขึ้นอยู่กับความลับของการนำไปใช้งานจริงหรือส่วนประกอบของมัน" (should not depend on the secrecy of the implementation or its components) (NIST SP 800-123) กุญแจเปลี่ยนใหม่ได้ในไม่กี่วินาทีโดยคนคนเดียว แต่ดีไซน์ที่ซ่อนไว้ในอุปกรณ์นับล้านเครื่องที่ส่งออกไปแล้วเปลี่ยนไม่ได้เลย แค่อุปกรณ์ถูกยึดไปเครื่องเดียวก็เปิดเผยสู่สาธารณะได้ทันที ฉะนั้นให้รวมความลับไว้ที่สิ่งที่เล็กที่สุดและเปลี่ยนได้ง่ายที่สุดที่คุณมี สิ่งนั้นคือกุญแจ

Some doors swing one way. Multiplying two large primes takes a modern processor a fraction of a millisecond. Recovering those primes from the product has no known classical method that finishes in a useful lifetime. Cryptography lives inside gaps like this — computations cheap forward and

ruinous backward. Here is the honest part: nobody has proved such functions exist. Their existence would imply P ≠ NP; the reverse implication is not known, so even settling that famous question would not settle this one (status). Every encrypted payment and every signed software update rests on a conjecture that has survived fifty years of attack — which is evidence, not proof.

> ประตูบางบานแกว่งได้ทางเดียว การคูณจำนวนเฉพาะขนาดใหญ่สองตัวใช้เวลาโปรเซสเซอร์สมัยใหม่แค่เศษเสี้ยวมิลลิวินาที แต่การย้อนหาจำนวนเฉพาะทั้งสองตัวจากผลคูณนั้นไม่มีวิธีแบบคลาสสิกวิธีไหนที่รู้จักกันที่จะเสร็จได้ในช่วงชีวิตที่มีประโยชน์ วิทยาการรหัสลับอาศัยอยู่ในช่องว่างแบบนี้แหละ — คำนวณไปข้างหน้าถูกๆ แต่ย้อนกลับแพงจนแทบเป็นไปไม่ได้ ส่วนที่ต้องพูดตรงๆ คือ ไม่มีใครพิสูจน์ได้ว่าฟังก์ชันทางเดียว (one-way function) แบบนี้มีอยู่จริง การมีอยู่ของมันจะบอกเป็นนัยว่า P ≠ NP แต่นัยกลับด้านไม่เป็นจริงเสมอไป ดังนั้นต่อให้ตัดสินคำถามชื่อดังข้อนั้นได้ ก็ยังไม่ได้ตัดสินข้อนี้ไปด้วย (status) การจ่ายเงินที่เข้ารหัสทุกครั้งและการอัปเดตซอฟต์แวร์ที่เซ็นกำกับทุกครั้งวางอยู่บนข้อสันนิษฐานที่รอดพ้นการโจมตีมาห้าสิบปี — นั่นคือหลักฐาน ไม่ใช่บทพิสูจน์

A secret can be agreed in public. The Diffie–Hellman construction is worth understanding once, because it feels like sleight of hand and is not. Both parties agree publicly on a large prime and a base number. Each privately picks an exponent, raises the base to it, and shouts the result. Each then raises the *other* person's shouted number to their own private exponent. Both arrive at the same value, because exponentiation does not care about the order. An eavesdropper holds both public results and cannot get there, because recovering an exponent from its result — the discrete logarithm — is one of those one-way doors. Two years later, Rivest, Shamir and Adleman turned the same asymmetry the other way round (RSA, 1978): not hiding a message, but signing it, so anyone can check who wrote it and no one can forge it. Diffie and Hellman received the Turing Award for 2015 (ACM).

> ความลับตกลงกันในที่แจ้งได้ โครงสร้าง Diffie–Hellman คุ้มค่าที่จะทำความเข้าใจสักครั้ง เพราะมันให้ความรู้สึกเหมือนกลมายากิ ทั้งที่จริงแล้วไม่ใช่เลย ทั้งสองฝ่ายตกลงกันในที่แจ้งเรื่องจำนวนเฉพาะขนาดใหญ่ตัวหนึ่งกับเลขฐานตัวหนึ่ง แต่ละฝ่าย

> เลือกเลขชี้กำลังเป็นการส่วนตัว ยกเลขฐานขึ้นเป็นเลขชี้กำลังนั้น แล้วตะโกนผลลัพธ์ออกไป จากนั้นแต่ละฝ่ายก็ยกตัวเลขที่ อีกฝ่าย ตะโกนออกมา ขึ้นเป็นเลขชี้กำลังส่วนตัวของตัวเอง ทั้งสองฝ่ายจะได้ค่าเดียวกัน เพราะการยกกำลังไม่สนใจลำดับก่อนหลัง คนดักฟังถือผลลัพธ์สาธารณะทั้งสองค่าไว้แต่ไปต่อไม่ได้ เพราะการย้อนหาเลขชี้กำลังจากผลลัพธ์ของมัน — ลอการิทึมไม่ต่อเนื่อง (discrete logarithm) — ก็เป็นประตูทางเดียวแบบหนึ่งเหมือนกัน สองปีต่อมา Rivest, Shamir และ Adleman พลิกความไม่สมมาตรแบบเดียวกันนี้กลับด้าน (RSA, 1978) ไม่ใช่การซ่อนข้อความ แต่เป็นการเซ็นกำกับมัน ให้ใครก็ตรวจได้ว่าใครเป็นคนเขียน และไม่มีใครปลอมได้ Diffie กับ Hellman ได้รับรางวัล Turing Award ประจำปี 2015 (ACM)

Encryption is one promise out of three, and usually not the one you needed. Confidentiality means nobody else can read it. Integrity means nobody altered it in flight. Authentication means it came from who it claims. These are separable, and the failure mode is unglamorous: an attacker who cannot read your ciphertext may still flip bits inside it, or replay yesterday's valid message today. Most real protocol disasters are integrity or authentication failures wearing an encrypted coat.

> การเข้ารหัสเป็นคำมั่นแค่หนึ่งในสามข้อ และมักไม่ใช่ข้อที่คุณต้องการจริงๆ ด้วยซ้ำ ความลับ (confidentiality) หมายถึงไม่มีใครอื่นอ่านมันออก ความครบถ้วน (integrity) หมายถึงไม่มีใครแก้ไขมันระหว่างทาง การพิสูจน์ตัวตน (authentication) หมายถึงมันมาจากคนที่มันอ้างว่าเป็นจริงๆ ทั้งสามอย่างนี้แยกจากกันได้ และรูปแบบความล้มเหลวก็ไม่หวือหวาเลย — ผู้โจมตีที่อ่าน ciphertext ของคุณไม่ออกก็ยังสลับบิตข้างในมันได้ หรือเอาข้อความที่ถูกต้องของเมื่อวานมาเล่นซ้ำวันนี้ได้ ความหายนะของโพรโทคอลในโลกจริงส่วนใหญ่คือความล้มเหลวด้าน integrity หรือ authentication ที่สวมเสื้อคลุมเข้ารหัสไว้เท่านั้นเอง

Design against an adversary, not for a user. Every other floor in this tower asks whether a system works. This one asks what happens when someone actively wants it not to — and gets to choose the floor. Saltzer and Schroeder set out the discipline in 1975 in eight principles that have not needed revision: economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common

mechanism, psychological acceptability (Proc. IEEE 63(9)). Least privilege is the one to carry: every component gets exactly the authority its job requires, so a compromise stays local. Heartbleed, disclosed on 7 April 2014, was a missing bounds check in OpenSSL that let any stranger ask a server for a slice of its own memory (CISA) — no cipher was broken; a leak two floors down drained the secrets held above it. That is chapter 00's law of leaky abstractions with an adversary standing at the crack.

> ออกแบบเพื่อรับมือฝ่ายตรงข้าม ไม่ใช่ออกแบบเพื่อผู้ใช้ ทุกชั้นอื่นในหอคอยนี้ถามแค่ว่าระบบทำงานได้ไหม แต่ชั้นนี้ถามว่าจะเกิดอะไรขึ้นเมื่อมีใครสักคนตั้งใจอยากให้มันทำงานไม่ได้ — แล้วยังเลือกได้ด้วยว่าจะโจมตีชั้นไหน Saltzer กับ Schroeder วางหลักวินัยนี้ไว้ในปี 1975 เป็นหลักการแปดข้อที่ยังไม่ต้องแก้ไขจนถึงวันนี้ — economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, psychological acceptability (Proc. IEEE 63(9)) ข้อที่ควรจำไว้คือ สิทธิ์น้อยที่สุด (least privilege) — แต่ละคอมโพเนนต์ได้รับอำนาจพอดีกับที่งานของมันต้องใช้เท่านั้น ทำให้ถ้าถูกเจาะก็เสียหายอยู่แค่จุดเดียว Heartbleed ที่เปิดเผยเมื่อวันที่ 7 เมษายน 2014 คือการลืมตรวจขอบเขต (bounds check) ใน OpenSSL ที่ปล่อยให้คนแปลกหน้าคนไหนก็ได้ขอหน่วยความจำส่วนหนึ่งของเซิร์ฟเวอร์ไปดู (CISA) — ไม่มีรหัสลับตัวไหนถูกถอดเลย มีแค่รอยรั่วสองชั้นข้างล่างที่ดูดความลับที่เก็บอยู่ชั้นบนออกไปหมด นี่คือกฎว่าด้วย abstraction ที่รั่วเสมอจากบทที่ 00 ที่มีฝ่ายตรงข้ามยืนอยู่ตรงรอยร้าวพอดี

The weakest layer is the one made of people. The Morris worm of 2 November 1988 spread through an estimated 6,000 of roughly 60,000 connected machines (FBI) — an estimate, still quoted as one. The ratio has not improved: Verizon's 2026 Data Breach Investigations Report, covering incid-

ents from 1 November 2024 to 31 October 2025, finds a human element in 62% of breaches (DBIR). Mathematics defends the wire. It does not defend the person who is tired at the end of a shift.

> ชั้นที่อ่อนแอที่สุดคือชั้นที่ทำจากคน หนอน Morris เมื่อวันที่ 2 พฤศจิกายน 1988 แพร่กระจายไปในเครื่องราว 6,000 เครื่องจากทั้งหมดราว 60,000 เครื่องที่เชื่อมต่ออยู่ในตอนนั้น (FBI) — เป็นตัวเลขประมาณการ ที่ทุกวันนี้ก็ยังถูกอ้างเป็นตัวเลขประมาณการอยู่ดี สัดส่วนนี้ไม่ได้ดีขึ้นเลย — รายงาน Data Breach Investigations Report ปี 2026 ของ Verizon ที่ครอบคลุมเหตุการณ์ตั้งแต่วันที่ 1 พฤศจิกายน 2024 ถึง 31 ตุลาคม 2025 พบว่า 62% ของการรั่วไหลมีปัจจัยจากมนุษย์เข้ามาเกี่ยวข้อง (DBIR) คณิตศาสตร์ป้องกันสายส่งได้ แต่ป้องกันคนที่เหนื่อยล้าตอนใกล้หมดกะงานไม่ได้

A clock is running on all of it. Peter Shor showed in 1994 that a sufficiently large quantum computer would factor integers and compute discrete logarithms efficiently — dissolving exactly the two one-way doors that public-key cryptography stands on. No such machine exists (chapter 14 weighs how far away it is), but encrypted traffic recorded today can be decrypted the day one arrives, a strategy standards bodies now name openly as *harvest now, decrypt later* (NIST IR 8547). The replacement is already standardized: NIST finalized FIPS 203, 204 and 205 on 13 August 2024, and selected the code-based algorithm HQC as a backup on 11 March 2025 (NIST). The engineering problem is not inventing the new mathematics. It is finding, in a civilization's worth of software, every place the old mathematics was quietly assumed.

> นาฬิกากำลังเดินอยู่กับเรื่องนี้ทั้งหมด Peter Shor แสดงไว้ในปี 1994 ว่าคอมพิวเตอร์ควอนตัมที่ใหญ่พอจะแยกตัวประกอบของจำนวนเต็มและคำนวณลอการิทึมไม่ต่อเนื่องได้อย่างมีประสิทธิภาพ — ซึ่งทำลายประตูทางเดียวสองบานพอดีที่วิทยาการรหัสลับแบบกุญแจสาธารณะ (public-key) ยืนอยู่บน ยังไม่มีเครื่องแบบนั้นอยู่จริง (บทที่ 14 ชั่งน้ำหนักว่ามันยังอีกไกลแค่ไหน) แต่ทราฟฟิกที่เข้ารหัสไว้แล้วบันทึกเก็บไว้วันนี้ ก็ถอดรหัสได้ในวันที่เครื่องแบบนั้นมาถึง กลยุทธ์นี้หน่วยงานมาตรฐานเรียกอย่างเปิดเผยแล้วว่า เก็บเกี่ยวตอนนี้ ถอดรหัสทีหลัง (harvest now, decrypt later)

(NIST IR 8547) ตัวแทนที่จะมาแทนก็ถูกกำหนดเป็นมาตรฐานไปแล้ว — NIST สรุป FIPS 203, 204 และ 205 เป็นทางการเมื่อวันที่ 13 สิงหาคม 2024 และเลือกอัลกอริทึมแบบ code-based ชื่อ HQC เป็นตัวสำรองเมื่อวันที่ 11 มีนาคม 2025 (NIST) โจทย์ทางวิศวกรรมไม่ใช่การคิดคณิตศาสตร์ใหม่ แต่คือการตามหา ในซอฟต์แวร์ทั้งหมดเท่าที่อารยธรรมนี้เคยเขียนขึ้นมา ทุกจุดที่คณิตศาสตร์แบบเก่าเคยถูกสมมติไว้เงียบๆ ว่าจะยังใช้ได้เสมอ

Connections · เชื่อมโยง

Below: chapter 03 supplies the raw material — a key is only as good as its unpredictability, which entropy measures. Chapters 04 and 05 hold the roots of trust, since a kernel that can be subverted makes every guarantee above it decorative. Alongside: chapter 09 explains why the internet's naming and routing were built for a cooperative world and had authentication bolted on later, and chapter 10 asks how parties agree when some of them have merely crashed — this chapter's adversary upgrades crashing to lying. Above: chapter 12, where the adversary attacks the training data rather than the wire.

ชั้นล่าง: บทที่ 03 ส่งวัตถุดิบมาให้ — กุญแจจะดีได้ก็แค่เท่าที่มันคาดเดาไม่ได้ ซึ่งวัดกันด้วยเอนโทรปี (entropy) บทที่ 04 กับ 05 ถือรากฐานของความไว้ใจไว้ เพราะเคอร์เนลที่ถูกล้มล้างได้ ทำให้การรับประกัน (guarantee) ทุกอย่างที่อยู่เหนือมันกลายเป็นแค่ของประดับ ชั้นข้างเคียง: บทที่ 09 อธิบายว่าทำไมการตั้งชื่อและการจัดเส้นทางของอินเทอร์เน็ตถูกสร้างมาเพื่อโลกที่ร่วมมือกัน แล้วค่อยเอาการพิสูจน์ตัวตนมาติดเพิ่มทีหลัง และบทที่ 10 ถามว่าฝ่ายต่างๆ ตกลงกันได้อย่างไรเมื่อบางฝ่ายแค่ล่มไปเฉยๆ — ฝ่ายตรงข้ามในบทนี้ยกระดับจาก "ล่ม" ไปเป็น "โกหก" ชั้นบน: บทที่ 12 ที่ฝ่ายตรงข้ามโจมตีข้อมูลเทรนแทนที่จะโจมตีสายส่ง

Open questions · คำถามที่ยังเปิดอยู่

Do one-way functions exist? Everything in this chapter is a bet that they do. Can the world's deployed cryptography be migrated to post-quantum algorithms before a cryptographically relevant quantum computer exists, given that nobody knows the date of either event? And the question the field

keeps failing rather than solving: can systems be made secure *and* usable, given that Saltzer and Schroeder listed psychological acceptability fifty years ago and it remains the least-honoured of their eight principles?

> ฟังก์ชันทางเดียวมีอยู่จริงไหม? ทุกอย่างในบทนี้คือการพนันว่ามันมีอยู่จริง วิทยาการรหัสลับที่ใช้งานอยู่ทั่วโลกจะย้ายไปใช้อัลกอริทึม post-quantum ได้ทันก่อนที่คอมพิวเตอร์ควอนตัมระดับที่ทำลายรหัสลับได้จะมีอยู่จริงไหม ในเมื่อไม่มีใครรู้วันที่ของเหตุการณ์ทั้งสองฝั่งเลย และคำถามที่วงการนี้ทำพลาดซ้ำแล้วซ้ำเล่ามากกว่าจะแก้ได้จริง — ระบบจะปลอดภัย และ ใช้งานง่ายไปพร้อมกันได้ไหม ในเมื่อ Saltzer กับ Schroeder ระบุ psychological acceptability ไว้ตั้งห้าสิบปีก่อน แล้วมันก็ยังเป็นข้อที่ถูกให้ความสำคัญน้อยที่สุดในแปดข้อของพวกเขาอยู่ดี

Go deeper · อ่านต่อ

- Alfred Menezes, Paul van Oorschot & Scott Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996 — the standard reference, all chapters free with the publisher's permission at cacr.uwaterloo.ca/hac.

  > Alfred Menezes, Paul van Oorschot & Scott Vanstone, Handbook of Applied Cryptography, CRC Press, 1996 — เอกสารอ้างอิงมาตรฐาน ทุกบทเปิดให้อ่านฟรีโดยได้รับอนุญาตจากสำนักพิมพ์ที่ cacr.uwaterloo.ca/hac

- Whitfield Diffie & Martin Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory* IT-22, no. 6 (1976): 644–654 — PDF.

  > Whitfield Diffie & Martin Hellman, "New Directions in Cryptography," IEEE Transactions on Information Theory IT-22, no. 6 (1976): 644–654 — PDF

- Jerome Saltzer & Michael Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE* 63, no. 9 (1975): 1278–1308 — PDF.

  > Jerome Saltzer & Michael Schroeder, "The Protection of Information in Computer Systems," Proceedings of the IEEE 63, no. 9 (1975): 1278–1308 — PDF

CHAPTER 12

AI & Machine Learning

ปัญญาประดิษฐ์และการเรียนรู้ของเครื่อง

This chapter is the tower's other stairwell — it touches every floor, and hides the oldest assumption in the tower: that somebody, somewhere, wrote the rule down.

บทนี้คือช่องบันไดอีกช่องหนึ่งของหอคอย — มันแตะทุกชั้น และซ่อนสมมติฐานที่เก่าแก่ที่สุดของหอคอยไว้ นั่นคือมีใครสักคน ที่ไหนสักแห่ง เขียนกฎนั้นไว้แล้ว

Can behavior be grown from data instead of written by hand?

พฤติกรรมงอกออกมาจากข้อมูลได้ไหม แทนที่จะถูกเขียนด้วยมือ?

In July 1959 the *IBM Journal of Research and Development* published Arthur Samuel's account of a checkers program that had started beating him (Samuel, 1959). This should have been impossible in the terms the rest of this book has used. Every floor below is built from rules a person stated: a gate's truth table, a protocol's state machine, a query optimizer's cost model. Samuel had stated the rules of checkers, not the rules of playing checkers *well* — that part the program accumulated by playing, adjusting the weight it gave to board features according to which ones had preceded winning. Its skill existed nowhere in his source code. It existed in numbers no human chose. Everything since has been an argument about how far that idea goes.

> ในเดือนกรกฎาคม 1959 วารสาร IBM Journal of Research and Development ตีพิมพ์บันทึกของ Arthur Samuel เรื่องโปรแกรมเล่นหมากฮอส (checkers) ที่เริ่มเอาชนะเขาเองได้ (Samuel, 1959) เรื่องนี้ควรจะเป็นไปไม่ได้ตามเงื่อนไขที่หนังสือเล่มนี้ใช้มาตลอด ทุกชั้นก่อนหน้านี้สร้างขึ้นจากกฎที่คนเป็นผู้กำหนด ไม่ว่าจะเป็นตารางค่าความจริงของเกต สเตตแมชชีนของโพรโทคอล หรือโมเดลต้นทุนของ query optimizer Samuel กำหนดกฎของหมากฮอสไว้ แต่ไม่ได้กำหนดกฎของการเล่น

> หมากฮอสให้ เก่ง — ส่วนนั้นโปรแกรมสะสมเอาจากการเล่นจริง ปรับน้ำหนักที่ให้กับฟีเจอร์ (feature) บนกระดาน ตามว่า feature ไหนเคยนำไปสู่ชัยชนะ ฝีมือของมันไม่ได้อยู่ในซอร์สโค้ดของ Samuel ที่ไหนเลย มันอยู่ในตัวเลขที่ไม่มีมนุษย์คนไหนเลือกเอง ตั้งแต่นั้นมาทุกอย่างคือการถกเถียงกันว่าไอเดียนี้ไปได้ไกลแค่ไหน

Learning means generalizing, and generalizing is the only hard part. A system that reproduces its training data perfectly has learned nothing; a lookup table does that. The measure that matters is performance on data the system has never seen, and the gap between the two — the model fitting noise it should have ignored, "overfitting" — is what every technique in the field is ultimately fighting. Wolpert and Macready's No Free Lunch theorem shows that averaged over *all* possible objective functions, every learning or optimization algorithm performs identically (1997). Every bit of real learning ability comes from assumptions baked in about which problems are likely — that nearby pixels are related, that word meaning depends on context, that the world is smoother than it could be. A learner with no prejudices cannot learn. Choosing the prejudices is the engineering.

> การเรียนรู้ก็คือการใช้ได้กับของที่ไม่เคยเห็น (generalization) และ generalization คือส่วนเดียวที่ยากจริงๆ ระบบที่จำข้อมูลเทรน (training data) ของตัวเองได้แม่นเป๊ะ ไม่ได้เรียนรู้อะไรเลย นั่นคืองานของตารางค้นหา (lookup table) ตัวชี้วัดที่สำคัญคือประสิทธิภาพบนข้อมูลที่ระบบไม่เคยเห็นมาก่อน และช่องว่างระหว่างสองอย่างนี้ — โมเดล (model) ไปฟิตกับ noise ที่ควรเมินไว้ เรียกว่า "overfitting" — คือสิ่งที่ทุกเทคนิคในสาขานี้พยายามเอาชนะมาตลอด ทฤษฎีบท No Free Lunch ของ Wolpert และ Macready แสดงว่า เมื่อเฉลี่ยข้าม objective function ทั้งหมด ที่เป็นไปได้ อัลกอริทึมเรียนรู้หรือ optimization ทุกตัวทำงานได้เท่ากันหมด (1997) ความสามารถในการเรียนรู้จริงทุกหยดมาจากสมมติฐานที่ฝังไว้ล่วงหน้าว่าโจทย์แบบไหนน่าจะเกิดขึ้น — พิกเซลที่อยู่ใกล้กันน่าจะเกี่ยวข้องกัน ความหมายของคำขึ้นกับบริบท โลกเรียบง่ายกว่าที่มันจะเป็นได้ ตัวเรียนรู้ที่ไม่มีอคติเลยเรียนรู้อะไรไม่ได้ การเลือกอคตินั่นแหละคืองานวิศวกรรม

If every step is differentiable, the machine can debug itself. Frank Rosenblatt's perceptron (1958) could adjust its own weights, and the US Navy's press conference promptly told the *New York Times* it expected a device that would eventually walk, talk, see, write, and be conscious of its existence. It could not compute exclusive-or, as Minsky and Papert demonstrated in 1969 for the single-layer case — a limit historians now think did far less damage to the field than the surrounding funding politics (*Perceptrons*). Stacking layers fixes the expressiveness but creates the credit-assignment problem: with millions of weights and one number for how wrong the answer was, which weight was at fault? Backpropagation answers it by running the chain rule of calculus backwards through the network, computing for every weight at once how much a small nudge would have reduced the error, at roughly the cost of one forward pass. Rumelhart, Hinton and Williams brought it to the field's attention in *Nature* in October 1986 (323: 533–536); Seppo Linnainmaa had published the underlying method as reverse-mode automatic differentiation in 1970 and Paul Werbos had applied it to neural networks in a 1974 thesis. This is the floor's real trick, and it is smaller than its reputation: a program you can take the derivative of is a program improved by measurement rather than by argument.

ถ้าทุกสเต็ป differentiable เครื่องก็ดีบักตัวเองได้ perceptron ของ Frank Rosenblatt (1958) ปรับน้ำหนักของตัวเองได้ และงานแถลงข่าวของกองทัพเรือสหรัฐฯ ก็รีบบอก New York Times ทันทีว่าคาดหวังว่าอุปกรณ์นี้จะเดินได้ พูดได้ มองเห็น เขียนได้ และรู้ตัวว่ามันมีอยู่จริงในที่สุด มันคำนวณ exclusive-or ไม่ได้ ตามที่ Minsky กับ Papert พิสูจน์ไว้ในปี 1969 สำหรับกรณี single-layer — ข้อจำกัดที่นักประวัติศาสตร์ยุคนี้เชื่อว่าสร้างความเสียหายให้วงการน้อยกว่าการเมืองเรื่องทุนวิจัยที่ตามมามาก (Perceptrons) การซ้อนเลเยอร์แก้ปัญหาเรื่อง expressiveness ได้ แต่สร้างปัญหา credit-assignment ขึ้นมาแทน — มีน้ำหนักเป็นล้านตัว มีตัวเลขบอกความผิดพลาดแค่ตัวเดียว แล้วน้ำหนักตัวไหนคือตัวที่ผิด backpropagation ตอบโจทย์นี้ด้วยการรัน chain rule ของแคลคูลัสย้อนกลับผ่านเครือข่าย คำนวณให้ทุกน้ำหนักพร้อมกันว่าถ้าขยับนิดหน่อยจะลดความผิดพลาดได้แค่ไหน ด้วยต้นทุนพอๆ กับการรัน forward pass หนึ่งครั้ง Rumelhart, Hinton และ Williams นำเรื่องนี้มาสู่ความสนใจของวงการใน Nature เดือนตุลาคม 1986 (323: 533–536) ส่วน Seppo

> Linnainmaa ตีพิมพ์วิธีที่อยู่เบื้องหลังไว้ในชื่อ reverse-mode automatic differentiation ตั้งแต่ปี 1970 และ Paul Werbos เอาไปใช้กับโครงข่ายประสาทเทียม (neural network) ในวิทยานิพนธ์ปี 1974 นี่คือกลเม็ดจริงของชั้นนี้ และมันเล็กกว่าชื่อเสียงที่มันได้รับ — โปรแกรมที่หาอนุพันธ์ได้ คือโปรแกรมที่พัฒนาด้วยการวัดผล ไม่ใช่ด้วยการถกเถียง

What gets learned is the representation, not just the answer. For decades the labour of machine learning was feature engineering — humans deciding which measurable properties of a photograph or a sentence the statistics should chew on. The shift was to let those intermediate descriptions be learned too (Bengio, Courville & Vincent, 2013). The public demonstration came in 2012, when Krizhevsky, Sutskever and Hinton's convolutional network cut the ImageNet top-5 error rate to 15.3% against the runner-up's 26.2% (NeurIPS 2012) — a margin large enough that a research community changed direction within a year. Nobody had told it what an edge or a texture was. Those appeared in its early layers because they were useful.

> สิ่งที่ถูกเรียนรู้จริงๆ คือ representation ไม่ใช่แค่คำตอบ มาเป็นสิบปีที่แรงงานหลักของ machine learning คือ feature engineering — งานที่คนต้องตัดสินใจเองว่า feature หรือคุณสมบัติที่วัดได้ของรูปถ่ายหรือประโยคแบบไหนที่สถิติควรเอาไปเคี้ยว จุดเปลี่ยนคือปล่อยให้คำอธิบายชั้นกลางเหล่านั้นถูกเรียนรู้ไปด้วย (Bengio, Courville & Vincent, 2013) การสาธิตต่อสาธารณะเกิดขึ้นในปี 2012 ตอนที่ convolutional network ของ Krizhevsky, Sutskever และ Hinton ตัด error rate แบบ top-5 ของ ImageNet ลงเหลือ 15.3% เทียบกับอันดับสองที่ 26.2% (NeurIPS 2012) — ห่างกันมากพอที่ทั้งวงการวิจัยจะเปลี่ยนทิศทางภายในปีเดียว ไม่มีใครบอกมันว่าขอบ (edge) หรือพื้นผิว (texture) คืออะไร สิ่งเหล่านั้นโผล่ขึ้นมาในเลเยอร์แรกๆ ของมันเองเพราะมันมีประโยชน์

Scale beat cleverness, repeatedly, and the people who noticed said so plainly. Rich Sutton's essay of 13 March 2019 states the pattern without hedging: "The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin" (*The Bitter Lesson*). Bitter, because the hu-

man insight painstakingly encoded into a system tends to be the part that has to be thrown away. But scale is not a slogan; it is a quantity with a recipe, and the recipe has already been corrected once. Kaplan and colleagues charted how loss falls with model size, data and compute (2020); two years later the Chinchilla result showed the prevailing practice had the ratio badly wrong — for a fixed compute budget, models were being built too large and fed too little, with roughly 20 training tokens per parameter closer to optimal (Hoffmann et al., 2022). The lesson survived. The arithmetic under it did not.

> สเกล (scale) ชนะความฉลาดแยบยลซ้ำแล้วซ้ำเล่า และคนที่สังเกตเห็นก็พูดตรงๆ ไม่อ้อมค้อม บทความของ Rich Sutton วันที่ 13 มีนาคม 2019 บอกแพตเทิร์นนี้แบบไม่เกรงใจใคร: "บทเรียนที่ใหญ่ที่สุดที่อ่านได้จากงานวิจัยปัญญาประดิษฐ์ (AI) มา 70 ปี คือวิธีการทั่วไปที่ใช้ประโยชน์จาก computation ได้ผลดีที่สุดในที่สุด และดีกว่าแบบทิ้งห่างมาก" ("The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin") (The Bitter Lesson) ที่เรียกว่าขม เพราะข้อมูลเชิงลึกของมนุษย์ที่ใส่เข้าไปในระบบอย่างพิถีพิถัน มักกลายเป็นส่วนที่ต้องถูกทิ้งไป แต่สเกลไม่ใช่สโลแกน มันคือปริมาณที่มีสูตรของมัน และสูตรนั้นเคยถูกแก้ไปแล้วครั้งหนึ่ง Kaplan และคณะวาดกราฟว่า loss ลดลงยังไงตามขนาดโมเดล ข้อมูล และ compute (2020) สองปีให้หลัง ผลของ Chinchilla แสดงว่าแนวทางที่ใช้กันอยู่คำนวณสัดส่วนผิดไปมาก — ด้วยงบ compute คงที่ โมเดลถูกสร้างใหญ่เกินไป และป้อนข้อมูลน้อยเกินไป โดยประมาณ 20 training token ต่อพารามิเตอร์ใกล้เคียงค่าที่เหมาะสมกว่า (Hoffmann et al., 2022) บทเรียนนั้นยังอยู่รอด ส่วนเลขคณิตที่รองรับมันไม่รอด

Attention made scale usable. Sequence models used to force a whole sentence through a fixed-size bottleneck, one step at a time, remembering worse the further back it went. The 2017 architecture proposed by eight authors at Google replaced the recurrence entirely: let every position in a sequence look directly at every other and weight what it finds relevant (*Attention Is All You Need*). What mattered was less memory than hardware — with the sequential dependency gone, training parallelizes across the ma-

chines of chapter 04, and the bitter lesson's compute could actually be spent. (Readers who want that one idea traced from 2014 onward will find it is the subject of its own book in this library.)

> Attention ทำให้สเกลใช้งานได้จริง เดิมทีโมเดลลำดับ (sequence model) ต้องบีบทั้งประโยคผ่านคอขวดขนาดคงที่ ทีละสเต็ป และจำได้แย่ลงเรื่อยๆ ยิ่งย้อนไปไกลเท่าไหร่ สถาปัตยกรรมปี 2017 ที่เสนอโดยผู้เขียนแปดคนจาก Google แทนที่ recurrence ไปทั้งหมด — ให้ทุกตำแหน่งในลำดับมองไปยังตำแหน่งอื่นทุกตำแหน่งได้โดยตรง แล้วถ่วงน้ำหนักตามความเกี่ยวข้องที่พบ (Attention Is All You Need) สิ่งที่สำคัญไม่ใช่หน่วยความจำ แต่เป็นฮาร์ดแวร์ — พอ dependency ตามลำดับหายไป การเทรนก็ทำแบบขนานได้ทั่วเครื่องจักรในบทที่ 04 และ compute ของ bitter lesson ก็ใช้ได้จริงในที่สุด (ผู้อ่านที่อยากตามไอเดียนี้ตั้งแต่ปี 2014 เป็นต้นมา จะพบว่ามันเป็นหัวข้อของหนังสือทั้งเล่มในห้องสมุดนี้)

The limits are structural, not embarrassing. These systems produce fluent, confident, false statements — "hallucination," surveyed rigorously by Ji and colleagues (ACM Computing Surveys, 2023). One prominent argument holds this is not a bug to be patched but the predictable result of training and benchmark regimes that reward a confident guess over an admission of ignorance (Kalai et al., 2025) — one live account among several, not a settled verdict. The deeper issue is that these systems optimize the objective you gave them, which is never quite the thing you wanted. That gap is why the field built machinery to train against human judgments of quality instead (Christiano et al., 2017; Ouyang et al., 2022) — which relocates the problem into the question of whose judgments, rather than dissolving it. And because the behavior is grown from data, the data is itself an attack surface: poison what a system learns from and you have programmed it — chapter 11's adversary, one floor up.

> ข้อจำกัดตรงนี้เป็นเรื่องเชิงโครงสร้าง ไม่ใช่เรื่องน่าอาย ระบบพวกนี้สร้างข้อความที่ลื่นไหล มั่นใจ แต่เท็จ — เรียกว่า "hallucination" ซึ่ง Ji และคณะสำรวจไว้อย่างเข้มงวด (ACM Computing Surveys, 2023) ข้อโต้แย้งหนึ่งที่มีน้ำหนักบอกว่านี่ไม่ใช่บั๊กที่แก้ได้ แต่เป็นผลลัพธ์ที่คาดเดาได้จากระบอบการเทรนและการทำเบนช์มาร์ก (benchmark) ที่ให้รางวัลกับการเดาแบบมั่นใจ มากกว่าการยอมรับว่าไม่รู้ (Kalai et

> al., 2025) — เป็นแค่มุมมองหนึ่งในหลายมุมมองที่ยังไม่นิ่ง ไม่ใช่คำตัดสินที่จบแล้ว ปัญหาที่ลึกกว่านั้นคือระบบพวกนี้ optimize ตาม objective ที่เราให้ไป ซึ่งไม่เคยตรงกับสิ่งที่เราอยากได้จริงๆ เป๊ะๆ ช่องว่างนั้นเองคือเหตุผลที่วงการสร้างกลไกมาเทรนโดยอิงการตัดสินคุณภาพของมนุษย์แทน (Christiano et al., 2017; Ouyang et al., 2022) — ซึ่งย้ายปัญหาไปอยู่ที่คำถามว่าเป็นการตัดสินของใคร ไม่ได้ทำให้ปัญหาหายไป และเพราะพฤติกรรมถูกปลูกขึ้นมาจากข้อมูล ข้อมูลเองก็เป็น attack surface — วางยาพิษให้สิ่งที่ระบบเรียนรู้จากมัน ก็เท่ากับเขียนโปรแกรมมันได้เลย นี่คือฝ่ายตรงข้าม (adversary) ของบทที่ 11 ที่อยู่ชั้นบนขึ้นไปหนึ่งชั้น

The field's own institutions have now conceded the point. Hopfield and Hinton took the 2024 Nobel Prize in Physics for the foundational work enabling machine learning with neural networks; the Chemistry prize that year went to Baker, Hassabis and Jumper for computational protein design and structure prediction (Nobel). A method for growing behavior from data had become an instrument of natural science.

> สถาบันของวงการเองก็ยอมรับประเด็นนี้แล้ว Hopfield กับ Hinton ได้รางวัลโนเบลสาขาฟิสิกส์ปี 2024 จากงานพื้นฐานที่ทำให้ machine learning ด้วย neural network เป็นไปได้ ส่วนรางวัลสาขาเคมีปีเดียวกันตกเป็นของ Baker, Hassabis และ Jumper จากงานออกแบบโปรตีนเชิงคำนวณและการทำนายโครงสร้าง (Nobel) วิธีการปลูกพฤติกรรมขึ้นจากข้อมูลได้กลายเป็นเครื่องมือของวิทยาศาสตร์ธรรมชาติไปแล้ว

Connections · เชื่อมโยง

Below: chapter 02, because a gradient step is an algorithm with a cost — and training is dominated by exactly the matrix multiplication whose exponent that floor is still chasing — and chapter 04, because this floor's history is the history of what hardware made affordable — 2012 was not a new idea but an old idea a graphics processor could finally afford. Chapter 03 supplies the framing of a model as compression, and chapter 01 draws the outer boundary: learning does not repeal undecidability. Sideways: like security, learning from data is not really a floor with a single tenant. It has been

quietly installed on nearly every floor — branch predictors in 04, query optimizers in 08, congestion control in 09 — which is why chapter 00's map gave it a stairwell rather than a floor.

> ชั้นล่าง: บทที่ 02 เพราะ gradient step หนึ่งครั้งก็คืออัลกอริทึมที่มีต้นทุน — และการเทรนถูกครอบงำโดยการคูณเมทริกซ์ตัวเดียวกับที่เลขชี้กำลังของชั้นนั้นยังไล่ตามอยู่ — และบทที่ 04 เพราะประวัติของชั้นนี้คือประวัติของสิ่งที่ฮาร์ดแวร์ทำให้จ่ายไหว — ปี 2012 ไม่ใช่ไอเดียใหม่ แต่เป็นไอเดียเก่าที่ graphics processor เพิ่งจ่ายไหวในที่สุด บทที่ 03 มอบกรอบคิดเรื่องโมเดลในฐานะการบีบอัด (compression) และบทที่ 01 ขีดขอบเขตด้านนอกไว้ว่า การเรียนรู้ไม่ได้ยกเลิกความตัดสินไม่ได้ (undecidability) ด้านข้าง: เหมือนกับความปลอดภัย การเรียนรู้จากข้อมูลไม่ใช่ชั้นที่มีผู้เช่ารายเดียวจริงๆ มันถูกติดตั้งเงียบๆ ไว้แทบทุกชั้น — branch predictor ในบทที่ 04, query optimizer ในบทที่ 08, congestion control ในบทที่ 09 — นี่คือเหตุผลที่แผนที่ในบทที่ 00 ให้มันเป็นช่องบันไดแทนที่จะเป็นชั้น

Open questions · คำถามที่ยังเปิดอยู่

Does the bitter lesson keep holding, or was it a description of one era with unusually cheap compute — and how would the field tell the difference before the fact? Is generalization from data a route to understanding, or a good imitation of it that fails in ways we notice only afterwards? And the question chapter 00 left standing: if people increasingly command machines in ordinary language, is natural language becoming the newest floor of the tower — a surface under which every abstraction in this book is hidden, including the leaks?

> the bitter lesson จะยังยืนอยู่ต่อไปไหม หรือมันเป็นแค่คำอธิบายของยุคหนึ่งที่ compute ราคาถูกอย่างผิดปกติ — แล้ววงการจะแยกความต่างนี้ได้ก่อนที่มันจะเกิดขึ้นจริงยังไง การใช้ได้กับของที่ไม่เคยเห็น (generalization) จากข้อมูล คือเส้นทางไปสู่ความเข้าใจ หรือเป็นแค่การเลียนแบบที่ดีของความเข้าใจ ซึ่งล้มเหลวในแบบที่เราสังเกตเห็นได้ก็ต่อเมื่อมันเกิดขึ้นไปแล้ว และคำถามที่บทที่ 00 ทิ้งค้างไว้ — ถ้าคนสั่ง

> เครื่องด้วยภาษาธรรมดามากขึ้นเรื่อยๆ ภาษาธรรมชาติกำลังจะกลายเป็นชั้นใหม่ล่าสุดของหอคอยหรือเปล่า — ผิว (surface) ที่นามธรรม (abstraction) ทุกอันในเล่มนี้ถูกซ่อนอยู่ข้างใต้ รวมถึงจุดที่มันรั่วด้วย

Go deeper · อ่านต่อ

— Ian Goodfellow, Yoshua Bengio & Aaron Courville, *Deep Learning*, MIT Press, 2016 — the standard graduate text, full text free at deeplearningbook.org.

> Ian Goodfellow, Yoshua Bengio & Aaron Courville, Deep Learning, MIT Press, 2016 — ตำราระดับบัณฑิตศึกษามาตรฐาน อ่านฉบับเต็มได้ฟรีที่ deeplearningbook.org

— David Rumelhart, Geoffrey Hinton & Ronald Williams, "Learning representations by back-propagating errors," *Nature* 323 (1986): 533–536 — nature.com.

> David Rumelhart, Geoffrey Hinton & Ronald Williams, "Learning representations by back-propagating errors," Nature 323 (1986): 533–536 — nature.com

— Jordan Hoffmann et al., "Training Compute-Optimal Large Language Models," 2022 — arXiv:2203.15556.

> Jordan Hoffmann et al., "Training Compute-Optimal Large Language Models," 2022 — arXiv:2203.15556

CHAPTER 13

HCI & Graphics

ปฏิสัมพันธ์มนุษย์–คอมพิวเตอร์ และกราฟิกส์

This chapter is the top floor — the only surface in the tower a person actually touches, and the one that has to hide every floor beneath it from someone who never asked for a tower.

บทนี้คือชั้นบนสุด — ผิว (surface) เดียวในหอคอยที่คนสัมผัสได้จริง และเป็นชั้นที่ต้องซ่อนทุกชั้นข้างใต้ไว้จากคนที่ไม่เคยขอหอคอยเลยด้วยซ้ำ

Where should the machine end and the person begin?

เครื่องควรจบตรงไหน แล้วคนควรเริ่มตรงไหน?

On 9 December 1968, in Brooks Hall in San Francisco's Civic Auditorium complex, Douglas Engelbart sat down in front of roughly a thousand computing professionals with a screen, a keyboard and a wooden box on wheels, and spent ninety minutes doing things none of them had seen (Doug Engelbart Institute). He moved a pointer with the box. He edited a document and it changed as he typed. He worked on the same file, live, with a colleague thirty miles away whose face was on the screen beside the text. Mouse, hypertext, real-time collaborative editing, video conferencing — in one sitting, in a decade when the normal way to use a computer was to hand over a deck of punched cards and come back tomorrow.

> วันที่ 9 ธันวาคม 1968 ที่ Brooks Hall ในกลุ่มอาคาร Civic Auditorium ของซานฟรานซิสโก Douglas Engelbart นั่งลงต่อหน้าคนทำงานคอมพิวเตอร์ราวพันคน พร้อมจอภาพ คีย์บอร์ด และกล่องไม้มีล้อ แล้วใช้เวลาเก้าสิบนาทีทำสิ่งที่ไม่มีใครในนั้นเคยเห็นมาก่อน (Doug Engelbart Institute) เขาขยับตัวชี้ตำแหน่งด้วยกล่องนั้น เขาแก้ไขเอกสารแล้วมันเปลี่ยนไปตามที่พิมพ์ เขาทำงานบนไฟล์เดียวกัน สดๆ ร่วมกับเพื่อนร่วมงานที่อยู่ห่างไปสามสิบไมล์ ซึ่งหน้าตาปรากฏอยู่บนจอข้างๆ ข้อความ

> เมาส์ ไฮเปอร์เท็กซ์ (hypertext) การแก้ไขร่วมกันแบบเรียลไทม์ การประชุมทางวิดีโอ — ทั้งหมดในการนั่งครั้งเดียว ในยุคที่วิธีปกติของการใช้คอมพิวเตอร์คือส่งมอบตั้งบัตรเจาะรู (punched cards) แล้วค่อยกลับมาเอาผลพรุ่งนี้

The demonstration is remembered for the inventions. The argument underneath it was more radical, and Engelbart had published it six years earlier: the point of a computer is not to automate human work but to *augment* human intellect, which requires machine and person to be inside the same continuous loop rather than taking turns (SRI, October 1962).

> การสาธิตครั้งนี้ถูกจดจำเพราะสิ่งประดิษฐ์ต่างๆ แต่ข้อโต้แย้งที่อยู่ข้างใต้มันสุดโต่งกว่านั้นมาก และ Engelbart ตีพิมพ์มันไว้ตั้งแต่หกปีก่อนหน้า — จุดมุ่งหมายของคอมพิวเตอร์ไม่ใช่การทำงานของมนุษย์ให้เป็นอัตโนมัติ แต่คือการ เสริม สติปัญญาของมนุษย์ ซึ่งต้องการให้เครื่องกับคนอยู่ในลูปต่อเนื่องเดียวกัน แทนที่จะผลัดกันทำทีละฝ่าย (SRI, October 1962)

An interface is shared state between two minds. The machine holds a model of the task. The person holds a model of the machine. An interface is the narrow channel keeping those two models in agreement, and essentially every interface failure is a divergence between them: the file the user believes was saved, the setting they believe is off, the button they believe did nothing. This is why feedback is not politeness. A system that acts without visibly acknowledging it has let the two models drift, and the person will keep operating on the stale one.

> อินเทอร์เฟซ (interface) คือสถานะ (state) ที่ใช้ร่วมกันระหว่างสองความคิด เครื่องถือแบบจำลองในหัว (mental model) ของงานเอาไว้ คนถือ mental model ของเครื่องเอาไว้ อินเทอร์เฟซคือช่องแคบๆ ที่คอยรักษาให้ mental model ทั้งสองนี้ตรงกัน และความล้มเหลวของอินเทอร์เฟซแทบทุกครั้งก็คือจุดที่ mental model ทั้งสองเบี่ยงออกจากกัน — ไฟล์ที่ผู้ใช้ (user) เชื่อว่าบันทึกแล้ว การตั้งค่าที่เขาเชื่อว่าปิดอยู่ ปุ่มที่เขาเชื่อว่ากดแล้วไม่มีอะไรเกิดขึ้น นี่คือเหตุผลที่การตอบสนอง (feedback) ไม่

> ใช่เรื่องมารยาท ระบบที่ทำงานโดยไม่แสดงให้เห็นว่ามันรับรู้ ปล่อยให้ mental model ทั้งสองเบี่ยงออกจากกัน และคนก็จะยังทำงานต่อไปบน mental model เก่าที่ล้าไปแล้ว

Show the object; act on it directly. Ivan Sutherland's Sketchpad, an MIT doctoral thesis submitted in 1963, let a user draw with a light pen on a screen and have the drawing answer back — constraints maintained, shapes copied and scaled (Cambridge reissue). Xerox PARC's Alto put the idea into a research machine in 1973; the Xerox Star of April 1981 shipped it commercially, with files as icons on a desktop (Star). Ben Shneiderman named the principle in 1983 and gave it three properties still worth reciting: continuous visual representation of the object of interest; physical actions or labeled button presses in place of complex command syntax; and rapid, incremental, reversible operations whose effect is immediately visible (*IEEE Computer* 16(8)). The third is the quietly humane one. Reversibility converts a machine from an examination into a place to think, because it makes exploring cost nothing.

> แสดงวัตถุ แล้วลงมือกระทำกับมันโดยตรง — นี่คือหลักการควบคุมโดยตรง (direct manipulation) Sketchpad ของ Ivan Sutherland ซึ่งเป็นวิทยานิพนธ์ปริญญาเอกที่ MIT ยื่นในปี 1963 ให้ผู้ใช้วาดด้วยปากกาแสง (light pen) บนจอภาพ แล้วภาพวาดก็ตอบสนองกลับมา — คงเงื่อนไขไว้ได้ คัดลอกและปรับขนาดรูปทรงได้ (Cambridge reissue) Alto ของ Xerox PARC เอาไอเดียนี้ใส่ลงในเครื่องวิจัยจริงในปี 1973 ส่วน Xerox Star เดือนเมษายน 1981 ก็ส่งออกขายเชิงพาณิชย์ โดยมีไฟล์เป็นไอคอนบนเดสก์ท็อป (Star) Ben Shneiderman ตั้งชื่อหลักการนี้ในปี 1983 และให้คุณสมบัติสามข้อที่ยังท่องได้จนถึงวันนี้ — การแสดงวัตถุที่สนใจอย่างต่อเนื่องด้วยภาพ การกระทำทางกายภาพหรือการกดปุ่มที่มีป้ายกำกับแทนไวยากรณ์คำสั่งที่ซับซ้อน และการทำงานที่รวดเร็ว เป็นขั้นๆ ย้อนกลับได้ ซึ่งผลลัพธ์มองเห็นได้ทันที (IEEE Computer 16(8)) ข้อที่สามคือข้อที่มีมนุษยธรรมอยู่เงียบๆ การย้อนกลับได้เปลี่ยนเครื่องจากสนามสอบให้กลายเป็นที่สำหรับคิด เพราะมันทำให้การลองผิดลองถูกไม่มีต้นทุนเลย

Affordance is a relationship; most of what you see is a signifier. J.J. Gibson introduced *affordance* in 1966 and developed it in his 1979 ecological theory: what an environment offers a particular creature — a chair-height surface affords sitting to an adult and not to an infant (affordance). Don Norman brought the word into design in 1988 in *The Psychology of Everyday Things*, later retitled *The Design of Everyday Things*, then spent twenty-five years watching designers misuse it to mean "visual hint" — so the 2013 revision introduced *signifier* for the perceptible mark advertising where an action is possible (Norman). The distinction is load-bearing on a touchscreen, because a sheet of glass affords almost nothing physically. Every apparent edge, shadow and handle on it is a signifier someone drew — which is why this discipline can be practiced badly with no immediate penalty: remove the signifiers and the affordances technically remain, invisible.

> Affordance คือความสัมพันธ์ ส่วนสิ่งที่เห็นส่วนใหญ่คือ signifier J.J. Gibson นำเสนอคำว่า affordance ในปี 1966 แล้วพัฒนามันต่อในทฤษฎีเชิงนิเวศปี 1979 — สิ่งที่สภาพแวดล้อมมอบให้สิ่งมีชีวิตชนิดหนึ่งโดยเฉพาะ พื้นผิวสูงระดับเก้าอี้เอื้อให้ผู้ใหญ่นั่งได้ แต่ไม่เอื้อให้ทารกนั่ง (affordance) Don Norman นำคำนี้เข้าสู่วงการดีไซน์ในปี 1988 ใน The Psychology of Everyday Things ซึ่งภายหลังเปลี่ยนชื่อเป็น The Design of Everyday Things แล้วใช้เวลายี่สิบห้าปีดูนักออกแบบใช้คำนี้ผิดความหมายไปเป็น "คำใบ้ทางสายตา" — ฉบับปรับปรุงปี 2013 จึงเสนอคำว่า signifier สำหรับร่องรอยที่รับรู้ได้ซึ่งบอกว่าตรงไหนทำอะไรได้ (Norman) ความแตกต่างนี้สำคัญมากบนทัชสกรีน เพราะแผ่นกระจกแทบไม่ afford อะไรในเชิงกายภาพเลย ขอบ เงา และมือจับที่ดูเหมือนมีอยู่บนจอ ล้วนเป็น signifier ที่มีคนวาดขึ้นมา ทั้งนั้น — นี่คือเหตุผลที่ศาสตร์นี้ทำได้แย่โดยไม่มีบทลงโทษทันที ถ้าเอา signifier ออก affordance ในทางเทคนิคก็ยังอยู่ แค่มองไม่เห็น

The human is a component with published specifications. Robert Miller measured the response times at which a conversation with a machine stops feeling like one, and Jakob Nielsen restated the same three thresholds decades later: about 0.1 second for an action to feel instantaneous, about one second to keep a person's flow of thought unbroken, about ten seconds before attention leaves altogether (Miller 1968 / Nielsen). Paul Fitts had

already quantified pointing in 1954 — the time to reach a target grows with the logarithm of the ratio between distance and target width (Fitts's law), which is why the corners and edges of a screen are its most valuable real estate: they are targets of effectively infinite width, because the pointer stops there. These are the only constants in this book that describe the user rather than the machine, and unlike Moore's law they have not moved in seventy years.

> มนุษย์คือส่วนประกอบที่มีสเปกตีพิมพ์ไว้ให้ดู Robert Miller วัดเวลาตอบสนองที่บทสนทนากับเครื่องจักรเริ่มไม่รู้สึกเหมือนบทสนทนาอีกต่อไป แล้ว Jakob Nielsen ก็พูดถึงเกณฑ์สามค่าเดียวกันซ้ำอีกครั้งหลายสิบปีให้หลัง — ราว 0.1 วินาทีเพื่อให้การกระทำรู้สึกเกิดขึ้นทันที ราวหนึ่งวินาทีเพื่อรักษาความต่อเนื่องของความคิดคนไว้ไม่ให้ขาด ราวสิบวินาทีก่อนที่ความสนใจจะหลุดไปเลย (Miller 1968 / Nielsen) Paul Fitts วัดค่าการชี้ตำแหน่งไว้ตั้งแต่ปี 1954 แล้ว — เวลาที่ใช้ไปถึงเป้าหมายโตตามลอการิทึมของอัตราส่วนระหว่างระยะทางกับความกว้างของเป้าหมาย (Fitts's law) นี่คือเหตุผลที่มุมและขอบของหน้าจอเป็นทำเลที่มีค่าที่สุด — มันคือเป้าหมายที่มีความกว้างเป็นอนันต์โดยพฤตินัย เพราะตัวชี้หยุดอยู่ตรงนั้นเอง นี่คือค่าคงที่เพียงกลุ่มเดียวในเล่มนี้ที่บรรยายผู้ใช้ ไม่ใช่เครื่องจักร และต่างจากกฎของมัวร์ตรงที่มันไม่เคยขยับเลยตลอดเจ็ดสิบปีที่ผ่านมา

Rendering is one integral you cannot solve, approached from two directions. Rasterization asks, for each triangle, which pixels does it cover — fast, local, and the reason real-time graphics existed at all for thirty years. Ray tracing asks the opposite question, for each pixel, which light arrived here, and follows rays backwards into the scene; Arthur Appel cast the first such rays in 1968 and Turner Whitted made them recursive in 1980, so surfaces could reflect and refract each other (CACM 23(6)). Jim Kajiya then wrote down what both were approximating: the rendering equation of 1986, stating that the light leaving any point is what it emits plus everything it reflects from everywhere else — recursive, with no closed-form solution (SIG-

GRAPH '86). All of photorealistic rendering is sampling that integral cheaply enough. It took until 2018 for the first GPUs with dedicated ray-tracing units to run the honest method at frame rate (NVIDIA Turing).

> การเรนเดอร์ (rendering) คืออินทิกรัลหนึ่งตัวที่แก้แบบปิดไม่ได้ เข้าถึงมันได้จากสองทาง rasterization ถามว่า สำหรับสามเหลี่ยมแต่ละรูป มันคลุมพิกเซล (pixel) ไหนบ้าง — เร็ว เฉพาะที่ และเป็นเหตุผลที่กราฟิกส์เรียลไทม์มีอยู่ได้เลยตลอดสามสิบปี ray tracing ถามคำถามตรงข้ามกัน สำหรับแต่ละพิกเซล แสงไหนมาถึงตรงนี้ แล้วไล่ตามรังสีย้อนกลับเข้าไปในฉาก — Arthur Appel ยิงรังสีแบบนี้เป็นครั้งแรกในปี 1968 และ Turner Whitted ทำให้มันเป็น recursive ได้ในปี 1980 ทำให้พื้นผิวสะท้อนและหักเหกันเองได้ (CACM 23(6)) จากนั้น Jim Kajiya ก็เขียนสิ่งที่ทั้งสองวิธีกำลังประมาณค่าอยู่ลงมาเป็นสูตร — สมการเรนเดอร์ (rendering equation) ปี 1986 ที่บอกว่าแสงที่ออกจากจุดใดๆ ก็คือแสงที่จุดนั้นปล่อยเอง บวกกับทุกอย่างที่มันสะท้อนมาจากที่อื่นทั้งหมด — เป็น recursive และไม่มีคำตอบแบบปิด (SIGGRAPH '86) การเรนเดอร์แบบสมจริงทั้งหมดก็คือการสุ่มตัวอย่างอินทิกรัลตัวนั้นให้ถูกพอ กว่าจะมี GPU ตัวแรกที่มีหน่วยประมวลผล ray tracing เฉพาะทาง มารันวิธีที่ตรงไปตรงมานี้ได้ที่เฟรมเรต (frame rate) จริง ก็ต้องรอถึงปี 2018 (NVIDIA Turing)

Accessibility is the floor, not a ramp bolted on afterwards. The World Health Organization estimates that 1.3 billion people — about one in six — live with significant disability (WHO, 7 March 2023). The web's standard, WCAG 2.2, became a W3C Recommendation on 5 October 2023 and rests on four principles: content must be perceivable, operable, understandable and robust (W3C); its successor remains a working draft (WCAG 3, March 2026). The argument for treating this as foundation rather than compliance chore has a name. Disabled activists in Berkeley took sledgehammers to curbs in the early 1970s, and the first official curb cut followed on Telegraph Avenue in 1972; the ramps turned out to serve parents with strollers, travelers with suitcases and delivery workers with carts — the *curb-cut effect*, named by

Angela Glover Blackwell in 2017 (SSIR). Designing for the constrained case reliably produces a better general case: the rare argument in this book that is both ethical and cheap.

> การเข้าถึงได้ (accessibility) คือชั้นพื้นฐาน ไม่ใช่ทางลาดที่มาติดทีหลัง องค์การอนามัยโลกประมาณว่าคน 1.3 พันล้านคน — ราวหนึ่งในหก — มีความพิการอย่างมีนัยสำคัญ (WHO, 7 March 2023) มาตรฐานของเว็บอย่าง WCAG 2.2 กลายเป็น W3C Recommendation เมื่อวันที่ 5 ตุลาคม 2023 และตั้งอยู่บนหลักการสี่ข้อ — เนื้อหาต้องรับรู้ได้ ใช้งานได้ เข้าใจได้ และแข็งแรงพอ (W3C) ส่วนรุ่นต่อไปยังเป็นแค่ working draft (WCAG 3, March 2026) ข้อโต้แย้งที่บอกว่าควรมองเรื่องนี้เป็นรากฐาน ไม่ใช่งานทำตามกฎเกณฑ์ มีชื่อเรียกของมันเอง นักเคลื่อนไหวเพื่อคนพิการใน Berkeley เอาค้อนปอนด์ทุบขอบถนนในช่วงต้นทศวรรษ 1970 แล้วทางลาดขอบถนนอย่างเป็นทางการจุดแรกก็ตามมาที่ Telegraph Avenue ในปี 1972 ปรากฏว่าทางลาดพวกนี้กลับเป็นประโยชน์กับพ่อแม่ที่เข็นรถเข็นเด็ก นักเดินทางที่ลากกระเป๋า และคนส่งของที่เข็นรถเข็นสัมภาระ — เรียกว่า curb-cut effect ตั้งชื่อโดย Angela Glover Blackwell ในปี 2017 (SSIR) การออกแบบเพื่อกรณีที่ถูกจำกัดเงื่อนไข มักให้ผลลัพธ์ที่ดีกว่าสำหรับกรณีทั่วไปด้วยเสมอ — เป็นข้อโต้แย้งเดียวในเล่มนี้ที่ทั้งถูกต้องเชิงจริยธรรมและมีต้นทุนต่ำในเวลาเดียวกัน

Sutherland received the Turing Award for 1988, Engelbart for 1997, and Ed Catmull and Pat Hanrahan for 2019 (ACM).

> Sutherland ได้รับรางวัล Turing Award ประจำปี 1988 Engelbart ได้รับปี 1997 ส่วน Ed Catmull และ Pat Hanrahan ได้รับปี 2019 (ACM)

Connections · เชื่อมโยง

Below: chapter 04, because every principle here is rationed by hardware — the frame budget, the specialized processors that made both interfaces and rendering possible. Chapter 05 owns the multiplexing that keeps an interface responsive while the machine does everything else, and chapter 09 is where an interface's delays come from once the other mind is remote. Sideways: chapter 07, since usability must survive years of change by many hands, and chapter 11, whose 1975 list of security principles ended with *psy-*

chological acceptability — the one this floor is responsible for and the one security most often sacrifices. Above: nothing. This is the top of the tower, and the reason all the hiding below it was worth doing.

> ชั้นล่าง: บทที่ 04 เพราะหลักการทุกข้อในบทนี้ถูกจำกัดปริมาณด้วยฮาร์ดแวร์ — งบเฟรม (frame budget) และตัวประมวลผลเฉพาะทางที่ทำให้ทั้งอินเทอร์เฟซและการเรนเดอร์เป็นไปได้ บทที่ 05 เป็นเจ้าของการสลับใช้งาน (multiplexing) ที่รักษาให้อินเทอร์เฟซตอบสนองได้ในขณะที่เครื่องทำงานอื่นทั้งหมดไปด้วย และบทที่ 09 คือจุดที่ความล่าช้าของอินเทอร์เฟซมาจาก เมื่อความคิดอีกฝ่ายอยู่ไกลออกไป ด้านข้าง: บทที่ 07 เพราะความใช้ง่าย (usability) ต้องอยู่รอดผ่านการเปลี่ยนแปลงหลายปีโดยมือหลายคน และบทที่ 11 ซึ่งรายการหลักความปลอดภัยปี 1975 จบลงด้วย psychological acceptability — ข้อที่ชั้นนี้รับผิดชอบ และเป็นข้อที่ความปลอดภัยมักสละทิ้งบ่อยที่สุด ด้านบน: ไม่มีอะไรอีกแล้ว นี่คือชั้นบนสุดของหอคอย และเป็นเหตุผลที่การซ่อนทุกอย่างข้างใต้มันคุ้มค่าที่จะทำ

Open questions · คำถามที่ยังเปิดอยู่

If natural language becomes the interface, what happens to direct manipulation — do Shneiderman's three properties survive when the object of interest is no longer continuously visible and actions are no longer obviously reversible? Interfaces that adapt to a person must first model that person; who holds that model, and what else it serves, is a question the field has largely left to other disciplines. And the oldest one, unanswered since 1968: Engelbart wanted augmentation, not automation. Sixty years on, most of what shipped was automation. Was that a technical inevitability or a commercial one?

> ถ้าภาษาธรรมชาติกลายเป็นอินเทอร์เฟซ direct manipulation จะเป็นยังไงต่อ — คุณสมบัติสามข้อของ Shneiderman จะยังอยู่รอดไหม เมื่อวัตถุที่สนใจไม่ได้ปรากฏให้เห็นต่อเนื่องอีกต่อไป และการกระทำก็ไม่ได้ย้อนกลับได้อย่างชัดเจนอีกแล้ว อินเทอร์เฟซที่ปรับตัวเข้าหาคนต้องสร้างแบบจำลองของคนคนนั้นก่อน ใครเป็นคนถือแบบจำลองนั้น และมันรับใช้อะไรอีกบ้าง เป็นคำถามที่วงการนี้ปล่อยให้เป็นเรื่องของสาขาอื่นไปเกือบทั้งหมด และคำถามที่เก่าแก่ที่สุด ซึ่งยังไม่มีคำตอบตั้งแต่ปี

> 1968 — Engelbart ต้องการการเสริม (augmentation) ไม่ใช่การทำให้เป็นอัตโนมัติ (automation) หกสิบปีผ่านไป สิ่งที่ถูกส่งออกมาใช้จริงส่วนใหญ่กลับเป็นการทำให้เป็นอัตโนมัติ นั่นเป็นความหลีกเลี่ยงไม่ได้เชิงเทคนิค หรือเชิงพาณิชย์กันแน่

Go deeper · อ่านต่อ

— Don Norman, *The Design of Everyday Things*, revised and expanded edition, Basic Books, 2013 — the discipline's one indispensable general-reader book; the author's note on the revision is at jnd.org.

> Don Norman, The Design of Everyday Things, revised and expanded edition, Basic Books, 2013 — หนังสือเล่มเดียวของศาสตร์นี้ที่ผู้อ่านทั่วไปขาดไม่ได้ บันทึกของผู้เขียนเกี่ยวกับฉบับปรับปรุงอยู่ที่ jnd.org

— Matt Pharr, Wenzel Jakob & Greg Humphreys, *Physically Based Rendering: From Theory to Implementation* — the full text of the 3rd and 4th editions, free, at pbr-book.org.

> Matt Pharr, Wenzel Jakob & Greg Humphreys, Physically Based Rendering: From Theory to Implementation — อ่านฉบับเต็มของรุ่นที่ 3 และ 4 ได้ฟรีที่ pbr-book.org

— Douglas Engelbart, "Augmenting Human Intellect: A Conceptual Framework," SRI Summary Report AFOSR-3223, 1962 — dougengelbart.org.

> Douglas Engelbart, "Augmenting Human Intellect: A Conceptual Framework," SRI Summary Report AFOSR-3223, 1962 — dougengelbart.org

CHAPTER 14

The Frontier

พรมแดน

This chapter is not a floor. It is the scaffolding on the roof and the cracks in the foundation — the places where the concrete is still wet, and where a reader who wants to build something has room to stand.

บทนี้ไม่ใช่ชั้น มันคือนั่งร้านบนหลังคาและรอยแตกที่ฐานราก — จุดที่ปูนยังไม่แห้ง และเป็นที่ที่ผู้อ่านที่อยากสร้างอะไรบางอย่างยังมีที่ให้ยืน

Which walls of the tower are still wet concrete?

กำแพงส่วนไหนของหอคอยที่ยังเป็นปูนเปียก

In May 1981, at MIT's Endicott House, Richard Feynman raised a problem computing had no answer for. Physicists wanted to simulate quantum systems, and the cost on an ordinary computer grows exponentially with the number of particles: a few dozen interacting particles exhaust any machine that will ever be built. His conclusion, published the following year, was not that the simulation should be made cleverer but that the computer was the wrong kind of object — nature is not classical, so a machine that simulates nature should be built out of quantum mechanics rather than in spite of it (*International Journal of Theoretical Physics* 21, 1982).

> เดือนพฤษภาคม 1981 ที่ Endicott House ของ MIT Richard Feynman ยกปัญหาที่วงการคอมพิวเตอร์ยังตอบไม่ได้ขึ้นมา นักฟิสิกส์อยากจำลอง (simulate) ระบบควอนตัม แต่ต้นทุนบนคอมพิวเตอร์ทั่วไปโตแบบเอกซ์โพเนนเชียลตามจำนวนอนุภาค — อนุภาคที่มีปฏิสัมพันธ์กันแค่ไม่กี่สิบตัวก็ใช้เครื่องที่จะสร้างขึ้นได้ทุกเครื่องจนหมด ข้อสรุปของเขาซึ่งตีพิมพ์ปีถัดมาไม่ใช่ว่าต้องทำการจำลองให้ฉลาดขึ้น แต่คือ

> คอมพิวเตอร์เป็นวัตถุผิดประเภทตั้งแต่ต้น — ธรรมชาติไม่ใช่ระบบเชิงคลาสสิก (classical) ฉะนั้นเครื่องที่จำลองธรรมชาติต้องสร้างขึ้นจากกลศาสตร์ควอนตัม ไม่ใช่สร้างขึ้นทั้งที่ขัดกับมัน (International Journal of Theoretical Physics 21, 1982)

Forty-five years on it is the most over-promised item on this list. Start with what it buys.

> สี่สิบห้าปีให้หลัง มันคือรายการที่ถูกโฆษณาเกินจริงที่สุดในลิสต์นี้ เริ่มจากสิ่งที่มันแลกมาได้จริงก่อน

Quantum computers are specialists, and the specialism is narrow. Peter Shor showed in 1994 that one could factor integers and compute discrete logarithms efficiently — an exponential speedup, and the reason chapter 11 has a deadline (Shor). Lov Grover showed in 1996 that unstructured search gets faster too, but only quadratically: $\sqrt{N}$ queries where a classical machine needs N, and that bound is provably the best possible. The difference between exponential and quadratic is the whole story. A quadratic speedup can be erased by a decade of ordinary hardware progress; an exponential one cannot be erased at all. And the class of problems most people assume is the target — NP-complete problems like scheduling, routing and packing — is not believed to fall. There is no proof either way, but there is an oracle relative to which they do not, and no candidate algorithm (Aaronson). A quantum computer is not a faster computer. It is a different instrument, good at simulating quantum systems and at breaking two specific pieces of mathematics we happened to build civilization's trust on.

> คอมพิวเตอร์เชิงควอนตัม (quantum computer) เป็นผู้เชี่ยวชาญเฉพาะทาง และความเชี่ยวชาญนั้นก็แคบมาก Peter Shor แสดงในปี 1994 ว่าแยกตัวประกอบจำนวนเต็มและคำนวณ discrete logarithm ได้อย่างมีประสิทธิภาพ — เป็นการเร่งความเร็ว (speedup) แบบเอกซ์โพเนนเชียล และนี่คือเหตุผลที่บทที่ 11 มีเส้นตาย (Shor) Lov Grover แสดงในปี 1996 ว่าการค้นหาแบบไม่มีโครงสร้างก็เร็วขึ้นได้เช่นกัน แต่เป็นแค่กำลังสอง (quadratically) เท่านั้น — ใช้ $\sqrt{N}$ ครั้งในการค้นหา แทนที่จะต้องใช้ N ครั้งแบบเครื่อง classical และขอบเขตนั้นพิสูจน์แล้วว่าดีที่สุดเท่าที่เป็นไปได้ ความต่างระหว่างเอกซ์โพเนนเชียลกับกำลังสองคือแก่นทั้งหมดของเรื่องนี้ การ

> เร่งความเร็วแบบกำลังสองถูกลบล้างได้ด้วยความก้าวหน้าของฮาร์ดแวร์ธรรมดาในสิบปี ส่วนการเร่งความเร็วแบบเอกซ์โพเนนเชียลลบล้างไม่ได้เลย และกลุ่มปัญหาที่คนส่วนใหญ่คิดว่าเป็นเป้าหมาย — ปัญหา NP-complete อย่างการจัดตาราง การจัดเส้นทาง และการจัดวาง — เชื่อกันว่าจะไม่ล้มลง ไม่มีบทพิสูจน์ไปทางไหนแน่ชัด แต่มี oracle ที่เทียบกับมันแล้วปัญหาเหล่านี้ไม่ล้ม และยังไม่มีอัลกอริทึมตัวไหนที่เป็นตัวเต็ง (Aaronson) คอมพิวเตอร์เชิงควอนตัมไม่ใช่คอมพิวเตอร์ที่เร็วกว่า มันคือเครื่องมือคนละชนิด เก่งเรื่องจำลองระบบควอนตัมและเรื่องทลายคณิตศาสตร์สองชิ้นเฉพาะที่บังเอิญเป็นฐานที่อารยธรรมฝากความเชื่อใจไว้

The engineering problem is not qubits; it is error. A qubit holds its state for microseconds before the environment ruins it, so useful machines must correct errors faster than they accumulate — spreading one logical qubit across many physical ones. The threshold theorem says this works only if the physical error rate falls below a critical value; above it, adding qubits makes things worse. Google's Willow processor demonstrated the crossing in a result published online in December 2024. Each increase of the surface-code distance by two suppressed the logical error rate by a factor of 2.14 ± 0.02, and a distance-7 code on 101 qubits reached 0.143% ± 0.003% error per cycle — below threshold, meaning bigger now means better (*Nature* 638: 920–926). That does not mean a useful machine exists. The best current public estimate for breaking RSA-2048 is under a million noisy physical qubits running for about a week at a 0.1% gate error rate — an architectural calculation on paper, not a device anyone has built (Gidney, 2025). Read every quantum announcement against that gap.

> ปัญหาทางวิศวกรรมไม่ใช่คิวบิต (qubit) แต่คือความผิดพลาด คิวบิตหนึ่งตัวคงสถานะ (state) ของมันไว้ได้แค่ไม่กี่ไมโครวินาทีก่อนที่สิ่งแวดล้อมจะทำลายมัน เครื่องที่ใช้งานได้จริงจึงต้องแก้ความผิดพลาดให้เร็วกว่าที่มันสะสม — ด้วยการกระจายคิวบิตเชิงตรรกะหนึ่งตัวไปไว้บนคิวบิตทางกายภาพหลายตัว ทฤษฎีบทเกณฑ์ (threshold theorem) บอกว่าวิธีนี้ใช้ได้ก็ต่อเมื่ออัตราความผิดพลาดทางกายภาพต่ำกว่าค่าวิกฤตค่าหนึ่ง สูงกว่านั้นการเพิ่มคิวบิตจะยิ่งทำให้แย่ลง ตัวประมวลผล Willow ของ Google แสดงจุดข้ามผ่านนี้ไว้ในผลลัพธ์ที่เผยแพร่ออนไลน์เมื่อเดือนธันวาคม 2024 ทุกครั้งที่เพิ่มระยะของ surface code ขึ้นสองหน่วย อัตรา

> ความผิดพลาดเชิงตรรกะลดลงด้วยตัวประกอบ 2.14 ± 0.02 และรหัสระยะ 7 บนคิวบิต 101 ตัวทำได้ถึง 0.143% ± 0.003% ความผิดพลาดต่อรอบ — ต่ำกว่าเกณฑ์หมายความว่ายิ่งใหญ่ขึ้นตอนนี้ยิ่งดีขึ้น (Nature 638: 920–926) นั่นไม่ได้แปลว่ามีเครื่องที่ใช้งานได้จริงอยู่แล้ว การประมาณการณ์สาธารณะที่ดีที่สุดตอนนี้สำหรับการทลาย RSA-2048 คือคิวบิตทางกายภาพที่มีสัญญาณรบกวนต่ำกว่า 1 million ตัว รันต่อเนื่องราวหนึ่งสัปดาห์ที่อัตราความผิดพลาดของเกต 0.1% — เป็นการคำนวณเชิงสถาปัตยกรรมบนกระดาษ ไม่ใช่อุปกรณ์ที่ใครสร้างขึ้นจริง (Gidney, 2025) อ่านทุกประกาศเรื่องควอนตัมโดยเทียบกับช่องว่างนี้ไว้เสมอ

Simulation became science's third instrument, and its leaks are epistemic. In June 2005 the US President's Information Technology Advisory Committee put it on the record: "Computational science now constitutes what many call the third pillar of the scientific enterprise, a peer alongside theory and physical experimentation" (PITAC). Climate, protein folding, drug binding, galaxy formation, crash safety — these are now studied primarily inside machines. This is an abstraction like any other in this book, with one uncomfortable difference. When a normal abstraction leaks, something breaks and you descend the stairs to find out why. When a simulation leaks, the output is still a plausible number, and nothing tells you from the inside whether a disagreement with the world is the world's fault or the model's. The discipline that grew up to handle this — verification (did we solve the equations right?) and validation (did we write the right equations?) — is the most transferable idea in this chapter and the least famous.

> การจำลอง (simulation) กลายเป็นเครื่องมือชิ้นที่สามของวิทยาศาสตร์ และจุดที่มันรั่วก็เป็นเรื่องของญาณวิทยา (epistemic) เดือนมิถุนายน 2005 US President's Information Technology Advisory Committee บันทึกไว้เป็นทางการว่า "วิทยาศาสตร์เชิงคำนวณตอนนี้เป็นสิ่งที่หลายคนเรียกว่าเสาหลักที่สามของงานวิทยาศาสตร์ เทียบเท่ากับทฤษฎีและการทดลองทางกายภาพ" (PITAC) ภูมิอากาศ การพับตัวของโปรตีน การจับตัวของยา การก่อตัวของกาแล็กซี ความปลอดภัยเมื่อชนกัน — เรื่องพวกนี้ตอนนี้ถูกศึกษาหลักๆ อยู่ในเครื่องคอมพิวเตอร์ นี่คือนามธรรม (abstraction) แบบเดียวกับทุกตัวในเล่มนี้ แต่มีข้อแตกต่างหนึ่งที่ไม่สบายใจ เวลา abstraction ปกติรั่ว จะมีบางอย่างพังแล้วเราลงบันไดไปดูว่าทำไม แต่เวลาการ

> จำลองรั่ว ผลลัพธ์ที่ออกมาก็ยังเป็นตัวเลขที่ดูสมเหตุสมผลอยู่ดี และไม่มีอะไรบอกจากข้างในว่าความไม่ตรงกับโลกจริงนั้นเป็นความผิดของโลกหรือของโมเดล ศาสตร์ที่เกิดขึ้นมาเพื่อรับมือเรื่องนี้ — verification (เราแก้สมการถูกไหม) กับ validation (เราเขียนสมการที่ถูกต้องหรือเปล่า) — คือไอเดียที่เอาไปใช้ที่อื่นได้มากที่สุดในบทนี้ และเป็นไอเดียที่ดังน้อยที่สุด

The binding constraint is now energy. For thirty years, Dennard scaling meant smaller transistors used proportionally less power, so every generation was free. It broke down around 2005–2007, when leakage current made heat the limit, and chapter 04 has lived with the consequences ever since. Efficiency still improves, but slower: the doubling time of computations per joule has stretched from roughly 1.6 years before 2000 to nearer 2.5 years since (Koomey's law). There is a floor underneath all of it. Rolf Landauer showed in 1961 that erasing one bit must dissipate at least $kT \ln 2$ of energy — about 2.8×10^{-21} joules at room temperature — the point in this book where information stops being a metaphor and becomes thermodynamics, the other half of chapter 03's story. Real processors sit between a million and a billion times above that floor, depending how you count, so the physics is not the problem yet. The bill is. The International Energy Agency reported on 16 April 2026 that data-center electricity demand grew 17% during 2025, with AI-focused facilities climbing faster still, and projects total data-center consumption roughly doubling between 2025 and 2030 (IEA). For the first time in the tower's history, the question limiting what gets computed is not what is computable, or what is fast, but what can be powered.

> ข้อจำกัดตัวคุมตอนนี้คือพลังงาน สามสิบปีที่ผ่านมา Dennard scaling แปลว่าทรานซิสเตอร์ (transistor) ที่เล็กลงใช้พลังงานน้อยลงตามสัดส่วน ฉะนั้นทุกรุ่นถัดไปได้มาฟรีๆ มันพังลงราวปี 2005–2007 ตอนที่กระแสรั่วไหล (leakage current) ทำให้ความร้อนกลายเป็นข้อจำกัด และบทที่ 04 ก็อยู่กับผลที่ตามมานี้เรื่อยมา ประสิทธิภาพยังดีขึ้นอยู่ แต่ช้าลง — เวลาที่จำนวนการคำนวณต่อจูลเพิ่มเป็นสองเท่ายืดจากราว 1.6 ปีก่อนปี 2000 มาเป็นใกล้ 2.5 ปีหลังจากนั้น (Koomey's law) มีพื้นต่ำสุดรองรับอยู่ใต้ทั้งหมดนี้ Rolf Landauer แสดงในปี 1961 ว่าการลบข้อมูลหนึ่งบิต

> ต้องระบายพลังงานอย่างน้อย kT ln 2 — ราว 2.8 × 10^{-21} จูลที่อุณหภูมิห้อง — จุดในเล่มนี้ที่สารสนเทศ (information) เลิกเป็นอุปมาแล้วกลายเป็นเทอร์โมไดนามิกส์ อีกครึ่งหนึ่งของเรื่องราวในบทที่ 03 โปรเซสเซอร์จริงอยู่สูงกว่าพื้นนั้นระหว่าง 1 million ถึง 1 billion เท่า แล้วแต่วิธีนับ ฉะนั้นฟิสิกส์ยังไม่ใช่ปัญหา บิลค่าไฟต่างหากที่เป็น International Energy Agency รายงานเมื่อวันที่ 16 เมษายน 2026 ว่าความต้องการไฟฟ้าของ data center โตขึ้น 17% ในปี 2025 โดยศูนย์ที่เน้น AI โตเร็วกว่านั้นอีก และคาดการณ์ว่าการใช้ไฟฟ้ารวมของ data center จะเพิ่มเป็นราวสองเท่าระหว่างปี 2025 ถึง 2030 (IEA) เป็นครั้งแรกในประวัติศาสตร์ของหอคอย ที่คำถามซึ่งจำกัดว่าจะคำนวณอะไรได้ ไม่ใช่ว่าอะไรคำนวณได้ หรืออะไรเร็ว แต่คือมีไฟเลี้ยงพอไหม

Several walls are unfinished, and the invitations are open. Whether P equals NP carries a million-dollar prize offered by the Clay Mathematics Institute on 24 May 2000, and remains unclaimed (Clay) — not a puzzle for specialists but the question of whether checking an answer is fundamentally easier than finding one, which is to say whether creativity is real or merely an unoptimized search. Whether one-way functions exist is chapter 11's foundation and equally unproven. Neuromorphic computing — machines organized like nervous systems rather than clocked pipelines — has produced Intel's Hala Point, 1.15 billion neurons at Sandia National Laboratories in April 2024 (Intel), with no confirmed larger system since. Reversible computing, which attacks the Landauer floor by not erasing bits in the first place, has moved from theory to silicon in a 2025 prototype recovering roughly half its switching energy (IEEE Spectrum). And room-temperature superconductivity, which would change every number above, has not arrived: the 2023 LK-99 signal was traced to a copper-sulfide impurity undergoing a structural phase transition, and no verified ambient-pressure room-temperature superconductor exists (phys.org).

> กำแพงหลายจุดยังสร้างไม่เสร็จ และคำเชิญยังเปิดอยู่ ว่า P เท่ากับ NP หรือไม่นั้นมีเงินรางวัล 1 million ดอลลาร์จาก Clay Mathematics Institute ที่ประกาศไว้เมื่อวันที่ 24 พฤษภาคม 2000 และยังไม่มีใครรับไปได้ (Clay) — ไม่ใช่ปริศนาสำหรับผู้เชี่ยวชาญเฉพาะทาง แต่เป็นคำถามว่าการตรวจคำตอบง่ายกว่าการหาคำตอบโดยพื้นฐานหรือเปล่า ซึ่งก็คือคำถามว่าความคิดสร้างสรรค์มีจริงหรือเป็นแค่การค้นหาที่

> ยังไม่ถูกปรับให้เหมาะ ว่าฟังก์ชันทางเดียว (one-way function) มีอยู่จริงหรือไม่คือฐานรากของบทที่ 11 และก็ยังพิสูจน์ไม่ได้เหมือนกัน การคำนวณเลียนแบบระบบประสาท (neuromorphic computing) — เครื่องที่จัดวางตัวเหมือนระบบประสาทแทนที่จะเป็น pipeline ที่เดินตามสัญญาณนาฬิกา — ผลิต Hala Point ของ Intel ออกมาแล้ว นิวรอน 1.15 billion ตัวที่ Sandia National Laboratories เมื่อเดือนเมษายน 2024 (Intel) และยังไม่มีระบบที่ใหญ่กว่านี้ที่ยืนยันได้ตั้งแต่นั้นมา การคำนวณผันกลับได้ (reversible computing) ซึ่งโจมตีพื้นของ Landauer ด้วยการไม่ลบบิตตั้งแต่แรก ได้ย้ายจากทฤษฎีมาสู่ซิลิคอนแล้วในต้นแบบปี 2025 ที่เอาพลังงานจากการสวิตช์กลับคืนมาได้ราวครึ่งหนึ่ง (IEEE Spectrum) และตัวนำยิ่งยวดที่อุณหภูมิห้อง (room-temperature superconductivity) ซึ่งถ้าเกิดขึ้นจริงจะเปลี่ยนตัวเลขทุกตัวข้างบนนี้ ยังไม่มาถึง — สัญญาณ LK-99 เมื่อปี 2023 สืบไปเจอว่าเป็นสิ่งเจือปนทองแดง-กำมะถัน (copper-sulfide) ที่กำลังเปลี่ยนเฟสเชิงโครงสร้าง และยังไม่มีตัวนำยิ่งยวดที่อุณหภูมิห้องและความดันบรรยากาศตัวไหนที่ยืนยันได้จริง (phys.org)

None of these are conclusions. They are the parts of the tower where the plans are still argued over, which is the only part of any building where a newcomer's hands are genuinely needed. The floors below took roughly eighty years to pour. Nothing about them suggests the pouring is finished.

> ไม่มีข้อไหนในนี้เป็นข้อสรุป มันคือส่วนของหอคอยที่แบบก่อสร้างยังถูกถกเถียงกันอยู่ ซึ่งเป็นส่วนเดียวของอาคารใดๆ ที่มือของคนใหม่จำเป็นต้องใช้จริงๆ ชั้นด้านล่างใช้เวลาราวแปดสิบปีในการเทปูน ไม่มีอะไรในนั้นบอกว่าการเทปูนเสร็จแล้ว

Connections · เชื่อมโยง

Below: chapter 01 owns P vs NP as a question about what is computable at what cost; this chapter only reports that it is still open. Chapter 03 supplies the bit, which Landauer's principle reveals to have a price in joules. Chapter 04 is where the energy wall is felt first, in the turn to specialized hardware.

Chapter 11 is where quantum computing stops being interesting and becomes urgent, and chapter 12 is the largest current consumer of everything this chapter says is scarce. Above: nothing yet — that is what a frontier is.

> ชั้นล่าง: บทที่ 01 เป็นเจ้าของคำถาม P vs NP ในฐานะคำถามว่าอะไรคำนวณได้ที่ต้นทุนเท่าไร บทนี้แค่รายงานว่าคำถามนั้นยังเปิดอยู่ บทที่ 03 ให้บิตมา ซึ่งหลักการของ Landauer เผยให้เห็นว่ามีราคาเป็นจูล บทที่ 04 คือจุดที่กำแพงพลังงาน (energy wall) ถูกรู้สึกได้ก่อนใคร ตอนหันไปหาฮาร์ดแวร์เฉพาะทาง บทที่ 11 คือจุดที่การคำนวณเชิงควอนตัม (quantum computing) เลิกเป็นเรื่องน่าสนใจแล้วกลายเป็นเรื่องเร่งด่วน และบทที่ 12 คือผู้บริโภครายใหญ่ที่สุดตอนนี้ของทุกอย่างที่บทนี้บอกว่าขาดแคลน ชั้นบน: ยังไม่มีอะไรเลย — นั่นแหละคือความหมายของพรมแดน

Open questions · คำถามที่ยังเปิดอยู่

Will a cryptographically relevant quantum computer arrive before the world finishes migrating away from the mathematics it would break, and how would anyone know in advance? If simulation is the third pillar of science, what is the equivalent of peer review for a model nobody can independently rebuild? Can the energy curve of computation bend without a physical breakthrough, or does the industry's growth now depend on one? And, quietly the largest: is P vs NP hard because we lack the technique, or because the techniques we have are provably incapable of settling it?

> คอมพิวเตอร์เชิงควอนตัมที่มีนัยสำคัญเชิงการเข้ารหัส (cryptographically relevant) จะมาถึงก่อนที่โลกจะย้ายออกจากคณิตศาสตร์ที่มันจะทลายได้เสร็จหรือเปล่า แล้วใครจะรู้ล่วงหน้าได้ยังไง ถ้าการจำลองเป็นเสาหลักที่สามของวิทยาศาสตร์ อะไรคือสิ่งที่เทียบเท่า peer review สำหรับโมเดลที่ไม่มีใครสร้างซ้ำได้เองอย่างอิสระ เส้นโค้งพลังงานของการคำนวณจะโค้งลงได้โดยไม่ต้องมีความก้าวหน้าทางฟิสิกส์ครั้งใหญ่ไหม หรือว่าการเติบโตของอุตสาหกรรมตอนนี้ต้องพึ่งความก้าวหน้านั้นแล้ว และคำถามที่ใหญ่ที่สุดแบบเงียบๆ คือ P vs NP ยากเพราะเรายังขาดเทคนิค หรือเพราะเทคนิคที่เรามีพิสูจน์แล้วว่าไม่มีทางตัดสินมันได้

Go deeper · อ่านต่อ

- Michael Nielsen, *Quantum Computing for the Very Curious*, 2019 — a free, unusually honest introduction that assumes no physics, at quantum.country/qcvc.

> Michael Nielsen, Quantum Computing for the Very Curious, 2019 — บทนำฟรีที่ตรงไปตรงมาผิดปกติ ไม่ต้องมีพื้นฐานฟิสิกส์มาก่อน ที่ quantum.country/qcvc

- Clay Mathematics Institute, *The Millennium Prize Problems* — the standing list, including P vs NP, at claymath.org.

> Clay Mathematics Institute, The Millennium Prize Problems — ลิสต์ที่ยังยืนอยู่ รวม P vs NP ด้วย ที่ claymath.org

- Rajeev Acharya et al., "Quantum error correction below the surface code threshold," *Nature* 638 (2024): 920–926 — nature.com.

> Rajeev Acharya et al., "Quantum error correction below the surface code threshold," Nature 638 (2024): 920–926 — nature.com

CLOSING

Closing — The Tower, Seen Whole

ปิดเล่ม — หอคอยเมื่อมองทั้งหลัง

This chapter is the roof — the place to stand when the climbing is done, and the map handed back on the way out.

บทนี้คือหลังคา — ที่ให้ยืนเมื่อปีนเสร็จแล้ว และแผนที่ที่ส่งคืนตอนเดินออกไป

What does the whole tower look like — and how does one person climb it?

หอคอยทั้งหลังหน้าตาเป็นยังไง — แล้วคนคนหนึ่งจะปีนมันได้อย่างไร

Press send.

> กดส่ง

A message to a friend on another continent: one tap of a thumb. The interface that took the tap is a sheet of glass whose every apparent edge and button was drawn there by hand, a signifier standing in for an affordance the glass does not have (13). The app that acted on it was written in a notation built for people to read, then lowered — stage by exact stage, each one easy because the one before made it easy — to instructions no person will ever see (06). It ran as one process among hundreds on a machine that lies to each of them, carefully, about how many machines there are (05), and the instructions executed on billions of switches that do not know what a number is, arranged so that arithmetic appears in the arrangement itself (04). The message became bits — units of surprise, priced by a theory that refuses on principle to ask what they mean (03) — and was cut into packets that found their own separate ways across a network nobody owns, carried by senders who slow themselves down voluntarily so that strangers can also talk (09). It traveled encrypted, under mathematics that lets two people who have never met agree on a secret in a room full of eavesdroppers (11). At the

far end it landed in a database that wrote its intention to a log before touching the data, so that a power cut between any two instructions could not make it lie (08), on replicas that cannot agree what time it is and have protocols for agreeing on things anyway (10). A model grown from data, its skill written down by nobody, decided the message was worth showing (12). And under the whole descent the bedrock held: everything that happened was computable, cost what the mathematics of cost says it must (02), and stayed inside the fence drawn in 1936, before any of these machines existed (01).

> ข้อความถึงเพื่อนอีกทวีปหนึ่ง: แตะนิ้วโป้งครั้งเดียว อินเทอร์เฟซที่รับการแตะนั้นคือแผ่นกระจกที่ทุกขอบและปุ่มที่เห็นถูกวาดขึ้นมาด้วยมือ เป็นตัวบ่งชี้ (signifier) ที่ยืนแทน affordance ที่กระจกไม่มีจริง (13) แอปที่ทำงานตามการแตะนั้นเขียนด้วยสัญกรณ์ที่สร้างมาให้คนอ่านได้ แล้วถูกลดขั้นลงมา — ทีละขั้นอย่างแม่นยำ แต่ละขั้นทำได้ง่ายเพราะขั้นก่อนหน้าทำให้มันง่าย — จนกลายเป็นคำสั่งที่ไม่มีมนุษย์คนไหนจะเห็นเลย (06) มันรันเป็นหนึ่งในโพรเซส (process) หลายร้อยตัวบนเครื่องที่โกหกแต่ละโพรเซสอย่างพิถีพิถันว่ามีเครื่องอยู่กี่เครื่อง (05) และคำสั่งเหล่านั้นถูกประมวลผลบนสวิตช์นับพันล้านตัวที่ไม่รู้ด้วยซ้ำว่าตัวเลขคืออะไร จัดวางกันจนเลขคณิตปรากฏขึ้นจากการจัดวางนั้นเอง (04) ข้อความกลายเป็นบิต — หน่วยของความประหลาดใจ ตีราคาโดยทฤษฎีที่ปฏิเสธโดยหลักการที่จะถามว่ามันหมายความว่าอะไร (03) — แล้วถูกตัดเป็นแพ็กเก็ต (packet) ที่หาทางของตัวเองแยกกันข้ามเครือข่ายที่ไม่มีใครเป็นเจ้าของ ขนส่งโดยผู้ส่งที่ยอมชะลอตัวเองโดยสมัครใจ เพื่อให้คนแปลกหน้าคนอื่นได้พูดด้วย (09) มันเดินทางแบบเข้ารหัส ภายใต้คณิตศาสตร์ที่ทำให้คนสองคนที่ไม่เคยเจอกันตกลงความลับกันได้ในห้องที่เต็มไปด้วยคนดักฟัง (11) ปลายทางมันไปตกที่ฐานข้อมูลที่เขียนเจตนาของมันลงบันทึก (log) ก่อนแตะข้อมูลจริง เพื่อให้ไฟดับระหว่างคำสั่งสองคำสั่งไม่มีทางทำให้มันโกหกได้ (08) บนสำเนา (replica) ที่ตกลงกันไม่ได้ด้วยซ้ำว่าตอนนี้กี่โมง แต่ก็มีโพรโทคอลไว้ตกลงเรื่องต่างๆ กันอยู่ดี (10) โมเดลที่งอกขึ้นจากข้อมูล ทักษะของมันไม่มีใครเขียนลงไปเอง เป็นคนตัดสินว่าข้อความนี้ควรค่าแก่การแสดงให้เห็น (12) และตลอดการลงมาทั้งหมด ฐานรากยังคงอยู่: ทุกอย่างที่เกิดขึ้นคำนวณได้ (computable) เสียต้นทุนเท่าที่คณิตศาสตร์ว่าด้วยต้นทุนบอกว่าต้องเสีย (02) และอยู่ในรั้วที่ขีดไว้ตั้งแต่ปี 1936 ก่อนเครื่องพวกนี้แม้แต่เครื่องเดียวจะถือกำเนิดขึ้น (01)

One tap; every floor. And no one who touched the message on its way — no author of any layer, however expert — holds all of it in mind at once. Chapter 00 said that this is not a failure of education but the design. Fifteen floors of descent and ascent later, the sentence should read differently: it is the *achievement*. The tower stands because each floor is a real language, precise on its own terms, and because the people on each floor could afford to take the floors below on trust.

> แตะครั้งเดียว ทุกชั้นทำงาน และไม่มีใครที่แตะข้อความนี้ระหว่างทาง — ไม่มีผู้เขียนชั้นไหนเลย ต่อให้เชี่ยวชาญแค่ไหน — ที่ถือทุกอย่างไว้ในหัวพร้อมกันได้ทั้งหมด บทที่ 00 บอกไว้ว่านี่ไม่ใช่ความล้มเหลวของการศึกษา แต่คือการออกแบบ สิบห้าชั้นของการลงและขึ้นผ่านไปแล้ว ประโยคนี้ควรอ่านต่างออกไป: มันคือ ความสำเร็จ หอคอยยืนอยู่ได้เพราะแต่ละชั้นเป็นภาษาจริง แม่นยำในเงื่อนไขของตัวเอง และเพราะคนบนแต่ละชั้นวางใจชั้นล่างได้โดยไม่ต้องตรวจสอบเอง

Seen from the roof, the whole building repeats three refrains. Each floor hides machinery and offers a surface simpler than what it conceals — that is the one trick, performed fifteen ways. Every surface leaks, and expertise in computing is mostly a nose for which floor a leak is coming from — the slow loop that is really a memory problem (04), the "saved" file that is really a directory-entry problem (05), the hijacked route that is really a trust problem (09). And two of the shafts running through the building are not floors at all but stairwells: the adversary (11) and the learner (12) enter on every level, which is why no floor can be finished without them. The roof itself is unfinished — wet concrete, open problems, an energy bill newly due (14) — and the bedrock is still being argued over (01). A building like that is not a monument. It is a construction site that happens to be inhabited.

> มองจากหลังคาลงมา อาคารทั้งหลังพูดซ้ำสามท่อนหลัก แต่ละชั้นซ่อนกลไกไว้ และยื่นผิว (surface) ที่ง่ายกว่าสิ่งที่มันปิดบัง — นั่นคือกลเดียวที่เล่นซ้ำสิบห้าแบบ ทุกผิวรั่วเสมอ และความเชี่ยวชาญในวงการคอมพิวเตอร์ส่วนใหญ่คือจมูกที่ดมออกว่ารอยรั่วมาจากชั้นไหน — ลูปที่ช้าซึ่งจริงๆ แล้วเป็นปัญหาหน่วยความจำ (04) ไฟล์ที่ "เซฟแล้ว" ซึ่งจริงๆ เป็นปัญหา directory entry (05) เส้นทางที่ถูกจี้ซึ่งจริงๆ เป็นปัญหาความเชื่อใจ (trust) (09) และช่องสองช่องที่ทะลุผ่านอาคารทั้งหลังไม่ใช่ชั้นเลยแต่

เป็นบันได: ฝ่ายตรงข้าม (11) กับผู้เรียน (12) เข้ามาได้ทุกชั้น ซึ่งเป็นเหตุผลที่ไม่มีชั้นไหนสร้างเสร็จสมบูรณ์ได้โดยไม่มีทั้งสองอย่างนี้ หลังคาเองยังสร้างไม่เสร็จ — ปูนยังเปียก ปัญหาที่ยังเปิดอยู่ บิลค่าพลังงานที่เพิ่งครบกำหนด (14) — และฐานรากก็ยังถูกถกเถียงกันอยู่ (01) อาคารแบบนี้ไม่ใช่อนุสาวรีย์ มันคือไซต์ก่อสร้างที่บังเอิญมีคนอาศัยอยู่

The thirty-day route · เส้นทาง 30 วัน

The route below assumes what every chapter assumed: intelligence, and no coursework. Two days per floor. On the first day, reread the chapter with its primary source open beside it — the sources were chosen because they are readable, and reading Turing or Shannon in the original is the fastest cure for the belief that foundations are beyond you. On the second day, put a hand on the real thing. Every resource here is canonical in its field and free; the links were verified as this book went to press. Two hours a day for a month will not make anyone an expert. It does something better: it makes every floor a place you have stood, so that nothing in computing is a rumor to you anymore.

เส้นทาง (route) ข้างล่างนี้ตั้งอยู่บนสมมติฐานเดียวกับทุกบทในเล่มนี้ คือมีสติปัญญาและไม่ต้องมีพื้นเรียนมาก่อน สองวันต่อหนึ่งชั้น วันแรกอ่านบทนั้นซ้ำโดยเปิดแหล่งข้อมูลปฐมภูมิ (primary source) ไว้ข้างๆ — แหล่งข้อมูลเหล่านี้ถูกเลือกเพราะอ่านได้จริง และการอ่าน Turing หรือ Shannon จากต้นฉบับคือทางรักษาที่เร็วที่สุดสำหรับความเชื่อที่ว่ารากฐานพวกนี้เกินความสามารถของตัวเอง วันที่สองให้ลงมือแตะของจริง แหล่งเรียน (resource) ทุกชิ้นที่นี่เป็นของมาตรฐานในสาขานั้นและฟรีทั้งหมด ลิงก์ถูกตรวจสอบแล้วตอนที่เล่มนี้เข้าโรงพิมพ์ วันละสองชั่วโมงเป็นเวลาหนึ่งเดือนไม่ทำให้ใครกลายเป็นผู้เชี่ยวชาญ แต่มันทำสิ่งที่ดีกว่านั้น — มันทำให้ทุกชั้นกลายเป็นที่ที่เราเคยยืนมาแล้วจริงๆ จนไม่มีอะไรในวงการคอมพิวเตอร์เป็นแค่คำเล่าลือสำหรับเราอีกต่อไป

Days	Floor	The two days' work
1–2	00 The Map	Reread ch. 00. Then start the lower tower for a general reader: Petzold's *Code* companion (codehiddenlanguage.com) and the Nand2Tetris materials (nand2tetris.org) you will use on days 9–10.
3–4	01 Theory of Computation	Turing 1936, §9, with ch. 01 as guide (PDF); then the Church–Turing entry in the Stanford Encyclopedia (plato.stanford.edu).
5–6	02 Algorithms & Data Structures	MIT OCW 6.006, first lectures (ocw.mit.edu); implement one divide-and-conquer algorithm by hand and time it against the naive version.
7–8	03 Information & Coding	Shannon 1948, Part I (PDF); then MacKay's opening chapters (inference.org.uk).
9–10	04 Computer Architecture	Nand2Tetris projects 1–3: from a single gate to an ALU. Arithmetic will appear in the arrangement, under your own hands.
11–12	05 Operating Systems	OSTEP, the virtualization dialogues and intro chapters (pages.cs.wisc.edu/~remzi/OSTEP); list the processes on your own machine and find what each one is lying about.
13–14	06 Languages & Compilers	*Crafting Interpreters* through the scanner and parser (craftinginterpreters.com) — a compiler's tower, built floor by floor in your editor.
15–16	07 Software Engineering	*Software Engineering at Google*, chs. 1–2 (abseil.io); put something of your own under version control and write one test that would catch you changing your mind.
17–18	08 Databases	*Readings in Database Systems* (redbook.io); then write ten SQL queries against any real dataset and let the optimizer route them.
19–20	09 Networks	*Computer Networks: A Systems Approach*, ch. 1 (systemsapproach.org); then Jacobson & Karels 1988 (PDF) — the network that ate itself, and the fix your phone still runs.
21–22	10 Distributed Systems	Lamport 1978 (podc.org); then van Steen & Tanenbaum, ch. 1 (distributed-systems.net).

Days	Floor	The two days' work
23–24	11 Security & Cryptography	Cryptopals, Set 1 (cryptopals.com) — break real ciphers with your own code; *Handbook of Applied Cryptography*, ch. 1 (cacr.uwaterloo.ca/hac).
25–26	12 AI & Machine Learning	Karpathy, *Neural Networks: Zero to Hero*, first two lessons (karpathy.ai) — build backpropagation yourself; *Deep Learning* as the reference shelf (deeplearningbook.org).
27–28	13 HCI & Graphics	Nielsen's three response-time limits (nngroup.com); *Physically Based Rendering*, ch. 1 (pbr-book.org); then take one interface you use daily and separate its affordances from its signifiers.
29–30	14 The Frontier	*Quantum Computing for the Very Curious* (quantum.country); the Millennium Problems (claymath.org); then write your own list of open questions. That list is the exam, and you grade it.

วัน	ชั้น	งานของสองวันนั้น
1–2	00 แผนที่	อ่านบทที่ 00 ซ้ำ จากนั้นเริ่มหอคอยชั้นล่างสำหรับผู้อ่านทั่วไป: Petzold's Code companion (codehiddenlanguage.com) และเอกสาร Nand2Tetris (nand2tetris.org) ที่จะใช้ในวันที่ 9–10
3–4	01 ทฤษฎีการคำนวณ	Turing 1936, §9 โดยใช้บทที่ 01 เป็นไกด์ (PDF) จากนั้นอ่านหัวข้อ Church–Turing ใน Stanford Encyclopedia (plato.stanford.edu)
5–6	02 อัลกอริทึมและโครงสร้างข้อมูล	MIT OCW 6.006 เลกเชอร์ (lecture) แรกๆ (ocw.mit.edu) ลงมือเขียนอัลกอริทึมแบบแบ่งแล้วพิชิต (divide and conquer) เองสักตัวหนึ่ง แล้วจับเวลาเทียบกับแบบ naive
7–8	03 สารสนเทศและการเข้ารหัส	Shannon 1948, Part I (PDF) จากนั้นอ่านบทเปิดของ MacKay (inference.org.uk)
9–10	04 สถาปัตยกรรมคอมพิวเตอร์	Nand2Tetris projects 1–3: จากเกตตัวเดียวไปจนถึง ALU เลขคณิตจะปรากฏขึ้นจากการจัดวางเอง ด้วยมือเราเอง
11–12	05 ระบบปฏิบัติการ	OSTEP บทสนทนาเรื่อง virtualization กับบทนำ (pages.cs.wisc.edu/~remzi/OSTEP) ลอง list โพรเซสบนเครื่องตัวเอง แล้วหาว่าแต่ละตัวกำลังโกหกเรื่องอะไรอยู่
13–14	06 ภาษาและคอมไพเลอร์	Crafting Interpreters จนถึงส่วน scanner กับ parser (craftinginterpreters.com) — หอคอยของคอมไพเลอร์ ที่สร้างทีละชั้นในเอดิเตอร์ของเราเอง

วัน	ชั้น	งานของสองวันนั้น
15–16	07 วิศวกรรมซอฟต์แวร์	Software Engineering at Google, บทที่ 1–2 (abseil.io) เอางานของตัวเองสักชิ้นเข้าระบบควบคุมเวอร์ชัน (version control) แล้วเขียนเทสต์หนึ่งตัวที่จะจับได้ถ้าเราเปลี่ยนใจภายหลัง
17–18	08 ฐานข้อมูล	Readings in Database Systems (redbook.io) จากนั้นเขียนคิวรี (query) SQL สิบตัวกับชุดข้อมูลจริงสักชุด แล้วปล่อยให้ query optimizer เลือกเส้นทางเอง
19–20	09 เครือข่าย	Computer Networks: A Systems Approach, บทที่ 1 (systemsapproach.org) จากนั้น Jacobson & Karels 1988 (PDF) — เครือข่ายที่กินตัวเองจนพัง กับวิธีแก้ที่โทรศัพท์ของเรายังใช้อยู่จนถึงทุกวันนี้
21–22	10 ระบบกระจาย	Lamport 1978 (podc.org) จากนั้น van Steen & Tanenbaum บทที่ 1 (distributed-systems.net)
23–24	11 ความปลอดภัยและวิทยาการรหัสลับ	Cryptopals, Set 1 (cryptopals.com) — ทลายไซเฟอร์ (cipher) จริงด้วยโค้ดของตัวเอง จากนั้น Handbook of Applied Cryptography, บทที่ 1 (cacr.uwaterloo.ca/hac)
25–26	12 ปัญญาประดิษฐ์และการเรียนรู้ของเครื่อง	Karpathy, Neural Networks: Zero to Hero, สองบทเรียนแรก (karpathy.ai) — สร้าง backpropagation ด้วยมือเราเอง จากนั้น Deep Learning เป็นชั้นหนังสืออ้างอิง (deeplearningbook.org)
27–28	13 ปฏิสัมพันธ์มนุษย์–คอมพิวเตอร์และกราฟิกส์	ขีดจำกัดเวลาตอบสนองสามระดับของ Nielsen (nngroup.com) Physically Based Rendering, บทที่ 1 (pbr-book.org) จากนั้นเลือกอินเทอร์เฟซที่เราใช้ทุกวันสักตัว แล้วแยก affordance ออกจาก signifier ของมัน
29–30	14 พรมแดน	Quantum Computing for the Very Curious (quantum.country) the Millennium Problems (claymath.org) จากนั้นเขียนลิสต์คำถามที่ยังเปิดอยู่ของตัวเอง ลิสต์นั้นแหละคือข้อสอบ และเราเป็นคนให้คะแนนเอง

The license to be curious · ใบอนุญาตให้อยากรู้

There is no credential for what this book assumes, and none for what it hopes to leave behind. The people in these chapters mostly did not wait for one. A student refuted his professor's conjecture within a week of hearing it (02). An engineer, tired of losing his weekends to a machine that could detect errors but not fix them, invented the codes that let machines repair their own mistakes (03). An operating system that runs the world's servers was written in a month because the budget refused anything larger (05). A psychologist's word for what a chair offers a body became the working vocabulary of every screen you touch (13). None of these people had permission in any form that mattered. They had a question they could not put down, and the nerve to believe the question was theirs to ask.

> ไม่มีใบรับรองสำหรับสิ่งที่เล่มนี้สมมติไว้ และไม่มีใบรับรองสำหรับสิ่งที่มันหวังจะทิ้งไว้เบื้องหลังด้วยเช่นกัน คนในบทต่างๆ เหล่านี้ส่วนใหญ่ไม่ได้รอใบอนุญาตนั้นเลย นักศึกษาคนหนึ่งหักล้างข้อคาดการณ์ของอาจารย์ตัวเองได้ภายในหนึ่งสัปดาห์หลังได้ยินมัน (02) วิศวกรคนหนึ่งเบื่อที่เสียวันหยุดสุดสัปดาห์ไปกับเครื่องที่ตรวจจับความผิดพลาดได้แต่แก้ไม่ได้ เลยคิดค้นรหัส (code) ที่ทำให้เครื่องซ่อมความผิดพลาดของตัวเองได้ (03) ระบบปฏิบัติการที่รันเซิร์ฟเวอร์ทั่วโลกถูกเขียนขึ้นภายในหนึ่งเดือน เพราะงบไม่ยอมให้ใหญ่ไปกว่านั้น (05) คำที่นักจิตวิทยาคนหนึ่งใช้เรียกสิ่งที่เก้าอี้มอบให้ร่างกาย กลายเป็นคำศัพท์ทำงานของทุกหน้าจอที่เราแตะ (13) ไม่มีใครในนี้ได้รับอนุญาตในรูปแบบใดที่มีความหมายจริงๆ พวกเขามีแค่คำถามที่วางไม่ลง กับความกล้าที่จะเชื่อว่าคำถามนั้นเป็นของพวกเขาที่จะถามได้

That is the license, and it was never anyone's to grant. What this book has tried to supply is only the map that makes the license usable: the names of the floors, the load-bearing ideas on each, the places where the surfaces leak, and the honest admission — chapter by chapter — of what nobody knows yet. The fences that matter (01) are drawn by mathematics, not by

gatekeepers. Everything inside them is climbable, and the chapters above have shown that each floor, met on its own terms, is not just climbable but *simple* — one or two ideas, held with precision, doing all the work.

> นั่นแหละคือใบอนุญาต และไม่เคยเป็นของใครที่จะมอบให้ใครได้ตั้งแต่แรก สิ่งที่เล่มนี้พยายามให้มามีแค่แผนที่ที่ทำให้ใบอนุญาตนั้นใช้งานได้จริง: ชื่อของแต่ละชั้น ไอเดียที่แบกรับน้ำหนักอยู่บนแต่ละชั้น จุดที่ผิวรั่ว และการยอมรับตรงๆ ทีละบท ว่ายังไม่มีใครรู้อะไรบ้าง รั้วที่สำคัญจริง (01) ถูกขีดโดยคณิตศาสตร์ ไม่ใช่โดยผู้เฝ้าประตู ทุกอย่างข้างในรั้วนั้นปีนได้ และบทต่างๆ ข้างบนก็แสดงให้เห็นแล้วว่าแต่ละชั้น เมื่อเจอกันตามเงื่อนไขของมันเอง ไม่ใช่แค่ปีนได้ แต่ เรียบง่าย ด้วย — หนึ่งหรือสองไอเดีย ที่ยึดไว้อย่างแม่นยำ ทำงานทั้งหมดด้วยตัวมันเอง

Chapter 00 ended its tour with a child dragging a photo across a screen, commanding billions of switches with one finger and needing to understand none of them. That is the tower working. But *needing* was always the wrong measure. Wanting to understand is another matter entirely, and wanting to is the whole of the license. The concrete at the top is still wet. The stairs are lit. Climb.

> บทที่ 00 จบทัวร์ของมันด้วยภาพเด็กคนหนึ่งลากรูปถ่ายไปมาบนหน้าจอ สั่งสวิตช์นับพันล้านตัวด้วยนิ้วเดียว โดยไม่ต้องเข้าใจสวิตช์เหล่านั้นสักตัว นั่นคือหอคอยที่ทำงานได้ แต่ ความจำเป็น ไม่เคยเป็นตัววัดที่ถูกต้องตั้งแต่แรก การ อยากเข้าใจ เป็นคนละเรื่องกันเลย และความอยากนั้นแหละคือใบอนุญาตทั้งหมด ปูนบนสุดยังเปียกอยู่ บันไดเปิดไฟไว้แล้ว ปีนเลย

The Tower of Abstractions · the AICE Library, 2026.

หอคอยนี้ยังสร้างไม่เสร็จ — ชั้นบนสุดยังเป็นปูนเปียก และนั่นคือที่ว่างสำหรับคนที่อยากลงมือ