

Vibe Handbook

Vibe Handbook

สารบัญ

Git & GitHub	6
ระบบควบคุมเวอร์ชันคืออะไร (และทำไมคุณถึงต้องใช้)	6
Repository, Commit และ History	7
Branch: ทดลองสิ่งต่าง ๆ โดยไม่ต้องกลัว	8
ดูว่าจริง ๆ แล้วมีอะไรเปลี่ยนไป (diff)	8
ปุ่ม “อุ้ยตาย”: undo, revert, reset	9
Remote และ GitHub: โปรเจกต์ของคุณบนคลาวด์	9
Pull request และ issue	10
.gitignore และการไม่ commit ความลับเด็ดขาด	10
Repo แรกของคุณ ตั้งแต่ต้นจนจบ	10
ให้ AI ขับเคลื่อน (ในขณะที่คุณยังคุมบังเหียนอยู่)	11
สรุปและแบบฝึกหัด	11
คอมมานด์ไลน์ (Command Line)	12
เทอร์มินัลและเชลล์คืออะไรกันแน่	12
การอ่านพรมต์	13
คำสั่งที่ใช้ในชีวิตประจำวัน	13
การรันโปรแกรมและ dev server	14
การอ่านผลลัพธ์และ exit code	14
การหยุดโปรแกรมด้วย Ctrl-C	15
PATH และ “command not found”	15
การรันคำสั่งที่ AI ให้คุณมา — อย่างปลอดภัย	15
Package manager ที่คุณจะได้พบ	16
สรุปและแบบฝึกหัด	16
เว็บทำงานอย่างไร	17
สองตัวละคร: client และ server	17
URL คืออะไรกันแน่	18
วงจร request และ response	18
HTTP methods: คำกริยา	19
Status codes: มันทำงานได้ไหม	19
Headers และ cookies	19
HTTPS และไอคอนแม่กุญแจ	20
จากชื่อสู่ server: DNS	20
สิ่งนี้เชื่อมโยงกับแอปของคุณอย่างไร	20
สรุปและแบบฝึกหัด	21
API, JSON และข้อมูล	22
API คืออะไรกันแน่	22
Endpoint และ REST แบบเข้าใจง่าย	23
JSON คืออะไร	23

ชนิดข้อมูลที่คุณจะได้เจอ	24
คำขอและคำตอบจริงๆ	25
API key และ auth header	25
Rate limit และกัณฑ์ล็อกค็อก	26
วิธีอ่านเอกสาร API	26
ปล่อยให้ AI เรียก API ไป — ขณะที่คุณตรวจสอบรูปทรง	26
สรุปและแบบฝึกหัด	27
 กายวิภาคของแอปหนึ่งตัว	28
ภาพรวมใหญ่	28
ฟรอนต์เอนด์ แบ็กเอนด์ ฐานข้อมูล โฮสติ้ง	29
HTML, CSS และ JavaScript	29
แพ็กเกจและดีเพนเดนซี	30
ตัวแปรสภาพแวดล้อมและการตั้งค่า	30
รูปร่างของโปรเจกต์หนึ่งชิ้น	31
คำขอไหลผ่านอย่างไร	32
สรุปและแบบฝึกหัด	32
 กรอบความคิดแบบ Vibe Coding	34
Vibe coding คืออะไรกันแน่	34
การเปลี่ยนกรอบความคิด	35
วงรอบ: ความตั้งใจ → สร้าง → ตรวจสอบ → ปรับแก้	36
จุดที่มันเปลี่ยนประกาย	37
จุดที่มันลั่นเหลว	37
กัณฑ์ที่พบบ่อย	38
รสนิยม (Taste) คือทักษะหลักตัวใหม่	38
ความคาดหวังที่สมจริง	38
สรุปและแบบฝึกหัด	39
 การตั้งค่าพื้นที่ทำงาน AI ของคุณ	40
เลือกเอดิเตอร์หรือผู้ช่วยที่ออกแบบมาเพื่อ AI	40
จัดโครงสร้างโปรเจกต์ของคุณให้ AI สามารถดูได้	41
เขียนไฟล์บริบท (AGENTS.md)	42
ทำให้การควบคุมเวอร์ชันเป็นปฏิกิริยาสะท้อน	43
รักษาวางจรป้อนกลับให้กระชับ	43
จัดการความล้นอย่างระมัดระวัง	44
การตั้งค่าโลคัลที่มีเหตุผลทำให้การเปลี่ยนแปลงของ AI ปลอดภัย	44
การวินิจฉัยแบบสด (LSP)	44
การขยายขีดความสามารถของเอเจนต์	45
macOS กับ Windows	46
TUI เทียบกับ GUI	46
วิศวกรรมสายรัดนิรภัย (Harness)	47
ความปลอดภัยด้านต้นทุน: CLI เทียบกับ API	48
สรุปและฝึกฝน	48
 การเขียนพอร์มต์แบบวิศวกร	49
ให้บริบทก่อนให้โจทย์	50
ระบุ Input, Output และกรณีขอบ (Edge Case)	50
ทำงานเป็นขั้นตอนเล็ก ๆ	51
ขอแผนก่อนขอโค้ด	51
แก้ทีละส่วน (Diff) ไม่ใช่เขียนใหม่ทั้งหมด	52
ให้ตัวอย่าง	52
บอก AI ว่าอะไรที่ไม่ควรทำ	53
จัดการบริบทเหมือนทรัพยากรอย่างหนึ่ง	53

รูปแบบที่ควรหลีกเลี่ยง (Anti-Pattern ที่พบบ่อย)	54
แก่นความคิดหลัก	54
สรุปและแบบฝึกหัด	55
จากไอเดียสู่สเปกในไม่กี่นาที	56
ทำไมต้องมีสเปก ไม่ใช่แค่ใช้สัญชาตญาณ	56
ขั้นตอนที่ 1: ทำความเข้าใจปัญหาให้ชัด	57
ขั้นตอนที่ 2: กำหนดขอบเขตและสิ่งที่ไม่ทำ	58
ขั้นตอนที่ 3: เขียน User Story	58
ขั้นตอนที่ 4: ร่าง PRD แบบเบา ๆ	59
ขั้นตอนที่ 5: ร่างโมเดลข้อมูล	60
ขั้นตอนที่ 6: แบ่งสเปกออกเป็นงานขนาดพอดีสำหรับ Vibe Coding	60
รักษาสเปกให้มีชีวิตอยู่	61
กับดักที่พบบ่อย: ระบุทางแก้ ไม่ใช่ระบุปัญหา	61
ผลตอบแทน	62
สรุปและฝึกปฏิบัติ	62
เลือกสแต็กของคุณ: ภาษาต่าง ๆ แบบเห็นภาพรวม	63
JavaScript / TypeScript	64
Python	64
Go	65
Rust	65
กล่าวถึงภาษาอื่น ๆ พอสังเขป	66
เปรียบเทียบแบบเห็นภาพรวม	66
ต้นทุนที่แท้จริงของทางเลือกแปลกใหม่	67
เลือกตามเป้าหมาย	67
สรุปและฝึกฝน	68
สร้างพีเจอร์ทึ่ละ Vibe	69
วงจรการสร้าง (The Build Loop)	69
“Vertical Slice” คืออะไร	70
ตัวอย่างที่ลงมือทำจริง: พีเจอร์ทึ่ “รายการโปรด”	71
การอ่าน Diff โดยไม่ต้องเข้าใจทุกบรรทัด	73
เมื่อไหร่ควรปล่อยให้ AI รันเอง กับเมื่อไหร่ควรคุมพวงมาลัยเอง	73
รักษาให้แอปทำงานได้อยู่เสมอ	73
คอมมิตเป็นขั้นเล็ก ๆ	74
เมื่อ Slice ใหญ่เกินไป	74
สรุปประเด็นสำคัญ	74
สรุปและฝึกปฏิบัติ	75
การดีบั๊กด้วย AI	76
แนวคิดหลัก	76
เช็กलिस्टการดีบั๊ก	77
การป้อนบริบทที่ถูกต้องให้ AI	78
การอ่านข้อผิดพลาดที่คุณไม่เข้าใจ	79
สมมติฐาน การทำ logging และการค้นหาแบบทวิภาค	79
สาเหตุที่แท้จริง เทียบกับ อาการ	80
เมื่อ AI วนเวียนไม่จบ	80
ทำความเข้าใจทางแก้ก่อนที่จะเก็บมันไว้	81
บทสรุป	81
สรุปและฝึกฝน	81
โครงสร้างพื้นฐานเซิร์ฟเวอร์	82

ห้าที่อยู่ที่โค้ดสามารถอาศัยอยู่ได้	83
ศัพท์เทคนิคสองสามคำ ถอดรหัสให้แล้ว	84
ตารางข้อแลกเปลี่ยน	84
จะเลือกอันไหนเมื่อไหร่	85
AI ต่อสายการ deploy และการตั้งค่าอย่างไร	85
เส้นทางเริ่มต้นที่แนะนำสำหรับผู้สร้างคนเดียว	86
“มีคนจัดการให้” หมายถึงอะไรกันแน่ (และคุณยังต้องรับผิดชอบอะไรอยู่)	87
รายการตรวจสอบก่อนเปิดตัวแบบรวดเร็ว	88
สรุปส่งท้าย	88
สรุปและฝึกฝน	88
โครงสร้างพื้นฐานฐานข้อมูล	89
รูปแบบทั้งสามของการจัดเก็บข้อมูล	90
ดูแลเองหรือใช้บริการที่มีคนดูแลให้	91
พื้นฐานการออกแบบสคีมา	91
Migration: การเปลี่ยนสคีมาอย่างปลอดภัย	92
การทำ Index	93
การแคช	94
คิว (Queue)	94
ข้อผิดพลาดทั่วไปของมือใหม่	94
สั่งให้ AI สร้างชั้นข้อมูลของคุณ	95
สรุปและแบบฝึกหัด	95
รื้อกันความปลอดภัย การทดสอบ และการรีวิว	96
ทำไมคุณถึงต้องมีรื้อกันความปลอดภัยเมื่อใช้งาน AI	97
การทดสอบ: ตาข่ายนิรภัยของคุณ	97
การตรวจสอบชนิดข้อมูลและ linting: จับความผิดพลาดได้ฟรี	98
จำกัดขอบเขตการเปลี่ยนแปลงของ AI	99
อ่าน diff อย่างมีวิจารณญาณ	99
ด้านรีวิว: เช็คลิสต์ก่อนการ merge	100
พื้นฐานของ CI: ทำให้การตรวจสอบเป็นแบบอัตโนมัติ	100
นิสัยที่ปลอดภัยโดยค่าเริ่มต้น	101
เมื่อรื้อกันความปลอดภัยกรี๊ดร้อง: คัดแยกก่อนแก้	101
สรุปและแบบฝึกหัด	102
การนำแอปขึ้นใช้งานจริง (Shipping to Production)	103
เลือกโฮสต์และ Deploy	104
โดเมนกำหนดเองและ DNS	105
ตัวแปรสภาพแวดล้อมและความลับ (Environment Variables and Secrets)	106
การเชื่อมต่อระบบชำระเงิน	106
Staging และ Rollback	107
การเฝ้าติดตามและติดตามข้อผิดพลาด	108
ประสิทธิภาพและการแคช	108
รายการตรวจสอบก่อนเปิดตัว	109
หลังจากที่คุณเปิดตัว	109
สรุปและฝึกปฏิบัติ	109
กรณีศึกษา: จากศูนย์สู่การเปิดตัว	110
กรณีศึกษาที่ 1: หน้า Landing Page หลายภาษา + ระบบชำระเงิน	111
กรณีศึกษาที่ 2: เครื่องมือ SaaS ขนาดเล็กที่มีระบบล็อกอินและฐานข้อมูล	112
กรณีศึกษาที่ 3: เครื่องมือทำงานอัตโนมัติส่วนตัว	114
กรณีศึกษาที่ 4: เว็บไซต์คอนเทนต์ที่ไม่ต้องคอยดูแล	116

สิ่งที่ทั้งสี่กรณีมีร่วมกัน	118
สรุปและแบบฝึกหัด	118
ความปลอดภัยสำหรับนัก Vibe Coding	119
ทำไมโค้ดที่ AI สร้างจึงไม่ปลอดภัยโดยค่าเริ่มต้น	120
Injection: เมื่อข้อมูลนำเข้ากลายเป็นโค้ด	120
ความลับและ API key: ก้นดักคีย์ที่ถูกเปิดเผย	121
Authentication vs Authorization: เอนด์พอยต์ที่ใครก็เรียกได้	122
การจัดการไฟล์และการอัปโหลด	123
ความเสี่ยงจากดีเพนเดนซี: แพ็กเกจที่ AI คิดขึ้นมาเอง	123
ประตูดตรวจสอบความปลอดภัย	124
ความประมาทจากระบบอัตโนมัติ ในเวอร์ชันความปลอดภัย	125
สรุปและฝึกฝน	125
การจัดการเอเจนต์ (Agent Orchestration)	126
สามระดับ	126
เมื่อไรควรปล่อยให้มันรัน vs. บังคับทิศทางด้วยตัวเอง	127
กำหนดขอบเขตงานให้เดินหลงไม่ได้	127
เอเจนต์แบบพื้นหลังและแบบขนาน	128
หลายเอเจนต์: กระจายงาน, ฝ่ายตรวจสอบปฏิบัติ, การสังเคราะห์	129
การตรวจสอบผลลัพธ์ในระดับใหญ่	130
ค่าใช้จ่ายและวงจรที่วิ่งไม่หยุด	131
ข้อจำกัดที่ต้องยอมรับจริงๆ	131
สรุปและแบบฝึกหัด	132

Git & GitHub



ถ้าคุณจะเรียนรู้เครื่องมือแค่ตัวเดียวก่อนเริ่ม vibe coding ให้เลือกตัวนี้ Git คือสิ่งที่ทำให้คุณกล้าได้ มันคือเหตุผลที่คุณสามารถบอก AI ว่า “เขียนไฟล์นี้ใหม่ทั้งหมด” ได้โดยไม่ต้องนอนไม่หลับ เพราะคุณสามารถย้อนกลับไปยังเวอร์ชันที่ใช้ทำงานได้เสมอ

นี่คือเหตุผลตรงไปตรงมาว่าทำไมเรื่องนี้ถึงสำคัญ เมื่อคุณสั่งให้ AI สร้างซอฟต์แวร์ มัน จะ ทำพังบ่อย ๆ อย่างแน่นอน มันอาจเขียนไฟล์ใหม่ทับด้วยความมั่นใจ ลบส่วนที่เคยใช้งานได้ออกไป แล้วส่งความยุ่งเหยิงกลับมาให้คุณ ความแตกต่างระหว่างบ่ายวันที่น่าหงุดหงิดกับสัปดาห์ที่เสียเปล่าไปเลยคือคุณมีตาข่ายนิรภัยหรือไม่ Git คือตาข่ายนั้น บทนี้จะสอนตั้งแต่ศูนย์ ไม่ต้องมีพื้นฐานการเขียนโค้ดมาก่อน

ระบบควบคุมเวอร์ชันคืออะไร (และทำไมคุณถึงต้องใช้)

ลองนึกภาพวิดีโอเกมที่มีจุดเชฟ คุณเล่นผ่านช่วงที่ยาก คุณเชฟไว้ แล้วลองทำอะไรที่เสี่ยง ๆ ถ้ามันพลาด คุณก็แค่โหลดเซฟล่าสุดแทนที่จะเริ่มเล่นเกมใหม่ทั้งหมด ระบบควบคุมเวอร์ชันคือจุดเชฟสำหรับโปรเจกต์ของคุณ

ถ้าไม่มีสิ่งนี้ โปรเจกต์ของคุณจะมีสถานะเดียวเท่านั้น คือสภาพปัจจุบันตรงหน้า ถ้า AI ทำโค้ดพัง เวอร์ชันที่ดีก็หายไปเลย แต่ถ้ามีระบบควบคุมเวอร์ชัน คุณจะเก็บไทม์ไลน์เต็มรูปแบบของทุกสถานะที่บันทึกไว้ และสามารถย้อนกลับไปยังจุดใดก็ได้

Git คือเครื่องมือควบคุมเวอร์ชันที่ใช้กันแพร่หลายที่สุด มันทำงานอยู่บนคอมพิวเตอร์ของคุณและเจียบ ๆ บันทึกภาพรวม (snapshot) ของโปรเจกต์ทุกครั้งที่คุณสั่ง มีสิ่งดี ๆ หลายอย่างที่คุณได้มาฟรี ๆ:

- ตาข่ายนิรภัย — ย้อนกลับการเปลี่ยนแปลงที่ผิดพลาดได้ แม้จะผ่านไปหลายชั่วโมงแล้ว

- ประวัติ — ดูได้แน่ชัดว่ามีอะไรเปลี่ยนไป เมื่อไหร่ และทำไม
- ความมั่นใจ — ทดลองได้อย่างอิสระ เพราะไม่มีอะไรสูญหายไปจริง ๆ

ข้อสุดท้ายนี้แหละคือรางวัลที่แท้จริง vibe coding จะได้ผลดีที่สุดเมื่อคุณเคลื่อนไหวเร็วและลองสิ่งต่าง ๆ Git คือสิ่งที่ทำให้การเคลื่อนไหวเร็วปลอดภัยแทนที่จะสุ่มเสี่ยง

ลองนึกภาพประวัติของโปรเจกต์คุณเป็นแถวของจุดเชฟที่คุณสามารถกลับไปได้เสมอ:

```

past -----> now

( C1 )-->( C2 )-->( C3 )-->( C4 )-->( C5 )
  |       |       |       |       |
"initial" "homepage" "login" "oops, broke" "fixed it"

each ( C ) = one commit = one save point you can jump back to

```

Repository, Commit และ History

มีสามคำที่คุณจะได้ยินบ่อยมาก มาไขความสับสนของมันกัน

repository (หรือ “repo”) ก็แค่โฟลเดอร์ที่ Git กำลังจับตาอยู่ คุณเปลี่ยนโฟลเดอร์ธรรมดาให้เป็น repo แค่ครั้งเดียว จากนั้น Git จะติดตามทุกอย่างที่อยู่ในนั้น

commit คือจุดเชฟหนึ่งจุด มันคือภาพรวมของทุกไฟล์ ณ ช่วงเวลาหนึ่ง บวกกับข้อความสั้น ๆ จากคุณที่อธิบายว่ามีอะไรเปลี่ยนไป (“เพิ่มหน้าโฮมเพจ”, “แก้ปุ่มสมัครสมาชิกที่พัง”) คุณจะ commit อยู่ตลอดเวลา — ลองคิดว่าแต่ละครั้งคือจุดตรวจสอบ (checkpoint) ที่คุณอาจอยากย้อนกลับมา

history คือรายการลำดับของ commit ทั้งหมดของคุณ มันคือไทม์ไลน์ คุณสามารถเลื่อนดูมันได้ อ่านว่าแต่ละการเปลี่ยนแปลงคืออะไร และย้อนเวลากลับไปยังจุดไหนก็ได้

นี่คือหน้าตาของมันในทางปฏิบัติ ไม่ต้องกังวลว่าต้องจำให้ได้ — ส่วนใหญ่คุณจะให้ AI รันคำสั่งเหล่านี้แทนคุณ — แต่อ่านตามไปเพื่อให้คำสั่งเหล่านี้เล็กเป็นเรื่องลึกลับ:

```

# Turn the current folder into a Git repository (do this once)
git init

# See which files have changed since your last save point
git status

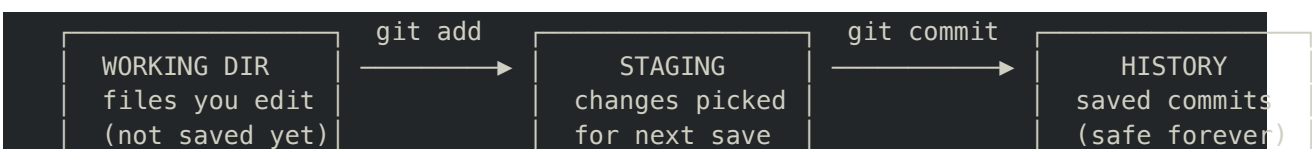
# Stage your changes, then save them as a commit with a message
git add .
git commit -m "Add the homepage layout"

# Look at your history - every commit, newest first
git log --oneline

```

git add . คือการที่คุณบอกว่า “รวมการเปลี่ยนแปลงทั้งหมดของฉันไว้ในจุดเชฟถัดไป” git commit คือการเซฟจริง ๆ ส่วน -m “...” คือบันทึกช่วยจำถึงตัวคุณเองในอนาคต เขียนบันทึกที่คุณจะเข้าใจได้แม้ผ่านไปหนึ่งเดือน

สองคำสั่งนี้จะเคลื่อนงานของคุณผ่านสามขั้นตอน ไฟล์ที่คุณแก้ไขจะยังไม่ถูกบันทึกจนกว่ามันจะเดินทางไปจนสุดทางด้านขวา:



you change
a file

you choose what
goes in the commit

it becomes a
permanent snapshot

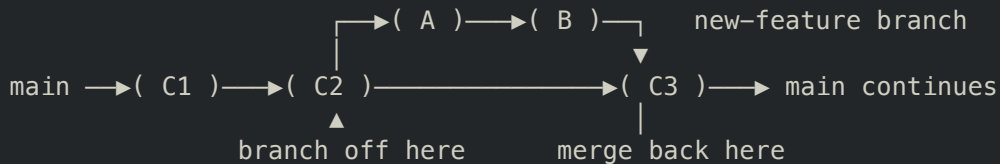
Branch: ทดลองสิ่งต่าง ๆ โดยไม่ต้องกลัว

branch คือสำเนาคู่ขนานของโปรเจกต์ที่คุณสามารถทดลองได้ งานหลักของคุณจะอยู่บน branch ที่มักเรียกว่า main เมื่อคุณอยากลองทำอะไรสักอย่าง — ฟีเจอร์ใหม่ การเขียนใหม่ที่เสี่ยงโดย AI — คุณสร้าง branch ใหม่ ทำงานอยู่ที่นั่น และ main ของคุณจะยังคงไม่ถูกแตะต้องและปลอดภัย

ถ้าการทดลองได้ผล คุณก็ merge มันกลับเข้าไปใน main ถ้าไม่ได้ผล คุณก็ทิ้ง branch นั้นไป และ main ก็จะไม่รู้เลยว่าเกิดอะไรขึ้น

นี่คือวิธีที่คุณควรปฏิบัติกับการเปลี่ยนแปลงใหญ่ ๆ ของ AI พอดี “สร้างระบบการชำระเงินทั้งหมด” สมควรมี branch เป็นของตัวเอง ถ้า AI สร้างสิ่งที่ต้อออกมา คุณก็ merge มัน ถ้ามันสร้างหายนะออกมา คุณก็ลบ branch นั้นทิ้ง แล้วคุณก็ไม่เสียอะไรเลย

ในเชิงภาพ branch จะแยกออกจาก main เติบโตด้วยตัวเอง แล้วค่อย merge กลับเข้ามา:



```
# Create a new branch and switch to it
git checkout -b new-feature

# ...do your work, make commits...

# Switch back to your safe main branch
git checkout main

# Merge the good work from your branch into main
git merge new-feature
```

ดูว่าจริง ๆ แล้วมีอะไรเปลี่ยนไป (diff)

ก่อนที่จะบันทึกการเปลี่ยนแปลง — และ โดยเฉพาะ ก่อนที่คุณจะยอมรับการเปลี่ยนแปลงที่ AI ทำ — คุณอยากเห็นแน่ชัดว่าอะไรต่างไปบ้าง นั่นคือ diff มันแสดงให้เห็นที่บรรทัดว่าอะไรถูกเพิ่มเข้ามา (ปกติเป็นสีเขียว) และอะไรถูกลบออกไป (ปกติเป็นสีแดง)

นี่คือหนึ่งในนิสัยที่สำคัญที่สุดของคุณ ปรชญาทั้งหมดของ vibe coding คือ เข้าใจสิ่งที่คุณส่งมอบ คุณไม่จำเป็นต้องพิมพ์โค้ดเอง แต่คุณต้องดูสิ่งที่เปลี่ยนไปและเข้าใจมันคร่าว ๆ diff คือที่ที่คุณทำสิ่งนั้น

```
# See what you've changed but not yet committed
git diff

# See what's staged and ready to be committed
git diff --staged
```

เมื่อ AI บอกว่า “เสร็จแล้ว ผมอัปเดตไฟล์แล้ว” ขั้นตอนต่อไปของคุณคือดู diff ถ้าคุณเห็นว่ามันลบบางอย่างที่คุณต้องการออกไป คุณจะจับมันได้ ตอนนี้ — ไม่ใช่หลังจากที่มันถูกส่งมอบไปแล้ว

ปุ่ม “อียิตาย”: undo, revert, reset

นี่คือส่วนที่เปลี่ยน Git จากงานจุกจิกให้กลายเป็นพลังวิเศษ เมื่อมีอะไรผิดพลาด — และมันจะเกิดขึ้นแน่ ๆ — นี่คือวิธีย้อนกลับ

- ทั้งการเปลี่ยนแปลงที่ยังไม่ได้บันทึก — คุณแก้ไข (หรือ AI แก้ไข) บางไฟล์แล้วทำให้แย่ลง แต่คุณยังไม่ได้ commit คุณสามารถยกเลิกทุกอย่างกลับไปยัง commit ที่ดีล่าสุดได้
- Revert commit — คุณบันทึกการเปลี่ยนแปลงไปแล้ว แต่ภายหลังพบว่ามันแย่ git revert จะสร้าง commit ใหม่ที่ย้อนกลับ commit แ่ ๆ นั้น โดยยังคงประวัติของคุณให้ซื่อสัตย์อยู่
- Reset ไปยังจุดก่อนหน้า — คุณอยากย้อนโปรเจกต์กลับไปเป็นแบบที่มันเคยเป็นเมื่อหลาย commit ก่อนหน้า

```
# Undo all uncommitted changes and go back to your last commit
git restore .

# Safely undo a specific bad commit (creates a new "undo" commit)
git revert <commit-id>

# Rewind to an earlier commit (powerful – ask the AI to explain before using)
git reset --hard <commit-id>
```

ข้อควรระวัง: git reset --hard จะทิ้งงานอย่างถาวร มันเป็นเครื่องมือที่ถูกต้องในบางครั้ง แต่ก็เป็นตัวที่ต้องระวังเช่นกัน เมื่อไม่แน่ใจ ให้ถาม AI ว่า “คำสั่งนี้จะทำอะไรกับไฟล์ของฉันบ้าง” ก่อนที่จะรันมัน

Remote และ GitHub: โปรเจกต์ของคุณบนคลาวด์

ทุกอย่างที่ผ่านมาเกิดขึ้นบนคอมพิวเตอร์ของคุณเอง GitHub คือเว็บไซต์ที่เก็บสำเนาของ repo ของคุณไว้บนคลาวด์ สำเนานั้นเรียกว่า remote

ทำไมต้องยุ่งยากด้วย มีสามเหตุผล: มันเป็นข้อมูลสำรองในกรณีที่แล็ปท็อปของคุณเสีย มันเป็นวิธีที่คุณแชร์โค้ดกับคนอื่น (หรือกับเครื่องมือ AI และบริการ deploy) และมันคือที่ที่การทำงานร่วมกันเกิดขึ้น

คุณเชื่อมโยง repo ในเครื่องของคุณเข้ากับ remote เพียงครั้งเดียว จากนั้นคุณจะ push (ส่ง commit ของคุณขึ้นไปยัง GitHub) และ pull (ดึง commit จาก GitHub ลงมา)

ลองนึกภาพสำเนาของโปรเจกต์คุณสองชุด — ชุดหนึ่งอยู่บนแล็ปท็อป อีกชุดอยู่บนคลาวด์ — ที่ถูกทำให้ตรงกันด้วย push และ pull:



```
# Connect your local repo to a GitHub repo (one time)
git remote add origin https://github.com/yourname/yourproject.git

# Send your commits up to GitHub
git push origin main

# Bring down any commits that are on GitHub but not on your computer
git pull origin main
```

ลองคิดว่า push คือ “บันทึกขึ้นคลาวด์” และ pull คือ “ดึงข้อมูลล่าสุดจากคลาวด์มา”

Pull request และ issue

มีสองฟีเจอร์ของ GitHub ที่คุณจะพบเจอในไม่ช้า

pull request (PR) คือข้อเสนอที่จะ merge branch หนึ่งเข้ากับอีก branch หนึ่ง พร้อมพื้นที่ในตัวสำหรับตรวจสอบการเปลี่ยนแปลงก่อนที่มันจะถูกรวมเข้าไป แม้จะทำงานคนเดียว PR ก็ยังมีประโยชน์: มันให้หน้าจอดีๆ ที่สะอาดที่แสดงการเปลี่ยนแปลงทุกอย่าง เพื่อให้คุณตรวจสอบงานของ AI ก่อนจะ merge เข้าไปใน main มันคือนิสัยการดู diff ที่ถูกทำให้เป็นทางการมากขึ้น

issue คือบันทึกที่ถูกติดตามเกี่ยวกับสิ่งที่ต้องทำหรือแก้ไข — บั๊ก ไอเดียฟีเจอร์ใหม่ หรือ TODO issue คือวิธีที่คุณเก็บรายการสิ่งที่ต้องทำให้ติดอยู่กับตัวโปรเจกต์เอง แทนที่จะกระจัดกระจายอยู่ในหัวคุณ

.gitignore และการไม่ commit ความลับเด็ดขาด

มีบางไฟล์ที่ ไม่ควร ถูกใส่เข้าไปใน Git เลย สิ่งสำคัญที่สุดคือ:

- ความลับ (secrets) — คีย์ API (Application Programming Interface) รหัสผ่าน ข้อมูลรับรองฐานข้อมูล ถ้าคุณ commit ความลับขึ้น GitHub ให้ถือว่ามันรั่วไหลไปแล้ว แม้ว่า คุณจะลบมันออกในภายหลัง เพราะประวัติยังจดจำมันไว้อยู่
- ขยะ — ไฟล์ชั่วคราว dependency ที่ดาวน์โหลดมา (เช่น node_modules) ผลลัพธ์จากการ build สิ่งเหล่านี้ใหญ่และไม่มีประโยชน์ที่จะติดตาม

คุณควบคุมสิ่งนี้ด้วยไฟล์ที่ชื่อว่า .gitignore คุณระบุรูปแบบต่าง ๆ ไว้ในไฟล์นั้น แล้ว Git จะทำเหมือนว่าไฟล์เหล่านั้นไม่มีอยู่จริง สิ่งนี้สำคัญมากสำหรับ vibe coder เพราะเครื่องมือ AI ชอบสร้างไฟล์ config ที่มีคีย์จริง ๆ อยู่ในนั้น และมันง่ายมากที่จะ commit ไฟล์แบบนั้นไปโดยไม่ได้ตั้งใจ

```
# A simple .gitignore file
.env
node_modules/
*.log
```

บรรทัด .env คือบรรทัดที่สำคัญที่สุด — นั่นคือที่ที่ความลับมักจะอยู่ ให้เพิ่ม .gitignore ก่อน commit แรกของคุณ และสร้างนิสัยถาม AI ว่า “มีอะไรใน commit นี้ที่ไม่ควรเปิดเผยต่อสาธารณะไหม”

Repo แรกของคุณ ตั้งแต่ต้นจนจบ

นี่คือวงจรทั้งหมดในที่เดียว สร้างโฟลเดอร์ ทำให้มันเป็น repo บันทึกงานของคุณ แล้ว push ขึ้น GitHub:

```
# 1. Start a new project
mkdir my-first-project
cd my-first-project
git init

# 2. Protect your secrets before anything else
echo ".env" > .gitignore

# 3. Make your first save point
git add .
git commit -m "Initial commit"

# 4. Connect to GitHub and push it up
git remote add origin https://github.com/yourname/my-first-project.git
git push -u origin main
```

แค่นี้เอง ตอนนี้คุณมีโปรเจกต์ที่มีตาข่ายนิรภัยและข้อมูลสำรองบนคลาวด์แล้ว ทุกครั้งที่คุณมีความคืบหน้า ให้ทำจิ้งหะ add → commit → push ซ้ำ ๆ

ให้ AI ขับเคลื่อน (ในขณะที่คุณยังคุมบังเหียนอยู่)

นี่คือส่วนที่ปลดปล่อยคุณ: คุณไม่จำเป็นต้องจำคำสั่งเหล่านี้เลย คุณสามารถบอก AI ว่า “commit สิ่งนี้ด้วยข้อความที่เหมาะสม” หรือ “สร้าง branch สำหรับฟีเจอร์ใหม่แล้วสลับไปที่นั่น” แล้วมันจะรันคำสั่งที่ถูกต้องให้ นี่เป็นวิธีทำงานที่ดีเหมาะสมอย่างยิ่ง

ข้อแม้ — และมันเป็นข้อแม้เดียวกับที่เหลือของหนังสือเล่มนี้ — คือคุณยังคงรับผิดชอบในการเข้าใจว่า มันทำอะไรไป แม้ว่าคุณจะได้พิมพ์เอง ทางสายกลางที่ใช้ได้จริงคือ:

- ให้ AI รันคำสั่ง แต่ขอให้มันบอกคุณว่าแต่ละคำสั่งทำอะไร เป็นภาษาธรรมดา ในช่วงแรก ๆ
- อ่าน diff ด้วยตัวเอง ก่อนทุก commit นี่คือสิ่งที่ต้องระวังไม่ได้ มันคือวิธีที่คุณจับได้ว่า AI ลมบางอย่างที่ไม่ควรลบ
- Commit ป่อย ๆ เป็นชิ้นเล็ก ๆ จุดเชฟเล็ก ๆ จำนวนมากดีกว่าจุดเชฟยักษ์จุดเดียว เพราะการย้อนกลับจะแม่นยำ แทนที่จะเป็นแบบทั้งหมดหรือไม่มีอะไรเลย
- สร้าง branch ก่อนทำอะไรใหญ่ ๆ บอก AI ให้สร้าง branch ก่อนการเขียนใหม่ครั้งใหญ่ เพื่อให้ main ยังคงปลอดภัย
- เมื่อคำสั่งฟังดูทำลายล้าง (อะไรก็ตามที่มี reset --hard, force-push หรือการลบ branch) ให้ถามว่า “สิ่งนี้ทำอะไรกันแน่” ก่อนที่จะอนุมัติ

คุณคือหัวหน้าทีมเทคนิค AI คือเพื่อนร่วมทีมที่รวดเร็ว มีความสามารถ แต่บางครั้งก็สับสน Git คือกระบวนการที่ป้องกันไม่ให้เพื่อนร่วมทีมคนนั้นสร้างความเสียหายจริง ๆ เรียนรู้มันให้ดีพอที่จะสั่งการมันได้อย่างมั่นใจ แล้วคุณจะไม่วันก๊วป “เขียนใหม่ทั้งหมด” อีกต่อไป — เพราะคุณจะมีทางกลับเสมอ

สรุปและแบบฝึกหัด

ประเด็นสำคัญ

- Git คือ จุดเชฟสำหรับโปรเจกต์ของคุณ — ไลน์ไทม์รูปแบบที่คุณสามารถย้อนกลับไปได้ ซึ่งเป็นสิ่งที่ทำให้คุณปล่อย AI ทำงานได้โดยไม่ต้องกลัว
- จิ้งหะหลักคือ add → commit → push: stage การเปลี่ยนแปลงของคุณ บันทึกลงเป็น commit พร้อมข้อความที่ชัดเจน แล้วส่งขึ้นไปยัง GitHub เพื่อสำรองข้อมูลและแชร์
- สร้าง branch ก่อนทำอะไรใหญ่ ๆ และ อ่าน diff ก่อนทุก commit — นั่นคือวิธีที่คุณจับได้ว่า AI ลมบางอย่างที่ไม่ควรลบ
- .gitignore ช่วยกันความลับ (เช่น .env) และขยะ (เช่น node_modules/) ไม่ให้เข้าไปอยู่ในประวัติของคุณ ความลับที่ถูก commit ขึ้น GitHub จะรั่วไหลตลอดไป แม้ว่า你会ลบมันออกในภายหลัง
- คุณให้ AI รันคำสั่งได้ แต่คุณยังคงรับผิดชอบในการเข้าใจว่าแต่ละคำสั่งทำอะไร โดยเฉพาะคำสั่งที่ทำลายล้างอย่างเช่น reset --hard

ลองทำดู

สร้างโฟลเดอร์ใหม่เอี่ยมแล้วรันวงจรทั้งหมดด้วยตัวเอง ด้วยมือ: git init สร้าง .gitignore ที่มี .env อยู่ในนั้น เพิ่มไฟล์ notes.txt ง่าย ๆ จากนั้น git add . และ git commit -m "Initial commit" ตอนนั้น git log --oneline เพื่อดูจุดเชฟของคุณ แก้ไขไฟล์ แล้วรัน git diff เพื่อดู Git แสดงให้คุณเห็นแน่ชัดว่าอะไรเปลี่ยนแปลงไป คุณเพิ่งใช้นิสัยหลักทุกอย่างในบทนี้กับโปรเจกต์ที่ไม่มีอะไรผิดพลาดได้เลย

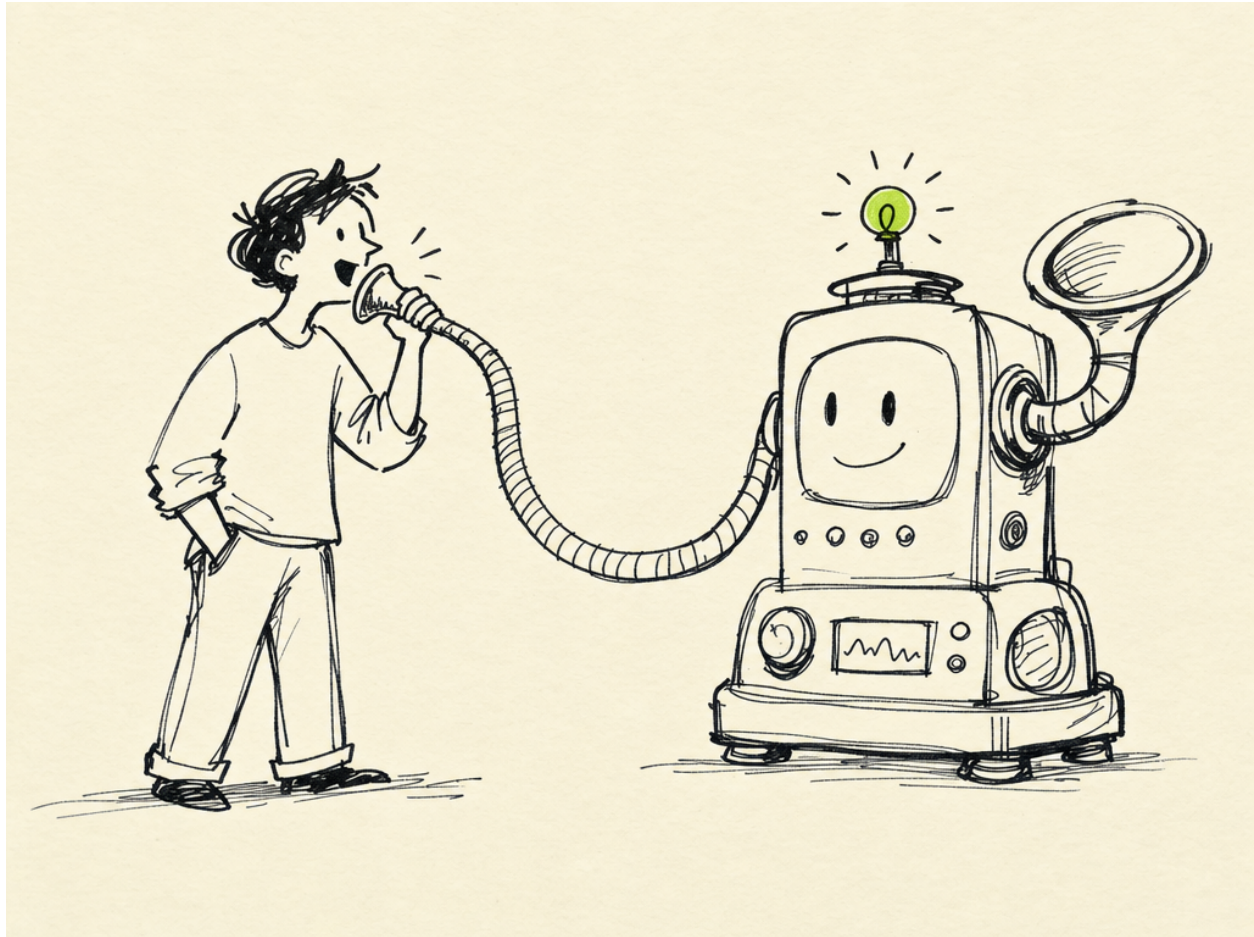
คำสั่งประจำบท

```
I'm learning Git as a beginner and I want you to run the commands for me,
but teach as you go. For the task below:
<describe what you want to do – e.g. "save my current work" or "try a risky change safely">

– Tell me which Git commands you'll run and what each one does in plain English.
```


- If anything is destructive (reset --hard, force-push, deleting a branch), warn me clearly and explain what I'd lose before doing it.
- Before any commit, show me the diff and summarize what changed.
- Remind me if there's anything in the change that shouldn't be public (secrets, keys).

คอมมานด์ไลน์ (Command Line)



ครั้งแรกที่คุณเห็นเครื่องมือ AI สำหรับเขียนโค้ดทำงาน จะมีหน้าต่างสีดำที่มีตัวอักษรไหลผ่านไปเรื่อยๆ พร้อมเคอร์เซอร์กะพริบรอให้คุณพิมพ์ หน้าต่างนั้นคือ “คอมมานด์ไลน์” (command line) สำหรับคนที่ไม่ใช่โปรแกรมเมอร์หลายคน นี่เป็นส่วนที่น่ากลัวที่สุดในการเดินทางทั้งหมด — มันดูเหมือนในหนัง เหมือนว่าถ้าพิมพ์ผิดนิดเดียวจะลบฮาร์ดดิสก์หมด แต่ความจริงไม่ใช่แบบนั้น คอมมานด์ไลน์เป็นเพียงอีกวิธีหนึ่งในการบอกคอมพิวเตอร์ว่าให้ทำอะไร: ด้วยคำพูดแทนการคลิก

บทนี้จะช่วยขจัดความน่ากลัวนั้นออกไป คุณไม่จำเป็นต้องจำคำสั่งเป็นร้อยคำสั่ง สิ่งที่คุณต้องมีคือความเข้าใจว่ากำลังเกิดอะไรขึ้น จดจำคำสั่งที่ใช้บ่อยๆ ได้สักหยิบมือ และรู้วิธีรันคำสั่งที่ AI ยืนยันอย่างปลอดภัย เท่านั้นก็เพียงพอที่จะเลิกกลัวและเริ่มใช้งานมันได้อย่างมีประสิทธิภาพ

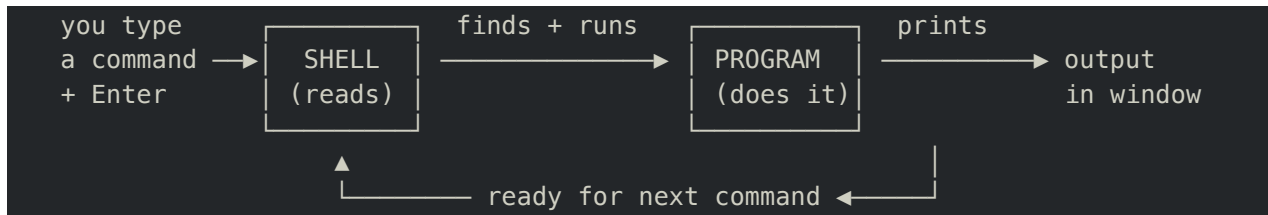
เทอร์มินัลและเชลล์คืออะไรกันแน่

เวลาคุณใช้แอปทั่วไป คุณคลิกปุ่มต่างๆ และแอปจะแปลงการคลิกของคุณเป็นคำสั่งให้คอมพิวเตอร์ทำงาน คอมมานด์ไลน์ตัดขั้นตอนปุ่มออกไป คุณพิมพ์คำสั่ง กด Enter แล้วคอมพิวเตอร์ก็ทำตามนั้น

มีสองคำที่คุณจะได้ยินบ่อยๆ:

- เทอร์มินัล (Terminal) — หน้าต่างที่คุณพิมพ์ใส่ มันก็แค่กล่องข้อความที่มีประวัติการพิมพ์ (บน Mac เรียกว่า “Terminal” หรือ “iTerm”; บน Windows เรียกว่า “PowerShell” หรือ “Windows Terminal”; บน Linux มักจะเรียกว่า “Terminal” เฉยๆ)
- เชลล์ (Shell) — โปรแกรมที่อยู่ข้างใน หน้าต่างนั้น ซึ่งอ่านสิ่งที่คุณพิมพ์แล้วรันมัน เชลล์ที่พบบ่อยที่สุดชื่อ bash; เครื่อง Mac ใช้ตัวที่เรียกว่า zsh ทั้งสองตัวทำงานคล้ายกันมากสำหรับทุกอย่างในหนังสือเล่มนี้

นี่คือวงจรที่เกิดขึ้นทุกครั้งที่คุณกด Enter: เชลล์อ่านสิ่งที่คุณพิมพ์ ค้นหาและรันโปรแกรมที่ตรงกัน แล้วพิมพ์ผลลัพธ์ที่โปรแกรมนั้นสร้างขึ้นกลับมาในหน้าต่าง



ทำไมเครื่องมือ AI สำหรับเขียนโค้ดถึงอยู่ที่นี่? เพราะคอมพิวเตอร์ออนไลน์คือแพลตฟอร์มสำหรับการพัฒนาซอฟต์แวร์ การติดตั้งเครื่องมือ การเริ่มแอปของคุณ การรันการทดสอบ การนำขึ้นสู่อินเทอร์เน็ต — ทั้งหมดนี้เกิดขึ้นผ่านคำสั่งที่พิมพ์เข้าไป AI เอะเจตทำงานโดย พิมพ์คำสั่งแบบเดียวกับที่คุณจะพิมพ์ เพียงแต่เร็วกว่าและไม่พิมพ์ผิด เมื่อคุณดู AI ทำงาน คุณกำลังดูมันใช้กระบวนการเดียวกับที่คุณจะใช้เอง

การอ่านพรอมต์

ก่อนที่คุณจะพิมพ์อะไรก็ตาม เชลล์จะแสดง พรอมต์ (prompt) ให้เห็น — ข้อความสั้นๆ ที่ลงท้ายด้วยสัญลักษณ์ รอการป้อนข้อมูล อาจมีหน้าต่างแบบนี้:

```
mini@laptop ~/projects/my-app $
```

อ่านจากซ้ายไปขวา สิ่งนี้บอกคุณว่า: ชื่อผู้ใช้ของคุณ (mini) คอมพิวเตอร์ของคุณ (laptop) ตำแหน่งที่คุณอยู่ตอนนี้ (โฟลเดอร์ ~/projects/my-app) และตามด้วย \$ ซึ่งหมายถึง “ถึงตาคุณแล้ว” ~ เป็นตัวย่อของโฟลเดอร์บ้าน (home folder) ของคุณ สัญลักษณ์ท้ายสุดมักจะเป็น \$ หรือ %; บางครั้งก็เป็น > อย่าพิมพ์สัญลักษณ์นั้นเข้าไป — มันเป็นเพียงเชลล์บอกว่ามันพร้อมแล้ว

เรื่องนี้สำคัญกว่าที่มันดูเหมือน ความสับสนส่วนใหญ่ในการใช้คอมพิวเตอร์ออนไลน์เกิดจากการไม่รู้ว่าคุณอยู่ที่ไหน พรอมต์จะบอกคุณทุกครั้งไป

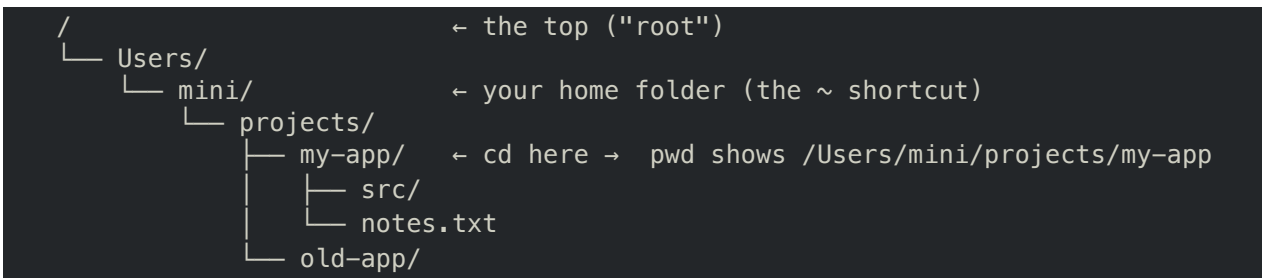
คำสั่งที่ใช้ในชีวิตประจำวัน

ต่อไปนี่คือคำสั่งที่คุณจะเห็นอยู่เป็นประจำ อ่านมันเหมือนอ่านประโยค: คำกริยา บางครั้งตามด้วยเป้าหมาย

```
pwd          # "print working directory" – where am I right now?
ls           # list the files and folders here
ls -la       # list everything, including hidden files, with details
cd projects  # change directory – move into the "projects" folder
cd ..        # move up one folder (".." means "the folder above")
cd ~         # go back to your home folder
mkdir my-app # make a new folder called "my-app"
cp notes.txt backup.txt # copy a file (original stays, copy is made)
mv old.txt new.txt      # move or rename a file
rm draft.txt           # remove (delete) a file
```

มีข้อสังเกตสองสามข้อที่จะช่วยให้คุณรอดพ้นจากปัญหาจริงๆ:

ไฟล์ของคุณอาศัยอยู่ในโครงสร้างต้นไม้ของโฟลเดอร์ที่ซ้อนกัน และ `cd` คือวิธีที่คุณเดินไปในโครงสร้างนั้น นี่คือตัวอย่างเล็กๆ พร้อมกับ `pwd` ที่แสดงว่าคุณจะยืนอยู่ตรงไหนถ้าคุณ `cd` เข้าไปที่ `my-app`:



- **cd** คือวิธีที่คุณเคลื่อนที่ไปมา เมื่อรวมกับ `pwd` และ `ls` แล้ว มันคือ 80% ของสิ่งที่คุณจะทำตลอดไป คุณย้ายเข้าไปในโฟลเดอร์ที่ถูกต้อง มองดูรอบๆ แล้วรีบบางอย่าง
- เส้นทาง (path) อาจเป็นแบบสัมพัทธ์ (relative) หรือแบบสมบูรณ์ (absolute) `cd projects` เป็นแบบสัมพัทธ์กับตำแหน่งที่คุณอยู่; `cd /Users/mini/projects` คือที่อยู่เต็มจากจุดบนสุด
- **rm** ลบแบบถาวร ไม่มีถังขยะ (Recycle Bin หรือ Trash) บนคอมพิวเตอร์ออนไลน์ เมื่อคุณ `rm` ไฟล์ มันหายไปเลย — ไม่มี “แน่ใจหรือไม่?” ไม่มีการยกเลิก ระวังเป็นพิเศษกับ `rm -rf` ซึ่งลบโฟลเดอร์และทุกอย่างข้างในโดยไม่ถาม ถ้าคุณไม่แน่ใจว่าคำสั่ง `rm` จะลบอะไรบ้าง อย่ารันมัน ให้ถาม AI ให้อธิบายก่อน

การรันโปรแกรมและ dev server

การสร้างซอฟต์แวร์หมายถึงการรันมัน ส่วนใหญ่แล้วคุณจะมีพิมพ์ชื่อโปรแกรมตามด้วยสิ่งที่คุณต้องการให้มันทำ

```
node app.js      # run a JavaScript file with Node
python script.py # run a Python file
npm run dev      # start the project's development server
```

คำสั่งสุดท้ายนั้นพิเศษ dev server (เซิร์ฟเวอร์สำหรับพัฒนา) คือโปรแกรมที่รันแอปของคุณบนคอมพิวเตอร์ของคุณเอง เพื่อให้คุณเห็นมันในเบราว์เซอร์ที่อยู่แบบ `http://localhost:3000` สิ่งสำคัญที่ต้องเข้าใจคือ: มัน ทำงานต่อเนื่องไปเรื่อยๆ เทอร์มินัลจะดูเหมือน “ค้าง” — มันจะหยุดให้พร้อมดีใหม่และแสดงกระแสของกิจกรรมแทน นั่นไม่ใช่การค้าง แต่มันกำลังทำงานอยู่ แอปของคุณจะทำงานอยู่ตรงหน้าเท่าที่คำสั่งนั้นยังรันอยู่ เปิดเบราว์เซอร์ของคุณ ไปที่ที่อยู่ที่คุณพิมพ์ออกมา แล้วคุณจะได้เห็นแอปของคุณ

เมื่อเสร็จแล้ว คุณหยุดมัน (ห้วนข้อถัดไป)

การอ่านผลลัพธ์และ exit code

หลังจากคำสั่งรันเสร็จ มันมักจะพิมพ์อะไรบางอย่างออกมา การเรียนรู้ที่จะ อ่าน ผลลัพธ์นั้นเป็นทักษะที่แท้จริง เพราะมันคือวิธีที่คุณและ AI จะรู้ว่าสิ่งต่างๆ ทำงานได้หรือไม่

- ผลลัพธ์ปกติ (Normal output) เป็นเพียงข้อมูล — รายการไฟล์ ข้อความความคืบหน้า เซิร์ฟเวอร์บอกว่ามันเริ่มทำงานแล้ว
- ข้อผิดพลาด (Errors) มักบอกตรงๆ: `Error:`, `command not found`, `permission denied`, `cannot find module` คำที่ตามหลังเครื่องหมายโคลอนคือเบาะแสของคุณ คัดลอกข้อผิดพลาดทั้งหมดแล้ววางให้ AI ของคุณ — นี่คือนิสัยที่มีประโยชน์ที่สุดสิ่งเดียวที่คุณสามารถส่งต่อได้เมื่อมีอะไรผิดพลาด

เบื้องหลังฉาก ทุกคำสั่งจะจบด้วย exit code (รหัสการออก) ที่มองไม่เห็น: 0 หมายถึงสำเร็จ ค่าอื่นใดหมายถึงมีปัญหา คุณแทบไม่จำเป็นต้องตรวจสอบมันด้วยตัวเอง แต่คุณจะได้ยินมันถูกพูดถึง คุณสามารถดูคำสั่งได้แบบนี้:

```
echo $? # prints the exit code of the last command (0 = OK)
```

ถ้าคำสั่งล้มเหลวแบบเจ๊าะแจ๊ะ และคุณไม่แน่ใจ `echo $?` จะบอกคุณว่ามันสำเร็จจริงหรือไม่

การหยุดโปรเซสด้วย Ctrl-C

dev server นั้นจะทำงานต่อไปเรื่อยๆ จนกว่าคุณจะหยุดมัน วิธี “หยุด” สากคือ Ctrl-C — กดปุ่ม Control ค้างไว้แล้วกด C มันจะส่งสัญญาณ “โปรดยุติตอนนี้” อย่างสุภาพไปยังสิ่งที่กำลังรันอยู่ และคุณจะได้พร้อมท์กลับคืนมา

- Ctrl-C — หยุดโปรแกรมที่กำลังรันอยู่ในเทอร์มินัลนี้ในขณะนี้ เป็นตัวเลือกหลักของคุณสำหรับการปิด dev server หรือยกเลิกคำสั่งที่ใช้เวลานานเกินไป
- การปิดหน้าต่างเทอร์มินัล ก็หยุดทกอย่างที่กำลังรันอยู่ในนั้นเช่นกัน แต่ Ctrl-C สะอาดกว่า

จำ Ctrl-C ไว้ แล้วคุณจะไม่มีความรู้สึกติดต่อกับเทอร์มินัลที่ “ค้าง” อีกต่อไป

PATH และ “command not found”

ไม่ช้าก็เร็วคุณจะมีพิกัดคำสั่งแล้วได้รับ command not found สิ่งนี้แทบไม่เคยหมายความว่าคุณโง่; มันหมายความว่าคอมพิวเตอร์ไม่รู้ว่โปรแกรมนั้นอยู่ที่ไหน

เซลล์ค้นหาโปรแกรมโดยการตรวจสอบรายการโพลเดอร์ที่เรียกว่า PATH เมื่อคุณพิมพ์ npm เซลล์จะไล่ตรวจดูแต่ละโพลเดอร์ใน PATH เพื่อหาโปรแกรมชื่อ npm ถ้ามันไม่อยู่ในโพลเดอร์ใดเลย คุณจะได้รับ command not found

```
you type: npm  
          |  
          ▼  
  
search PATH folders, in order, for "npm":  
 /usr/local/bin → not here  
 /usr/bin       → not here  
 /opt/bin       → FOUND ✓ → run it  
  
if none of the folders have it → "command not found"  
  
echo $PATH # show the list of folders the shell searches
```

ในทางปฏิบัติ `command not found` มักหมายถึงหนึ่งในสองอย่าง: เครื่องมือยังไม่ได้ถูกติดตั้ง หรือมันถูกติดตั้งไว้ในที่ที่ไม่อยู่ใน `PATH` ของคุณ ทั้งสองกรณีพบได้บ่อยและแก้ไขได้ วางข้อผิดพลาดให้ AI ของคุณและบอกว่าคุณกำลังพยายามอะไร — การแก้ปัญหา `PATH` เป็นงานตั้งค่าที่ยังยากเล็กน้อยแบบที่ AI ถนัดในการช่วยพาคุณผ่านมันไปได้

การรันคำสั่งที่ AI ให้คุณมา — อย่างปลอดภัย

ส่วนใหญ่ของการทำ vibe coding คือ AI พูดว่า “รันคำสั่งนี้” แล้วยืนบรรทัดหนึ่งให้คุณวาง นี่เป็นเรื่องปกติและไม่เป็นไร แต่จงสร้างนิสัยหนึ่งที่ปกป้องคุณตลอดไป: อ่านก่อนรัน

คุณไม่จำเป็นต้องเข้าใจทุกแฟล็ก (flag) คุณต้องการแค่ความรู้สึกคร่าวๆ ว่าคำสั่งนั้น ทำอะไร โดยเฉพาะคำกริยาที่อยู่
ด้านหน้า:

- ls, pwd, cd, cat, echo — ไม่เป็นอันตราย มันแค่ดูสิ่งต่างๆ หรือเคลื่อนที่ไปมา รันได้อย่างอิสระ
- npm install, pip install — มันดาวน์โหลดและตั้งค่าเครื่องมือ โดยทั่วไปปลอดภัย แต่มันเปลี่ยนแปลงโปรเจกต์ของคุณ
- rm, mv, อะไรก็ตามที่มี sudo, อะไรก็ตามที่ส่งการดาวน์โหลดตรงเข้าไปในเชลล์ของคุณ (curl ... | bash) — สะดวกไว้ก่อน สิ่งเหล่านี้สามารถลบข้อมูลหรือรันโค้ดด้วยอำนาจเต็มเหนือเครื่องของคุณ

ถ้าคำสั่งดูน่ากลัวหรือคุณไม่รู้จักคำกริยานั้น ให้ถามว่า: “คำสั่งนี้ทำอะไร และปลอดภัยที่จะรันหรือไม่?” AI ที่ดีจะอธิบายให้เข้าใจง่าย ๆ อธิบายคำสั่งจากเว็บไซต์สรุปโดยไม่ตรวจสอบก่อนเด็ดขาด คอมมานด์ไลน์ทำตามที่มีคนถูกบอกให้ทำเป๊ะๆ — รวมถึงสิ่งอันตรายด้วย — ดังนั้นนิสัยเล็กๆ ของการอ่านก่อนคือเข็มขัดนิรภัยของคุณ

Package manager ที่คุณจะได้พบ

ซอฟต์แวร์สมัยใหม่ถูกสร้างจากชิ้นส่วนเล็กๆ ที่เขียนไว้ล่วงหน้าจำนวนหลายพัน Package manager คือเครื่องมือที่ดาวน์โหลดและจัดระเบียบชิ้นส่วนเหล่านั้น คุณจะพบสองตัวอยู่เป็นประจำ:

```
npm install           # install everything this JavaScript project needs
npm install react     # add a specific package called "react"
pip install requests  # install a Python package called "requests"
```

- npm (Node Package Manager) คือ package manager สำหรับโปรเจกต์ JavaScript และ Node npm install (มักแสดงเป็น npm i) อ่านไฟล์ที่ระบุรายการสิ่งที่โปรเจกต์ต้องการและดาวน์โหลดทั้งหมด
- pip ทำหน้าที่เดียวกันสำหรับ Python

คุณมักจะรัน npm install ครั้งเดียวตอนที่คุณเปิดโปรเจกต์เป็นครั้งแรก แล้วก็ลืมมันไป เมื่อ AI เพิ่มฟีเจอร์ที่ต้องการ package ใหม่ มันจะบอกให้คุณติดตั้งมัน ตอนนี้คุณรู้แล้วว่ามันหมายถึงอะไร: มันแค่ดึงชิ้นส่วนพื้นฐานมา ไม่ใช่อะไรที่ต้องกลัว

นี่แหละคอมมานด์ไลน์ ไม่ใช่อาวุธของแฮกเกอร์ — เป็นเพียงกล่องข้อความที่ทำตามที่คุณสั่งเป๊ะๆ เรียนรู้ที่จะอ่านพรอมต์ จดจำคำสั่งสักสิบกว่าตัว หยุดโปรเซสที่ควบคุมไม่ได้ด้วย Ctrl-C และอ่านก่อนรัน ส่วนที่เหลือ คุณสามารถถามได้เสมอ

สรุปและแบบฝึกหัด

ประเด็นสำคัญ

- คอมมานด์ไลน์เป็นเพียงกล่องข้อความที่ทำตามสิ่งที่你说บอกมันเป๊ะๆ — ด้วยคำพูดแทนการคลิก AI เอเจนต์ทำงานโดยพิมพ์คำสั่งแบบเดียวกับที่คุณจะพิมพ์
- พรอมต์ บอกคุณว่าคุณอยู่ที่ไหน ทุกครั้งไป; ความสับสนส่วนใหญ่เกิดจากการไม่รู้ว่าคุณอยู่โฟลเดอร์ไหนในตอนนั้น pwd, ls, และ cd คือ 80% ของสิ่งที่你会จะทำตลอดไป
- rm ลบแบบถาวร — ไม่มีถังขยะ ไม่มีการยกเลิก ชะลอไว้ก่อนเมื่อเจอ rm, sudo, และอะไรก็ตามที่ส่งการดาวน์โหลดเข้าไปในเชลล์ของคุณ (curl ... | bash)
- dev server จะทำงานต่อเนื่องไปเรื่อยๆ และดูเหมือน “ค้าง” โดยตั้งใจ — นั่นคือมันกำลังทำงานอยู่ Ctrl-C หยุดโปรแกรมที่กำลังรันอยู่และคืนพรอมต์ให้คุณ
- อ่านก่อนรัน คุณไม่จำเป็นต้องรู้ทุกแฟล็ก แค่รู้คร่าวๆ ว่าคำกริยาคืออะไร เมื่อไม่แน่ใจ ให้ถาม AI ว่า “คำสั่งนี้ทำอะไร และปลอดภัยหรือไม่?”

ลองทำดู

เปิดเทอร์มินัลของคุณและสำรวจโดยไม่เปลี่ยนแปลงอะไร: รัน pwd เพื่อดูว่าคุณอยู่ที่ไหน ls เพื่อดูว่ามีอะไรอยู่ตรงนั้น cd เข้าไปในโฟลเดอร์แล้ว ls อีกครั้ง จากนั้น cd .. เพื่อปีนกลับออกมา สังเกตว่าพรอมต์อัปเดตทุกครั้งอย่างไร ตอนนี้ลองรันคำสั่งที่ล้มเหลวโดยตั้งใจ — พิมพ์ lss (พิมพ์ผิดโดยตั้งใจ) แล้วอ่านข้อความ command not found คุณเพิ่งฝึกใช้คำสั่งที่ปลอดภัยและใช้บ่อยที่สุดสองคำสั่ง และเรียนรู้ที่จะอ่านข้อผิดพลาด ทั้งหมดนี้โดยไม่ได้แตะไฟล์แม้แต่ไฟล์เดียว

พรอมต์ประจำบท

```
I'm a beginner on the command line and a bit nervous about running things.
Here's a command you (or a tutorial) gave me to run:
<paste the command>
```

- Explain in plain English what this command does, step by step.
- Tell me clearly whether it's safe, or whether it could delete files, change my system, or run code with full permissions.
- If it's risky, suggest a safer way to do the same thing, or what to

```
check first.  
- Tell me what folder I should be in before I run it.
```

เว็บทำงานอย่างไร



ก่อนที่คุณจะเริ่มสร้างสิ่งต่าง ๆ บนเว็บ การเข้าใจว่าเว็บคืออะไรจริง ๆ จะช่วยได้มาก คุณไม่จำเป็นต้องมีปริญญาวิทยาการคอมพิวเตอร์ และไม่ต้องท่องจำอะไรเลย สิ่งที่คุณต้องมีคือแบบจำลองความคิด (mental model) ที่ใช้งานได้จริง — ภาพในหัวว่าเกิดอะไรขึ้นตอนที่คุณพิมพ์ที่อยู่เว็บแล้วหน้าเว็บก็ปรากฏขึ้นมา

บทนี้จะช่วยสร้างภาพนั้นให้คุณ ทุกสิ่งที่แอปที่สร้างด้วย vibe coding ทำ ตั้งแต่ฟอร์มล็อกอินไปจนถึงการชำระเงิน ล้วนเป็นเวอร์ชันหนึ่งของการเดินร้านแบบง่าย ๆ เดียวกัน คือคอมพิวเตอร์เครื่องหนึ่งถาม แล้วอีกเครื่องหนึ่งตอบ เมื่อคุณมองเห็นการเดินร้านนี้ได้ชัดเจนแล้ว ส่วนที่เหลือของหนังสือเล่มนี้จะไม่รู้สึกเหมือนเวทมนตร์อีกต่อไป แต่จะรู้สึกเหมือนงานทอประปาธรรมดา

สองตัวละคร: client และ server

เกือบทุกอย่างบนเว็บคือบทสนทนาระหว่างสองบทบาท

- client คือฝ่ายที่ถาม ส่วนใหญ่แล้วก็คือเว็บเบราว์เซอร์ — Chrome, Safari, Firefox — ที่ทำงานอยู่บนโทรศัพท์หรือแล็ปท็อปของคุณ มันคือลูกค้าที่เดินเข้ามาที่เคาน์เตอร์
- server คือฝ่ายที่ตอบ มันคือคอมพิวเตอร์เครื่องหนึ่งที่ตั้งอยู่ในศูนย์ข้อมูล (data center) ที่ไหนสักแห่ง ทำงานตลอดวัน คอยรับคำขอ มันคือครัวที่อยู่หลังเคาน์เตอร์

client ไม่เคยทำหน้าที่ของ server และ server ก็ไม่เคยทำหน้าที่ของ client เบราวเซอร์ถามว่า “ขอหน้าแรกหน่อย” server ตอบว่า “นี่ไง” นั่นคือความสัมพันธ์ทั้งหมด server ไม่ใช้กล่องเวทมนตร์พิเศษอะไร — มันเป็นแค่คอมพิวเตอร์ธรรมดาที่มีหน้าที่รองรับคำถามแล้วส่งคำตอบกลับไป

อีกคำหนึ่งที่คุณจะได้ยินบ่อยมาก คือ request (คำขอที่ client ส่งไป) และ response (คำตอบที่ server ส่งกลับมา) จำสองคำนี้ไว้ให้ขึ้นใจ เพราะเป็นแกนหลักของทุกอย่างที่จะพูดถึงต่อไป

URL คืออะไรกันแน่

URL (Uniform Resource Locator) คือที่อยู่ที่คุณขอเข้าถึง มันดูเหมือนสตริงยาว ๆ เส้นเดียว แต่จริง ๆ แล้วเป็นชิ้นส่วนที่มีป้ายกำกับหลายชิ้นเย็บติดกัน ลองแยกชิ้นนี้ดู:

`https://shop.example.com/products/shoes?size=10`

<code>https</code>	<code>scheme</code>	– how to talk (use HTTPS, the secure way)
<code>shop.example.com</code>	<code>host</code>	– which server to talk to
<code>/products/shoes</code>	<code>path</code>	– which page or thing you want from it
<code>?size=10</code>	<code>query</code>	– extra details ("the size-10 ones")

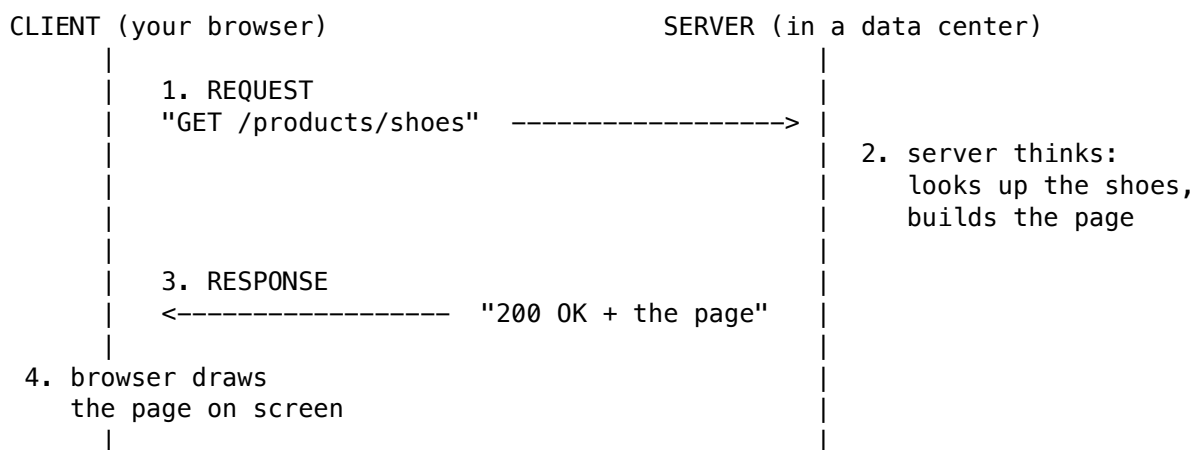
อ่านมันแบบที่อยู่ไปรษณีย์ที่กลับด้าน:

- scheme (https) คือโปรโตคอล — กฎของบทสนทนา ทุกวันนี้แทบจะเป็น https เสมอ
- host (shop.example.com) ระบุว่า server ไหน ที่จะติดต่อด้วย
- path (/products/shoes) ระบุว่า สิ่งไหน บน server นั้นที่คุณต้องการ
- query (?size=10) ที่เป็นตัวเลือกเสริม จะพาพารามิเตอร์เพิ่มเติมมาด้วย เช่น ตัวกรองหรือคำค้นหา

เวลามีคนพูดว่า “endpoint” หรือ “route” พวกเขาแทบจะหมายถึง path เฉพาะบน host เฉพาะเสมอ นั่นคือความหมายทั้งหมดของคำที่ดูน่ากลัวเหล่านั้น

วงจร request และ response

นี่คือการเดินร่า ตั้งแต่ต้นจนจบ คุณพิมพ์ที่อยู่ และเบื้องหลังฉากสิ่งนี้เกิดขึ้นภายในเสี้ยววินาที:



สี่ขั้นตอน ทุกครั้ง:

1. client ส่ง request ไปยัง URL หนึ่ง
2. server ประมวลผล มัน — อาจจะค้นหาบางอย่าง รันตรรกะบางส่วน ตรวจสอบว่าคุณเป็นใคร
3. server ส่ง response กลับมา: สถานะ (สำเร็จหรือไม่) บวกกับเนื้อหาบางอย่าง (หน้าเว็บ รูปภาพ หรือข้อมูลบางอย่าง)
4. client ทำบางอย่างกับ response นั้น — โดยปกติคือวาดมันขึ้นบนหน้าจอของคุณ

การโหลดหน้าเว็บหนึ่งหน้าไม่ใช่ request เดียว — มันคือหลายสิบ request เบราวเซอร์ขอหน้าเว็บ แล้วเห็นว่าต้องการรูปภาพ ฟอนต์ และสคริปต์ จึงยิง request ใหม่ออกไปสำหรับแต่ละอย่าง การเดินร่าแบบเดิม เกิดขึ้นหลายครั้ง และเร็วมาก

HTTP methods: คำกริยา

ทุก request จะพก method — คำกริยาที่บอกว่าคุณต้องการทำอะไร มีหลายแบบ แต่สี่แบบนี้ครอบคลุมแทบทุกอย่าง และเทียบได้ตรง ๆ กับแนวคิดในชีวิตประจำวัน:

- GET — “ขอสิ่งนี้ให้ฉันหน่อย” การอ่าน การดึงข้อมูล การดู การโหลดหน้าเว็บหรือรายการ GET ไม่ควรเปลี่ยนแปลงอะไรเลย มันแค่ดึงข้อมูลออกมา
- POST — “นี่คือสิ่งใหม่” การสร้าง การส่งฟอร์ม การสมัครสมาชิก การโพสต์คอมเมนต์
- PUT — “อัปเดตสิ่งนี้” การเปลี่ยนแปลงสิ่งที่มีอยู่แล้ว เช่นการแก้ไขโปรไฟล์ของคุณ (คุณจะได้ยินคำว่า PATCH ด้วยสำหรับการอัปเดตบางส่วน — แนวคิดเดียวกันแต่ขอบเขตเล็กกว่า)
- DELETE — “ลบสิ่งนี้” ตรงตัวตามชื่อ — การลบข้อมูลหรือไฟล์

ถ้าคุณจำอะไรไม่ได้เลยนอกจากอย่างเดียว จงจำนี้: GET อ่าน, POST สร้าง, PUT อัปเดต, DELETE ลบ เวลาแอปของคุณทำงานผิดปกติ การรู้ว่า method ไหนถูกส่งไปมักจะเป็นเบาะแสแรก

Status codes: มันทำงานได้ไหม

ทุก response จะกลับมาพร้อมกับ status code สามหลัก — คำตัดสินแบบมองปราดเดียวของ server ว่า request นั้นเป็นอย่างไร พวกมันถูกจัดกลุ่มตามหลักแรก:

- 2xx — สำเร็จ มันทำงานได้ 200 OK คือคำตอบธรรมดา “นี่คือสิ่งที่คุณขอ”
- 3xx — เปลี่ยนเส้นทาง (redirect) “ไม่ได้อยู่ที่นี้ ไปดูตรงนั้นแทน” พบได้บ่อยและไม่เป็นอันตราย เบราวเซอร์ของคุณจะตามไปโดยอัตโนมัติ
- 4xx — ตัวคุณ (client) ทำผิดพลาด ตัวที่มีชื่อเสียงที่สุดคือ 404 Not Found — คุณขอ path ที่ไม่มีอยู่จริง (ยังมี 401/403 — คุณยังไม่ได้ล็อกอิน หรือไม่ได้รับอนุญาต)
- 5xx — server ทำผิดพลาด 500 Internal Server Error หมายความว่าโค้ดของ server เองพัง หรือสมมติว่าคุณกำลังจัดการ request ของคุณ

การแยก 4xx/5xx นี้ควรสังเกตตมองไว้ เพราะมันบอกคุณว่า ต้องไปดูตรงไหน:

- 404 หมายความว่า request นั้นผิด — มักจะเป็น URL ที่ผิดหรือ path ที่พิมพ์ผิด ให้อูที่สิ่งที่กำลังถูกขอ
- 500 หมายความว่า server สม request ของคุณไม่มีปัญหา แต่โค้ดฝั่งหลังบ้าน (backend) พัง ให้อูที่ log ของ server ไม่ใช่ URL

เวลามืออะไรพังในแอปที่สร้างด้วย vibe coding คำถามที่ว่า “มันเป็น 4xx หรือ 5xx” คือหนึ่งในคำถามที่ใช้คัดกรองปัญหาได้เร็วที่สุด

status code		
2xx	→ SUCCESS	it worked (200 OK)
3xx	→ REDIRECT	"go look over there" (auto-followed)
4xx	→ CLIENT error	YOUR request was wrong → check the URL/path
5xx	→ SERVER error	the backend broke → check server logs

Headers และ cookies

นอกเหนือจากเนื้อหาหลัก ทุก request และ response จะพก headers ไปด้วย — บันทึกละเอียด ๆ ที่มีป้ายกำกับพร้อมข้อมูลเพิ่มเติม มันเหมือนตัวหนังสือที่เขียนบนซองจดหมาย แยกจากตัวจดหมายข้างใน headers บอกอะไรอย่างเช่นรูปแบบ

ของเนื้อหาคืออะไร (Content-Type: text/html) หรือใครเป็นคนถาม

cookie คือการใช้งาน headers แบบหนึ่งที่เฉพาะเจาะจงและสำคัญ เว็บมีความจำสั้น: โดยปกติแล้วแต่ละ request จะยืนอยู่โดดเดี่ยว และ server จะลืมคุณทันทีที่มันตอบเสร็จ cookie แก้ปัญหานี้ หลังจากคุณล็อกอินแล้ว server จะมอบโทเคนเล็ก ๆ ให้เบราว์เซอร์ของคุณ — นั่นคือ cookie — และเบราว์เซอร์ของคุณจะส่งมันกลับไปโดยอัตโนมัติพร้อมกับทุก request ในอนาคต นั่นคือวิธีที่เว็บไซต์ “จำ” ว่าคุณล็อกอินอยู่ในทุกหน้า ถ้าไม่มี cookie ก็ไม่มีความจำ คุณจะต้องล็อกอินใหม่ในทุกครั้งที่คลิก

HTTPS และไอคอนแม่กุญแจ

คุณเคยเห็นไอคอนแม่กุญแจเล็ก ๆ ในแถบที่อยู่ มันหมายความว่ามีการเชื่อมต่อที่ใช้ HTTPS (HyperText Transfer Protocol Secure) — คือ HTTP (HyperText Transfer Protocol) ที่ถูกห่อด้วยชั้นการเข้ารหัสที่เรียกว่า TLS

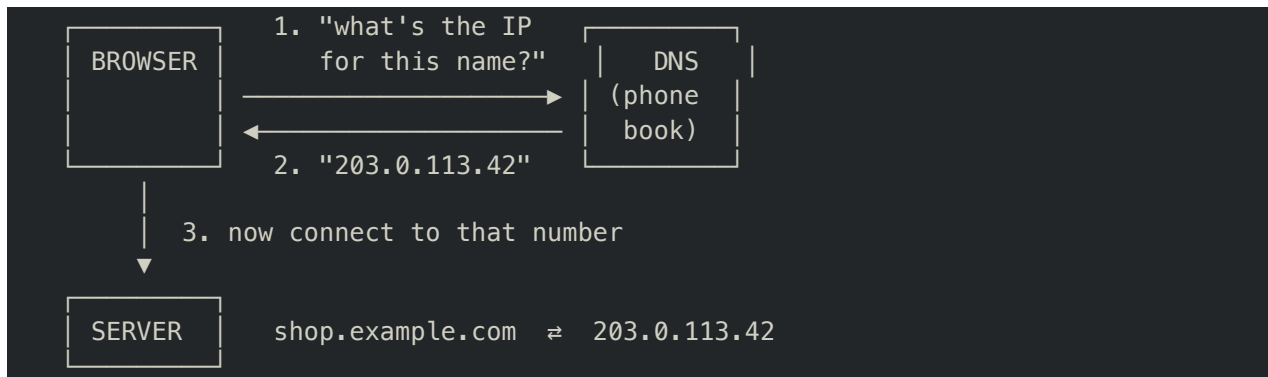
HTTP ธรรมดาส่งทุกอย่างที่เป็นข้อความที่อ่านได้ ใครก็ตามที่อยู่ระหว่างคุณกับ server — เช่นบน Wi-Fi สาธารณะ — สามารถอ่านรหัสผ่านของคุณขณะที่มันวิ่งผ่านไปมาได้ เหมือนโปสการ์ดที่ใครก็ตามในสายส่งไปรษณีย์อ่านได้ HTTPS ผนึกบทสนทนาไว้ในซองจดหมายที่มีแค่คุณกับ server เท่านั้นที่เปิดได้ แม่กุญแจนั้นยืนยันสองสิ่ง:

- ความเป็นส่วนตัว — ไม่มีใครตรงกลางที่จะอ่านสิ่งที่คุณส่งได้
- ตัวตน — คุณกำลังคุยกับ server ที่คุณคิดว่าคุยด้วยจริง ๆ ไม่ใช่ตัวปลอม

กฎในทางปฏิบัติสั้น ๆ คือ: แอปจริงใช้ HTTPS เสมอ ไม่มีข้อยกเว้น อะไรก็ตามที่จัดการเรื่องล็อกอินหรือการชำระเงินผ่าน HTTP ธรรมดา ถือว่าออกแบบผิดพลาดตั้งแต่ต้น

จากชื่อสู่ server: DNS

เหลืออีกเรื่องหนึ่งที่ยังค้างอยู่ คุณพิมพ์ shop.example.com แต่คอมพิวเตอร์ไม่ได้ค้นหาด้วยชื่อจริง ๆ — พวกมันใช้ IP address ที่เป็นตัวเลข เช่น 203.0.113.42 DNS (Domain Name System) คือสมุดโทรศัพท์ของเว็บ: ก่อนที่เบราว์เซอร์ของคุณจะส่ง request ได้ มันจะถาม DNS อย่างเงียบ ๆ เพื่อแปลงชื่อ domain ที่เป็นมิตรให้กลายเป็นที่อยู่ตัวเลข แล้วจึงเชื่อมต่อไปยังที่อยู่นั้น



คุณชื่อ domain หนึ่ง ชี้อะไร DNS ของมันไปยัง server ที่รันแอปของคุณ และจากนั้นเป็นต้นไป ชื่อนั้นก็จะพาผู้คนไปยังเครื่องของคุณ มันเป็นส่วนของการค้นหาที่คุณแทบจะไม่เห็นเลย แต่มันคือเหตุผลที่ชื่อที่จำง่ายสามารถแทนที่ชุดตัวเลขได้

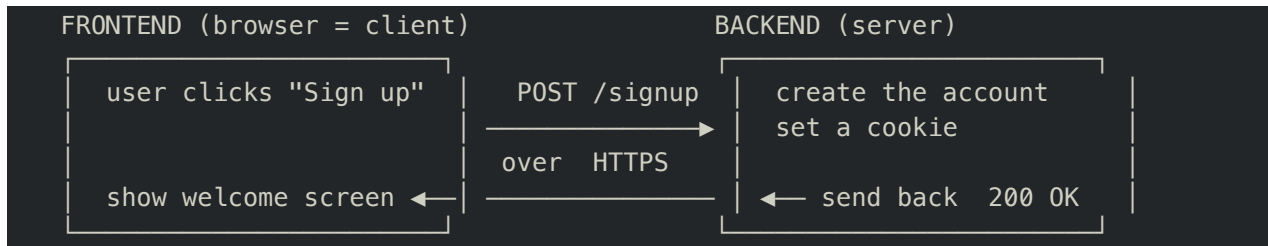
สิ่งนี้เชื่อมโยงกับแอปของคุณอย่างไร

ตอนนี้ลองวางภาพทั้งหมดทับลงบนแอปที่สร้างด้วย vibe coding เพราะทุกชิ้นส่วนที่คุณจะสร้างล้วนอยู่ฝั่งไคลเอนต์หนึ่งของการเดินร้าน:

- frontend คือทุกอย่างที่ทำงานอยู่ในเบราว์เซอร์ — ปุ่ม ฟорм และเลย์เอาต์ที่ผู้ใช้เห็นและคลิก มันคือ client มันส่ง request และแสดง response เป็นส่วนใหญ่

- backend คือโค้ดของคุณที่ทำงานอยู่บน server — ตรรกะที่ตัดสินใจว่าจะทำอะไรกับ request คุยกับฐานข้อมูล และสร้าง response ขึ้นมา มันคือ server

นี่คือการเดินทางทั้งหมดของการสมัครสมาชิกในภาพเดียว — ทุกคำที่ตัวหนาจากบทนี้อยู่ในตำแหน่งของมัน:



ปฏิสัมพันธ์จริงเย็บทุกอย่างเข้าด้วยกัน ผู้ใช้คลิก “สมัครสมาชิก” (frontend) เบราวเซอร์ส่ง request แบบ POST ผ่าน HTTPS ไปยัง path อย่างเช่น /api/signup พร้อมพาข้อมูลฟอร์มไปด้วย backend ของคุณรับมันมา สร้างบัญชี ตั้งค่า cookie เพื่อให้ผู้ใช้อยู่คงที่อีกวัน และส่ง 200 กลับไป frontend เห็นความสำเร็จนั้นแล้วแสดงหน้าจอต้อนรับ ทุกคำที่ตัวหนาตรงนั้นคือสิ่งที่คุณเพิ่งเรียนรู้ไป — และนั่นคือรูปร่างทั้งหมดของแทบทุกอย่างที่คุณจะสร้าง

คุณไม่จำเป็นต้องจำทั้งหมดนี้ไว้ในหัวพร้อมกัน แต่เมื่อบทต่อ ๆ ไปบอกว่า “request ไปถึง server แล้วแต่กลับมาเป็น 500” หรือ “ตั้งค่า header นี้” หรือ “route นี้จัดการ POST” ตอนนี้คุณจะรู้แล้วว่ามันกำลังชี้ไปที่ส่วนไหนของการเดินทางนี้ แบบจำลองความคิดนั้นคือข้อกำหนดเบื้องต้นที่แท้จริง — และตอนนี้คุณมีมันแล้ว

สรุปและแบบฝึกหัด

ประเด็นสำคัญ

- เว็บคือการเดินทางแบบง่าย ๆ เดียวที่เกิดซ้ำไม่รู้จบ: client (โดยปกติคือเบราว์เซอร์) ส่ง request และ server ส่ง response กลับมา ที่เหลือทั้งหมดคือรายละเอียดเพิ่มเติมบนพื้นฐานนี้
- URL คือชิ้นส่วนที่มีป้ายกำกับเย็บติดกัน — scheme, host, path, query method คือคำกริยา: GET อ่าน, POST สร้าง, PUT อัปเดต, DELETE ลบ
- status codes คือคำตัดสินแบบมองปราดเดียวของ server การแยก 4xx/5xx บอกคุณว่าต้องดูตรงไหน: 4xx หมายความว่า request ผิด (URL ผิด) 5xx หมายความว่าโค้ดของ server เองพัง
- HTTPS (แม่กุญแจ) ผนึกบทสนทนาไว้เพื่อไม่ให้ใครตรงกลางอ่านมันได้ — แอปจริงใช้มันเสมอ cookies คือวิธีที่เว็บไซต์จำได้ว่าคุณล็อกอินอยู่ข้ามหลาย request
- frontend ของคุณคือ client (ทำงานในเบราว์เซอร์) backend ของคุณคือ server (รันตรรกะของคุณและคุยกับฐานข้อมูล) ทุกฟีเจอร์ที่คุณสร้างอยู่ฝั่งใดฝั่งหนึ่งของการเดินทางนี้

ลองทำดู

เปิดเว็บไซต์ใดก็ได้ แล้วเปิด developer tools ของเบราว์เซอร์ (คลิกขวา → “Inspect” จากนั้นแท็บ “Network”) แล้วโหลดหน้าใหม่ ดูรายการ request ที่ค่อย ๆ เพิ่มขึ้น — คุณจะเห็น request ของหน้าเว็บหนึ่งครั้ง ตามด้วยอีกหลายสิบครั้งสำหรับรูปภาพ ฟอนต์ และสไตล์ ตรงตามทีอธิบายไว้ทุกประการ คลิกที่หนึ่งรายการแล้วดู method ของมัน (GET) status code ของมัน (หวังว่าจะเป็น 200) และ headers ของมัน ตอนนี้คุณกำลังเห็นการเดินทางของ request/response เกิดขึ้นแบบเรียลไทม์ พร้อมคำศัพท์ที่ใช้อ่านมันได้แล้ว

คำสั่งประจำบท

I'm learning how the web works as a beginner. Using my own app as the example:
<describe your app, or paste a URL or an error you're seeing>

- Walk me through the request/response cycle for one specific action (like loading a page or submitting a form), step by step.
- Tell me which part is the client (frontend) and which is the server (backend).
- If I'm seeing a status code or error, explain whether it's a 4xx (my

request was wrong) or a 5xx (the server broke), and where to look.
– Keep it concrete and beginner-friendly – no assumed knowledge.

API, JSON และข้อมูล



แทบไม่มีแอปไหนที่คุณสร้างขึ้นมาแล้วจะ “อยู่คนเดียว” ได้ ทันทีที่คุณอยากแสดงแผนที่ เก็บเงินผ่านบัตรเครดิต ส่งอีเมล หรือเช็คสภาพอากาศ แอปของคุณต้องคุยกับแอปของคนอื่น ภาษาที่มันใช้พูดคุยกันเรียกว่า API (Application Programming Interface หรือ ส่วนต่อประสานโปรแกรมประยุกต์) และคำพูดที่ส่งไปมาระหว่างกันเกือบทั้งหมดจะเขียนในรูปแบบที่เรียกว่า JSON (JavaScript Object Notation) บทนี้พูดถึงทั้งสองเรื่อง — ไม่ใช่เพื่อให้คุณไปเขียนระบบทอส่งข้อมูลเองด้วยมือ แต่เพื่อให้คุณอ่านมันออก ตรวจสอบสามัญสำนึกได้ว่า AI สร้างอะไรมา และสังเกตได้ว่ามีอะไรผิดพลาดก่อนที่จะมันจะทำให้คุณเสียเงิน

คุณไม่จำเป็นต้องท่องจำอะไรในบทนี้เลย สิ่งที่คุณต้องทำคือจดจำ “รูปทรง” ของมันให้ได้ เมื่อ AI บอกว่า “ผมจะเรียก Stripe API แล้วแปลง JSON ที่ได้กลับมา” คุณควรจะนึกภาพออกคร่าวๆ ว่ามันหมายความว่าอะไร และอะไรที่อาจจะผิดพลาดได้

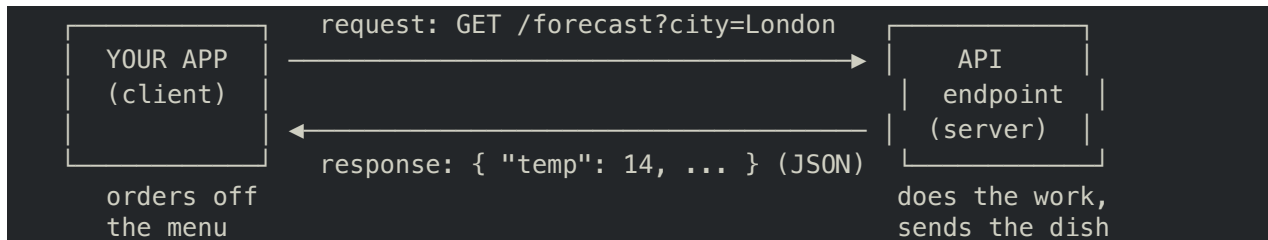
API คืออะไรกันแน่

วิธีที่เข้าใจง่ายที่สุดคือมอง API เป็น เมนูของคำขอที่โปรแกรมหนึ่งเปิดให้อีกโปรแกรมหนึ่งสั่งได้

ร้านอาหารจะไม่ปล่อยให้คุณเดินเข้าไปในครัวแล้วทำอาหารเอง มันจะยื่นเมนูให้คุณแทน: นี่คือสิ่งที่你能สั่งได้ นี่คือนิสั่ง และนี่คือสิ่งที่你能จะได้กลับมา คุณไม่จำเป็นต้องรู้ว่าครัวทำงานอย่างไร คุณแค่ต้องมีเมนู API ก็เป็นแบบนั้นเป๊ะ —

บริษัทหนึ่งบอกว่า “นี่คือคำขอที่คุณสั่งบริการของเราได้ และนี่คือสิ่งที่แต่ละคำขอจะให้กลับมา”

เมื่อ AI เขียนโค้ดที่ “เรียก weather API” มันกำลังสั่งของจากเมนูนั่น: ขอพยากรณ์อากาศของเมืองนี้หน่อย เซิร์ฟเวอร์ของบริษัทพยากรณ์อากาศจะทำงานแล้วส่งคำตอบกลับมา แอปของคุณไม่มีวันเห็นฐานข้อมูลหรือโค้ดของพวกเขา คุณแค่ได้รับ “จาน” ที่เสิร์ฟมา



การเปลี่ยนมุมมองที่สำคัญคือ: API คือ สัญญา บริษัทอีกฝ่ายสัญญาว่าถ้าคุณถามในรูปแบบที่กำหนดไว้ คุณจะค่าตอบในรูปแบบที่กำหนดไว้เช่นกัน หน้าที่คุณ — และของ AI — คือการถามให้ถูกต้องและอ่านคำตอบให้ถูกต้อง บิ๊กของ API ส่วนใหญ่เกิดจากฝ่ายใดฝ่ายหนึ่งทำผิดสัญญานั้น

Endpoint และ REST แบบเข้าใจง่าย

Endpoint คือรายการหนึ่งในเมนู มันเป็นแค่ URL (Uniform Resource Locator หรือ ตัวระบุตำแหน่งทรัพยากรสากล) ที่ทำหน้าที่เฉพาะอย่างหนึ่ง ตัวอย่างเช่น:

- <https://api.weather.com/forecast> — ขอพยากรณ์อากาศ
- <https://api.weather.com/cities> — ดูรายชื่อเมืองที่มีข้อมูล

ครัวเดียวกัน แต่จานอาหารต่างกัน แต่ละ endpoint คือความสามารถหนึ่งอย่าง

API ส่วนใหญ่ที่คุณจะได้เจอทำตามสไตล์กว้างๆ ที่เรียกว่า REST (Representational State Transfer) คุณไม่จำเป็นต้องรู้นิยามทางวิชาการ แต่คุณต้องเข้าใจสองแนวคิดนี้:

- URL ระบุ “สิ่งของ” (เรียกว่า “resource” หรือ ทรัพยากร) — ผู้ใช้คนหนึ่ง คำสั่งซื้อหนึ่งรายการ พยากรณ์อากาศหนึ่งชุด
- คำกริยาบอกว่าคุณอยากทำอะไรกับมัน คำกริยาเหล่านี้คือเมธอดของ HTTP (HyperText Transfer Protocol) ซึ่งเป็นตัวเดียวกับที่มาจากบทเรื่องเว็บทำงานอย่างไร:
 - GET — อ่านข้อมูล (ดึงพยากรณ์อากาศ) ปลอดภัย ไม่เปลี่ยนแปลงอะไร
 - POST — สร้างข้อมูลใหม่ (สั่งซื้อของ สมัครสมาชิกผู้ใช้ใหม่)
 - PUT / PATCH — อัปเดตข้อมูลที่มีอยู่แล้ว
 - DELETE — ลบข้อมูล

ดังนั้น GET /orders/42 หมายถึง “อ่านคำสั่งซื้อหมายเลข 42” และ DELETE /orders/42 หมายถึง “ลบมันทิ้ง” คำานามเดียวกัน คำกริยาต่างกัน ผลลัพธ์ต่างกันโดยสิ้นเชิง เมื่อคุณอ่านเอกสาร API หรือ diff คำสองคำนี้ — เมธอด และ URL — จะบอกคุณเกือบทั้งหมดว่าคำขอนั้นทำอะไร

JSON คืออะไร

เมื่อแอปของคุณถามคำถาม API คำตอบจะกลับมาเป็นข้อความในรูปแบบที่เรียกว่า JSON มันกลายเป็นคำเริ่มต้นเพราะมนุษย์อ่านออก และ โปรแกรมจัดการง่าย คุณจะเจอมันบ่อยมาก จึงคุ้มค่าที่จะอ่านมันให้เป็น

JSON สร้างจาก คีย์และค่า (keys and values) คีย์คือป้ายชื่อในเครื่องหมายคำพูด ส่วนค่าคือข้อมูลที่ผูกอยู่กับคีย์นั้น นี่คือตัวอย่างจริงที่อธิบายผู้ใช้คนหนึ่ง:

```
{
  "id": 42,
  "name": "Ada Lovelace",
}
```

```

"email": "ada@example.com",
"isAdmin": false,
"loginCount": 137,
"tags": ["beta", "early-access"],
"profile": {
  "city": "London",
  "avatarUrl": null
}
}

```

อ่านมันเหมือนแบบฟอร์มที่มีคนกรอกไว้ "name" คือช่องกรอกข้อมูล "Ada Lovelace" คือสิ่งที่เขากรอกลงไป มีโครงสร้างสองแบบที่ทำงานทั้งหมดนี้:

- object (อ็อบเจกต์) ห่อด้วยปีกกา { } มันคือกลุ่มของคู่คีย์/ค่า — เหมือนหนึ่งระเบียบ ตัวอย่างทั้งหมดข้างบนคือหนึ่ง object
- array (อาร์เรย์) ห่อด้วยวงเล็บเหลี่ยม [] มันคือรายการค่าที่เรียงลำดับ ในตัวอย่างข้างบน "tags" เก็บอาร์เรย์ของข้อความสองอัน

สังเกตว่า "profile" เป็น object ที่ซ่อนอยู่ ภายใน object อีกที JSON สามารถซ่อนกันได้ — ค่าหนึ่งค่าอาจเป็น object หรือ array ในตัวมันเองก็ได้ ซ่อนลึกไปได้เท่าที่ต้องการ นี่คือวิธีที่คำตอบเดียวสามารถอธิบายผู้ใช้ ที่อยู่ของเขา และคำสั่งซื้อสินค้ารายการล่าสุดของเขาได้พร้อมกันในคราวเดียว

เมื่อวาดเป็นแผนภูมิต้นไม้ object ผู้ใช้ตัวเดียวกันจะแสดงรูปทรงให้เห็นในทันที — ฟิลด์แบบเรียบๆ อยู่ด้านบน ส่วนลิสต์และ object ที่ซ่อนกันแตกกิ่งออกไป:

```

{ } user (object)
├── id ..... 42 (number)
├── name ..... "Ada Lovelace" (string)
├── isAdmin ..... false (boolean)
├── tags ..... [ ] (array)
│   ├── "beta"
│   └── "early-access"
└── profile ..... { } (object)
    ├── city ..... "London" (string)
    └── avatarUrl . null (null = no value)

```

ชนิดข้อมูลที่คุณจะได้เจอ

ทุกค่าใน JSON เป็นหนึ่งในชนิดข้อมูลกลุ่มเล็กๆ นี้ การรู้จักมันช่วยให้คุณสังเกตได้ว่าค่าหนึ่งเป็นชนิดที่ผิดหรือไม่ — ซึ่งเป็นสาเหตุบั๊กที่พบบ่อย

- string (ข้อความ) — ตัวอักษร อยู่ในเครื่องหมายคำพูดคู่เสมอ: "Ada Lovelace", "ada@example.com"
- number (ตัวเลข) — ตัวเลขล้วนๆ ไม่มีเครื่องหมายคำพูด: 42, 3.14 (เครื่องหมายคำพูดเปลี่ยนความหมาย: "42" คือ ข้อความ สี่-สอง ไม่ใช่ตัวเลข)
- boolean (ค่าตรรกะ) — true หรือ false ไม่มีเครื่องหมายคำพูด ใช้สำหรับระบุใช่/ไม่ใช่ เช่น "isAdmin"
- null — “ไม่มีค่าตรงนี้” ที่ตั้งใจไว้ "avatarUrl": null หมายถึงผู้ใช้ไม่มีรูปประจำตัว — ต่างจากกรณีที่คีย์นั้นหายไปทั้งหมด
- array (อาร์เรย์) — รายการที่เรียงลำดับ: ["beta", "early-access"]
- object (อ็อบเจกต์) — กลุ่มของคู่คีย์/ค่า: บล็อก "profile"

นั่นคือคำศัพท์ทั้งหมด ข้อผิดพลาดที่พบบ่อยที่สุดในโลกจริงคือกับดักข้อความกับตัวเลข: โค้ดคาดหวังตัวเลข 42 แต่ API ส่งข้อความ "42" มา แล้วการเปรียบเทียบก็ล้มเหลวอย่างเงิบๆ เมื่อมีอะไรทำงานแปลกๆ การเช็ค ชนิด ของค่านั้นมักจะเป็นทางแก้

คำขอและคำตอบจริงๆ

นี่คือตัวอย่างการเรียก API จริงตั้งแต่ต้นจนจบ — คำขอที่แอปของคุณส่งไป และ JSON ที่ได้กลับมา:

```
GET /forecast?city=London HTTP/1.1
Host: api.weather.com
Authorization: Bearer sk_live_8f4b2c...
Accept: application/json
```

```
{
  "city": "London",
  "unit": "celsius",
  "current": {
    "temp": 14,
    "condition": "cloudy"
  },
  "forecast": [
    { "day": "Mon", "high": 16, "low": 9 },
    { "day": "Tue", "high": 18, "low": 11 }
  ]
}
```

บรรทัด Authorization นั้นคือหัวข้อถัดไป — และเป็นหัวข้อที่คุ้มค่าที่สุดที่จะทำให้ถูกต้อง

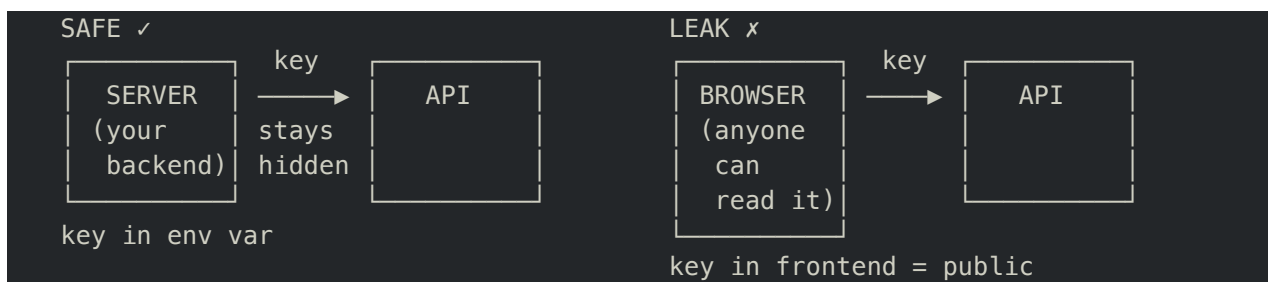
API key และ auth header

API ที่มีประโยชน์ส่วนใหญ่ไม่ได้เปิดให้โลกทั้งใบเข้ามาใช้ได้ฟรีๆ บริษัทต้องรู้ว่า ใคร กำลังถามอยู่ เพื่อจะได้ติดตามการใช้งานและเรียกเก็บเงินคุณ พวกเขาให้ API key แก่คุณ — ข้อความลับยาวๆ ที่ใช้ระบุบัญชีของคุณ คุณต้องแนบมันไปกับทุกคำขอ โดยปกติจะอยู่ใน HTTP header ที่ชื่อว่า Authorization เหมือนบรรทัด Bearer sk_live... ข้างบน

จง treat API key เหมือนรหัสผ่านทุกประการ เพราะมันคือรหัสผ่านนั่นแหละ ใครก็ตามที่มีมันสามารถส่งคำขอ ในนามของคุณ ได้ — และอาจทำให้บิลของคุณพุ่งสูงหรืออ่านข้อมูลของคุณได้ เรื่องนี้เชื่อมโยงตรงกับบทความเรื่องความปลอดภัยและมีกฎข้อหนึ่งที่สำคัญที่สุด:

- ห้ามใส่ API key ในโค้ดที่ส่งไปที่เบราว์เซอร์เด็ดขาด ทุกอย่างใน frontend ของคุณเป็นสาธารณะ — ผู้ใช้เปิด dev tools แล้วอ่านมันได้ ถ้าคุณสั่ง AI ว่า “เรียก API จากหน้าเว็บ” มันจะเอา secret key ของคุณไปวางตรงนั้นอย่างง่ายดาย หยอดมันซะ secret key ต้องอยู่บน เซิร์ฟเวอร์ เท่านั้น
- ห้าม commit key ลง git เด็ดขาด ถึงแม้คุณจะลบมันออกทีหลัง มันจะอยู่ในประวัติตลอดไป และบอทสแกนหา key ใน public repo ภายในไม่กี่นาที
- เก็บ key ไว้ใน environment variable ไม่ใช่ในโค้ดโดยตรง — รูปแบบเดียวกับที่บทความเรื่องความปลอดภัยพูดถึง

Key ต้องเดินทางมาจากที่ปลอดภัยเท่านั้น ไม่ใช่จากเบราว์เซอร์ ลองนึกภาพสองเส้นทาง — เส้นทางปลอดภัย อีกเส้นทางหนึ่งรั่วไหล:



ถ้าคุณเคยเห็น key จริงๆ อยู่ใน diff, ในไฟล์ frontend, หรือในข้อความแชท ให้ถือว่ามันรั่วไหลไปแล้ว: หมุนเปลี่ยน

key ทันที (สร้างอันใหม่ แล้วปิดใช้งานอันเก่า)

Rate limit และกัปปดักบิลช็อก

มีอีกสองเรื่องที่เมนูไม่ได้ป่าวประกาศ แต่มันจะกัปปดักคุณแน่ๆ

Rate limit (การจำกัดอัตราค่าขอ) API จะจำกัดจำนวนคำขอที่คุณส่งได้ในช่วงเวลาหนึ่ง — เช่น 100 ครั้งต่อนาที ถ้าเกินขีดจำกัด คุณจะถูกปฏิเสธด้วย error 429 Too Many Requests จนกว่าคุณจะช้าลง สิ่งนี้มีไว้เพื่อป้องกันไม่ให้ลูกค้านหนึ่งถล่มบริการจนล่ม กับดักคือ: โค้ดที่ AI เขียนมักจะเรียก API ในลูปโดยไม่มีการหยุดพัก ทำให้เกินขีดจำกัดในพริบตา ถ้าพีเจอร์ทหนึ่งทำงานได้กับข้อมูลชิ้นเดียว แต่พังเมื่อรันกับข้อมูลเป็นพันชิ้น rate limit คือผู้ต้องสงสัยอันดับแรก

API แบบวัดปริมาณและมีค่าใช้จ่าย API หลายตัวเรียกเก็บเงินต่อคำขอ — เศษเสี้ยวของเซ็นต์ต่อครั้ง นั่นไม่มีความหมายอะไรสำหรับการเรียกไม่กี่ครั้ง แต่จะกลายเป็นบิลจริงเมื่อบางอย่างวนลูปไม่หยุด เรื่องสยองขวัญคลาสสิก: บั๊กหนึ่งพยายามเรียกซ้ำการเรียกที่เสียเงินซึ่งลุ่มเหลวไม่มีที่สิ้นสุด หรือฟอร์มสาธารณะปล่อยให้คนแปลกหน้าสั่งการเรียกที่เสียเงินได้ แล้วคุณก็ตื่นมาพบใบแจ้งหนี้สี่หลัก ป้องกันตัวเองด้วยวิธีนี้:

- อ่านค่าบริการ ก่อน ที่จะเชื่อมต่อ API ที่เสียเงิน รู้ต้นทุนต่อการเรียกหนึ่งครั้ง
- ตั้ง เพดานค่าใช้จ่ายหรือการแจ้งเตือนงบประมาณ ในแดชบอร์ดของผู้ให้บริการถ้ามีตัวเลือกนี้ ส่วนใหญ่มักจะมี
- ระแวงโค้ดใดๆ ที่เรียก API เสียเงินภายในลูปหรือทุกครั้งที่โหลดหน้าเว็บ

วิธีอ่านเอกสาร API

คุณจะต้องใช้ชีวิตรอยู่กับเอกสาร API ดังนั้นนี่คือวิธีหาสิ่งที่คุณต้องการอย่างรวดเร็วโดยไม่ต้องอ่านทั้งหมด สำหรับ endpoint ไหนก็ตาม ให้มองหาหาลังนี้:

- เมธอดและ URL — GET /forecast มันทำอะไร บอกในสองคำ
- การยืนยันตัวตน (Authentication) — มันต้องการ header หรือ key แบบไหน โดยปกติจะมีหนึ่งส่วนที่ใช้ร่วมกันสำหรับทั้ง API
- พารามิเตอร์ — ข้อมูลนำเข้าที่คุณส่งได้ (ในที่นี้คือ city) ตัวไหนจำเป็น และชนิดของมัน
- ตัวอย่างคำตอบ — รูปทรง JSON ที่คุณจะได้กลับมา นี่คือของดี คัดลอกมันได้เลย
- Error (ข้อผิดพลาด) — สิ่งทีกลับมาเมื่อมีอะไรผิดพลาด และรหัสสถานะ (401 ไม่ได้รับอนุญาต, 404 หาไม่พบ, 429 ถูกจำกัดอัตรา)

เอกสารที่ดีจะมีตัวอย่างคำขอและคำตอบที่คัดลอกวางใช้ได้เลย โค้ดสองกอนั้นบอกคุณได้มากกว่าย่อหน้าข้อความยาวๆ — เก็บมันแล้วส่งให้ AI

ปล่อยให้ AI เรียก API ไป — ขณะที่คุณตรวจสอบรูปทรง

นี่คือการแบ่งงานที่ใช้ได้จริง AI เก่งในการเขียนโค้ดเรียก API จริงๆ มันรู้จักไลบรารี รู้ไวยากรณ์ header รู้การจัดการ error ปล่อยให้มันทำไป คุณไม่ต้องเขียนระบบท่อส่งข้อมูลนี้ด้วยมือ

แต่มีอย่างหนึ่งที่คุณต้องรับผิดชอบเอง เพราะ AI ทำได้ไม่ดี: การรู้ว่ารูปทรงของข้อมูลถูกต้องจริงๆ หรือไม่ ขั้นตอนการทำงานที่ดี:

- ให้เอกสารจริงแก่ AI คัดลอกตัวอย่างคำขอและคำตอบจากเอกสารมาวางในพรอมต์ของคุณ การที่ AI เดารูปทรงของ API จากความจำเป็นสาเหตุก็อันดับต้นๆ — มันจะประดิษฐ์คีย์ที่ดูสมเหตุสมผลแต่ไม่มีอยู่จริงขึ้นมาเอง ตัวอย่างจริงจะช่วยยึดมันไว้กับความเป็นจริง
- ขอให้มันแสดงคำตอบดิบให้คุณดูก่อน “ก่อนที่จะแปลงข้อมูล ให้ log JSON จริงที่ API ส่งกลับมา” แล้วเทียบด้วยตาว่าโค้ดคาดหวังอะไร คีย์ที่มันอ่าน (response.current.temp) มีอยู่จริงในคำตอบหรือไม่ บั๊ก API ครึ่งหนึ่งเกิดจากข้อผิดพลาดตรงกัน
- ตรวจสอบชนิดข้อมูลด้วยสามัญสำนึก มันเป็นตัวเลขหรือข้อความ ฟิลด์ที่ควรจะมีอยู่เสมอบางครั้งเป็น null หรือเปล่า ตอนนี้อยู่ JSON ออกแล้ว — ใช้มันให้เป็นประโยชน์

- จัดการเส้นทางที่ไม่พึงประสงค์ ถามว่า: “จะเกิดอะไรขึ้นถ้าการเรียกนี้ล้มเหลว หมดเวลา หรือโดน rate limit” ถ้าคำตอบคือ “แอปพัง” นั่นคือบั๊ก ไม่ใช่พีเจอร์

คุณไม่จำเป็นต้องเขียนคำขอเอง คุณต้องอ่านคำตอบและถามว่า “นี่คือรูปทรงที่ฉันคาดหวังไว้ไหม” นิสัยเดียวนั้น — การตรวจสอบข้อมูลเทียบกับสัญญา — จะจับบั๊กการเชื่อมต่อส่วนใหญ่ได้ก่อนที่ผู้ใช้ของคุณจะเจอมัน

สรุปและแบบฝึกหัด

ประเด็นสำคัญ

- API คือเมนูของคำขอที่โปรแกรมหนึ่งเปิดให้อีกโปรแกรมหนึ่งสั่งได้ และมันคือ สัญญา: ถามในแบบที่ตกลงกันไว้ ได้คำตอบในรูปทรงที่ตกลงกันไว้ บั๊ก API ส่วนใหญ่เกิดจากฝ่ายใดฝ่ายหนึ่งทำผิดสัญญานั้น
- JSON คือรูปแบบที่คำตอบกลับมา — คีย์และค่า สร้างจาก object { } และ array [] ซึ่งซ่อนกันได้ ชนิดข้อมูลคือ string, number, boolean, null, array, object
- กับดักข้อความกับตัวเลข (42 เทียบกับ "42") และ ค่าที่หายไปเทียบกับ null คือบั๊กรูปทรงที่พบบ่อยที่สุด ตอนนี้คุณอ่าน JSON ออกแล้ว — เช็กชนิดข้อมูลเมื่อมีอะไรทำงานแปลกๆ
- API key คือรหัสผ่าน ห้ามส่งไปที่เบราว์เซอร์ ห้าม commit ลง git เก็บมันไว้ใน environment variable Key ที่รั่วไหลควรถูกหมุนเปลี่ยนทันที
- ระวัง rate limit (429) และ การเรียกเก็บเงินแบบวัดปริมาณ — ลูปที่ AI เขียนขึ้นที่เรียก API เสียเงินโดยไม่หยุดพักคือวิธีที่ทำให้คุณได้ใบแจ้งหนี้สี่หลัก อ่านค่าบริการและตั้งการแจ้งเตือนงบประมาณก่อนเสมอ

ลองทำดู

หา API ฟรีที่ไม่ต้องใช้ key ในเบราว์เซอร์ของคุณ — ตัวอย่างเช่นเปิด <https://api.github.com/users/octocat> โดยตรงในแท็บใหม่ คุณจะเห็นคำตอบ JSON ดิบๆ อ่านมันเหมือนแบบฟอร์ม: หา object { } หา array ถ้ามี และระบุชนิดของค่าสามค่าที่ต่างกัน (อันไหนเป็น string อันไหนเป็น number อันไหนเป็น boolean มีอันไหนเป็น null ไหม) คุณเพิ่งทำการตรวจสอบด้วยสามัญสำนึกแบบเดียวกับที่บทนี้ขอให้คุณทำกับทุกคำตอบ API ที่ AI เชื่อมต่อให้

พร้อมตีประจำบท

I'm calling an API and I want to check the data shape myself, as a beginner. Here are the real docs (example request + example response):
<paste the example request and JSON response from the API's docs>

- Before writing parsing code, show me the raw JSON the call returns and point out the keys, their types, and anything that might be null or missing.
- Confirm the keys my code reads actually exist in that response.
- Make sure the API key is read from an environment variable on the server, never hardcoded or sent to the browser.
- Tell me what happens if the call fails, times out, or hits a rate limit.

กายวิภาคของแอปหนึ่งตัว



เวลาคุณขอ AI ให้ “เพิ่มหน้าล็อกอิน” หรือ “บันทึกข้อมูลนี้ลงฐานข้อมูล” มันกำลังตัดสินใจเรื่องชั้นส่วนที่ขยับได้หลายชั้นอยู่เจียบๆ เบื้องหลัง ถ้าคุณไม่รู้ว่ามีชั้นส่วนเหล่านั้นมีอยู่ คำตอบที่ได้จะอ่านเหมือนภาษาต่างดาว — และคุณจะไม่ออกกว่าคำตอบไหนดีหรือคำตอบไหนสับสน

บทนี้คือแผนที่ มันจะไม่ทำให้คุณกลายเป็นวิศวกร แต่จะทำให้คุณเข้าไปที่ชั้นส่วนไหนของแอปยุคใหม่ก็ได้ แล้วบอกได้ว่ามันทำอะไรและอยู่ตรงไหน พอมีแผนที่แล้ว ทุกบทที่เหลือจะไม่เป็นเรื่องนามธรรมอีกต่อไป คุณจะรู้ว่าปัญหาอยู่ใน “กล่อง” ไหน ก่อนที่จะเริ่มอธิบายมันด้วยซ้ำ เราจะพูดกันแบบเรียบง่าย — ไม่มีศัพท์เทคนิคที่ไม่อธิบายความหมาย และจะใช้ตัวอย่างเดียวตลอดทั้งบท (ผู้ใช้กดปุ่มหนึ่งปุ่ม) แล้วตามรอยมันผ่านทุกชั้น

ภาพรวมใหญ่

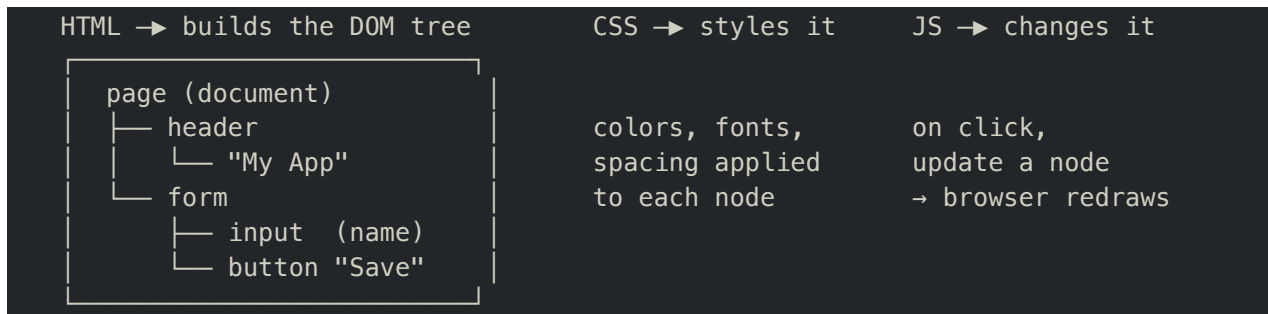
แอปเกือบทุกตัวที่คุณจะสร้างประกอบด้วยสี่ส่วนที่คุยกัน:

- ฟรอนต์เอนด์ (Frontend) — สิ่งที่ใช้เห็นและคลิก หน้าจอ ปุ่ม และข้อความภายในเบราว์เซอร์หรือแอปมือถือ
- แบ็กเอนด์ (Backend) — ตรรกะที่ทำงานอยู่บนเซิร์ฟเวอร์ที่คุณควบคุม มันบังคับใช้กฎ ทำงานจริง และตัดสินใจว่าใครมีสิทธิ์ทำอะไรได้บ้าง
- ฐานข้อมูล (Database) — ที่ที่ข้อมูลอยู่เพื่อให้มันยังอยู่ในวันพรุ่งนี้ บัญชีผู้ใช้ โพสต์ คำสั่งซื้อ
- โฮสติ้ง (Hosting) — คอมพิวเตอร์ ณ ที่ใดที่หนึ่งในโลก ที่รันสิ่งข้างต้นทั้งหมดจริงๆ เพื่อให้คนอื่นเข้าถึงได้

นี่คือวิธีที่พวกมันเชื่อมต่อกัน:

ต่างๆ ได้ มันคือคำกริยา (ภาษาเดียวกันนี้มักจะรันแบ็กเอนด์ด้วย ซึ่งเป็นเหตุผลที่คุณจะเห็นมันอยู่ทุกที่)

วิธีหนึ่งที่จะจินตนาการถึงการแบ่งงาน: HTML สร้างต้นไม้ (DOM), CSS ทาสีมัน และ JavaScript เอื้อมือเข้าไปเปลี่ยนมัน:



แพ็คเกจและดีเพนเดนซี

ไม่มีใครเขียนแอปทั้งตัวขึ้นมาใหม่ทั้งหมด ปัญหาที่ยากและพบบ่อย — การจัดรูปแบบวันที่ การจัดการการชำระเงิน การแสดงปฏิทิน — มีคนอื่นแก้ไขและบรรจุไว้เรียบร้อยแล้ว แพ็คเกจ (package) คือชิ้นส่วนโค้ดที่น่ากลับมาใช้ใหม่ได้ ซึ่งคุณดึงเข้ามาในโปรเจกต์ของคุณแทนที่จะเขียนมันขึ้นมาเอง ดีเพนเดนซี (dependencies) ของโปรเจกต์คุณคือรายการแพ็คเกจที่มันพึ่งพา

ในโลกของ JavaScript เครื่องมือที่จัดการเรื่องนี้เป็น npm (Node Package Manager) มีสองไฟล์ที่ทำหน้าที่บันทึกบัญชี:

- **package.json** — รายการที่มนุษย์อ่านได้ว่าโปรเจกต์ของคุณต้องการอะไรบ้าง บวกกับคำสั่งที่ใช้รันมัน คุณ (หรือ AI) เป็นคนแก้ไขไฟล์นี้
- ไฟล์ล็อก (lockfile) (package-lock.json หรือคล้ายกัน) — บันทึกที่แม่นยำและสร้างขึ้นโดยเครื่องจักรของแพ็คเกจ ทุกตัว และเวอร์ชันที่แน่นอนของมัน เพื่อให้แอปถูกสร้างขึ้นเหมือนกันทุกประการบนเครื่องของคุณ เครื่องของเพื่อนร่วมทีม และเซิร์ฟเวอร์ คุณไม่แก้ไขไฟล์นี้ด้วยมือ คุณปล่อยให้เครื่องมือจัดการมันเอง

```
{
  "name": "my-app",
  "scripts": {
    "dev": "next dev",
    "build": "next build"
  },
  "dependencies": {
    "next": "16.0.0",
    "react": "19.0.0"
  }
}
```

ทำไมถึงต้องสนใจเรื่องนี้? เมื่อ AI บอกว่า “มาเพิ่มไลบรารีสำหรับเรื่องนี้กันเถอะ” มันกำลังแก้ไข package.json และคุณกำลังไว้ใจโค้ดของคนอื่น — ดีเพนเดนซีที่น้อยและเป็นที่ยุติกันนั้นปลอดภัยกว่ากองดีเพนเดนซีที่ไม่ค่อยมีใครรู้จัก และเมื่อการสร้างแอปพังขึ้นมาอย่างลึกลับ ดีเพนเดนซีที่ไม่ตรงกันมักจะเป็นผู้ต้องสงสัยอันดับต้นๆ ซึ่ง “ลบและติดตั้งแพ็คเกจใหม่” เป็นวิธีแก้ที่ใช้ได้จริงและพบบ่อย

ตัวแปรสภาพแวดล้อมและการตั้งค่า

ค่าบางอย่างเขียนลงในโค้ดของคุณไม่ได้: คีย์ลับ รหัสผ่านฐานข้อมูล ที่อยู่ของบริการหนึ่งๆ และค่าบางอย่างต้องเปลี่ยนไปตามที่ที่แอปทำงานอยู่ โค้ดที่คุณทดสอบบนแล็ปท็อป (สภาพแวดล้อม dev) ควรคุยกับฐานข้อมูลทดสอบ; แอปจริงที่ผู้ใช้ของคุณสัมผัส (สภาพแวดล้อม prod หรือ production) ต้องคุยกับฐานข้อมูลจริง — โดยใช้ โค้ดชุดเดียวกัน

คำตอบคือ ตัวแปรสภาพแวดล้อม (environment variables) (มักเรียกว่า “env vars”): ค่าที่มีชื่อซึ่งอยู่ นอก โค้ดและถูกส่งเข้าไปให้มันตอนที่มันทำงาน โค้ดบอกว่า “ขอ DATABASE_URL หน่อย” และสภาพแวดล้อมเป็นคนตัดสินใจว่าจะส่งค่าไหนให้

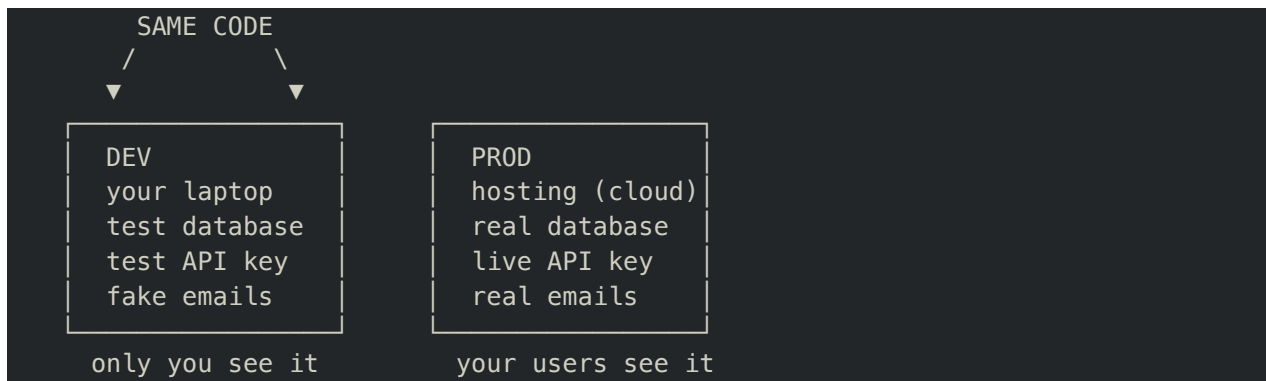
```
# A config file (e.g. .env) – values live here, not in the code

DATABASE_URL=postgres://localhost/myapp_dev
STRIPE_SECRET_KEY=sk_test_51H...      # dev = a test key
SEND_REAL_EMAILS=false                 # don't email real users while testing
```

มีสองกฎที่จะช่วยให้คุณพ้นจากความปวดหัวจริงๆ:

- คีย์ลับต้องไม่อยู่ในโค้ดเด็ดขาด สิ่งใดก็ตามที่อยู่ในโค้ดของคุณอาจจบลงด้วยการเปิดเผยต่อสาธารณะ (โดยเฉพาะในรีโพที่แชร์กัน) และคีย์การชำระเงินหรือรหัสผ่านฐานข้อมูลที่รั่วไหลออกไปคือเหตุการณ์ตัวจริง คีย์ลับควรอยู่ในตัวแปรสภาพแวดล้อม ที่ไม่ถูกเก็บไว้ในระบบควบคุมเวอร์ชัน
- Dev และ prod แตกต่างกัน ประเด็นคือโค้ดชุดเดียวกันทำงานถูกต้องในทั้งสองที่ เพราะสภาพแวดล้อม — ไม่ใช่โค้ด — เป็นคนตัดสินใจว่าจะใช้ฐานข้อมูลและคีย์ตัวไหน

โค้ดเดียวกัน สองบ้าน — ตัวแปรสภาพแวดล้อมคือสิ่งที่แตกต่างกัน:



รูปร่างของโปรเจกต์หนึ่งชิ้น

เปิดโปรเจกต์ทั่วไปดู แล้วรายการไฟล์เดออร์อาจจะรู้สึกเหมือนความอลหม่าน แต่จริงๆ แล้วไม่ใช่ — โปรเจกต์ส่วนใหญ่ทำตามแพตเทิร์นที่จำได้ นี่คือเว็บแอปแบบง่ายๆ:

```
my-app/
├── package.json          # dependencies + run commands (above)
├── package-lock.json     # the lockfile – exact versions
├── .env                  # local secrets/config – NOT shared
├── .gitignore            # files Git should ignore (like .env)
├── README.md             # what this project is, how to run it
├── public/               # static files served as-is (images, icons)
├── src/                  # your actual code lives here
│   ├── components/       # reusable frontend pieces (a button, a card)
│   ├── pages/ (or app/)  # the screens/routes users navigate to
│   ├── lib/              # shared helpers and backend logic
│   └── styles/           # CSS
```

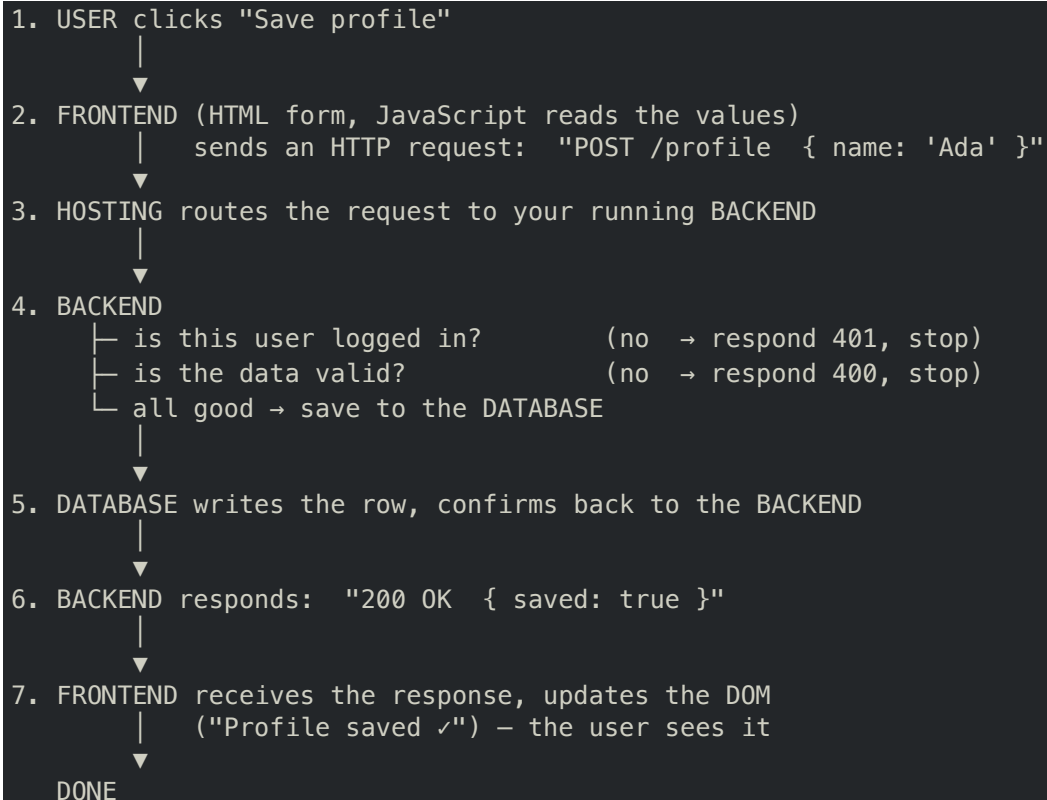
คุณไม่จำเป็นต้องท่องจำสิ่งนี้ — แค่จำได้ว่ามันคืออะไร เพื่อว่าเมื่อ AI บอกว่า “ฉันจะเพิ่มสิ่งนี้ลงใน src/lib” คุณจะรู้คร่าวๆ ว่ามันอยู่ตรงไหนและทำไมถึงอยู่ตรงนั้น มีธรรมเนียมสำคัญบางข้อ:

- **src/** เก็บโค้ดที่คุณเขียน ความรกที่ระดับบนสุดส่วนใหญ่เป็นการตั้งค่า

- ไฟล์การตั้งค่าที่รก (package.json ไฟล์ dotfile ต่างๆ) ตั้งค่าเครื่องมือ ไม่ใช่พฤติกรรมของแอปคุณ
- **.gitignore** ระบุรายการไฟล์ที่ไม่ควร ถูกบันทึกลงในระบบควบคุมเวอร์ชัน — .env ของคุณควรอยู่ในรายการนี้ ซึ่งเป็นวิธีที่ทำให้คีย์ลับอยู่นอกรีโพที่แชร์กันได้พอดี

คำขอไหลผ่านอย่างไร

ที่นี้มาผูกทุกอย่างเข้าด้วยกัน ลองตามรอยผู้ใช้ที่คลิก “บันทึกโปรไฟล์” ผ่านทุกชั้นส่วนที่คุณเพิ่งรู้จักไป นี่คือโมเดลความคิดที่มีประโยชน์ที่สุดเพียงหนึ่งเดียวในหนังสือเล่มนี้ ถ้าคุณจะจำแผนภาพเดียว ให้จำอันนี้



ลองไล่ดูว่าแต่ละชั้นทำอะไรบ้าง: ฟรอนต์เอนด์ รวบรวมข้อมูลนำเข้าและส่งมันไป; โฮสติง ทำให้คำขอไปถึงที่ถูกต้อง; แบ็กเอนด์ บังคับใช้กฎ (ล็อกอิน การตรวจสอบความถูกต้อง) และตัดสินใจ; ฐานข้อมูล ทำให้มันคงอยู่ถาวร; และคำตอบไหลย้อนกลับมาทั้งหมดเพื่อให้ ฟรอนต์เอนด์ อัปเดตหน้าจอได้ ทุกแอปคือการแปรผันของลูปนี้ ที่ถูกรันหลายพันครั้งต่อวินาที

สังเกตว่าการตรวจสอบทั้งหมดเกิดขึ้นในแบ็กเอนด์ ไม่ใช่ในฟรอนต์เอนด์เลย — นี่คือกฎจากส่วนแรกที่กำลังทำงานจริง: ฟรอนต์เอนด์มีไว้เพื่อผู้ใช้ แบ็กเอนด์มีไว้เพื่อความจริง เมื่อคุณรู้ว่าแต่ละชั้นตอนเกิดขึ้น ที่ไหน คุณจะอธิบายฟีเจอร์หนึ่งได้อย่างแม่นยำ (“ตรวจสอบอีเมลที่แบ็กเอนด์ก่อนบันทึก”) แทนที่จะอธิบายอย่างคลุมเครือ (“ทำให้ฟอร์มทำงาน”) — และความแม่นยำคือเกมทั้งหมด

สรุปและแบบฝึกหัด

ประเด็นสำคัญที่ควรจำ

- ทุกแอปประกอบด้วยสี่ส่วน: ฟรอนต์เอนด์ (สิ่งที่ผู้ใช้เห็น), แบ็กเอนด์ (กฎและตรรกะ), ฐานข้อมูล (ที่ที่ข้อมูลคงอยู่), และ โฮสติง (ที่ที่ทุกอย่างทำงาน) ฟรอนต์เอนด์ไม่เคยแตะฐานข้อมูลโดยตรง — มันต้องผ่านแบ็กเอนด์
- ในเบราว์เซอร์ HTML คือโครงสร้าง CSS คือสไตล์ และ JavaScript คือพฤติกรรม; โค้ดเปลี่ยนหน้าเว็บโดยการเปลี่ยน DOM

- คุณไม่ได้เขียนทุกอย่างเอง — แพ็กเกจ (จัดการโดย npm ผ่าน package.json และไฟล์ล็อก) จัดหาปัญหาที่ถูกแก้ไขแล้วในรูปแบบ ดีเฟนเดนซี
- ตัวแปรสภาพแวดล้อม ทำให้คีย์ลับอยู่นอกโค้ด และทำให้โค้ดชุดเดียวกันทำงานถูกต้องทั้งใน dev และ prod
- คำขอไหลจากฟรอนต์เอนด์ → โฮสติ้ง → แบ็กเอนด์ → ฐานข้อมูล แล้วย้อนกลับ; การระบุชื่อชั้นที่ปัญหาอาศัยอยู่คือครึ่งหนึ่งของการแก้ไขมัน

ลองทำดู

เลือกแอปที่คุณใช้ทุกวันสักตัว แล้วเล่าการกระทำหนึ่งอย่างผ่านชั้นต่างๆ ออกมาดังนี้: “ฉันเตะ ส่ง (ฟรอนต์เอนด์) → มันไปที่เซิร์ฟเวอร์ของพวกเขา (แบ็กเอนด์) → ซึ่งตรวจสอบว่าฉันได้รับอนุญาตและบันทึกข้อความ (ฐานข้อมูล) → แล้วเซิร์ฟเวอร์ก็อัปเดต (ฟรอนต์เอนด์)” จากนั้นลองเดาว่าปัญหาแต่ละอย่างอยู่ในชั้นไหน: ปุ่มดูเหมือนจะเลื่อนหลุดตำแหน่ง; แอปบอกว่า “คุณยังไม่ได้ล็อกอิน”; ข้อความจากเมื่อวานหายไป การฝึกฝนการระบุชื่อแบบนี้ให้เก่งคือทักษะที่หนังสือเล่มนี้ทั้งเล่มจะค่อยๆ ขยายออกไป

พรมปีประจําบท

I'm learning how apps fit together. Here is a feature I want:
<one sentence describing what the user should be able to do>

Before writing any code, break it down by layer for me:

- Frontend: what the user sees and what gets sent
- Backend: what rules/checks must run, and where
- Database: what data needs to be stored or read
- Any env vars or config (especially secrets) involved

Then describe the request flow from click to result in numbered steps.
Keep it in plain language — I want to understand the shape before the code.

กรอบความคิดแบบ Vibe Coding



Vibe coding เป็นคำที่มีความหมายซ่อนอยู่หลายชั้น สำหรับบางคนมันหมายถึงการพิมพ์พรอมป์ (prompt) สั้น ๆ บรรทัดเดียว ยอมรับทุกอย่างที่โมเดลปล่อยออกมา แล้วส่งขึ้นใช้งานจริงโดยไม่อ่านโค้ดแม้แต่บรรทัดเดียว นั่นคือเวอร์ชันของเล่น มันผลิตแค่เดโมที่ฟังทันทีที่ผู้ใช้จริงเข้ามาแตะ

หนังสือเล่มนี้พูดถึงอีกเวอร์ชันหนึ่ง คือการสั่งการให้ AI สร้างซอฟต์แวร์ระดับโปรดักชันแบบเดียวกับที่วิศวกรอาวุโสส่งงานที่มของตัวเอง คุณยังคงเป็นคนควบคุมว่า สร้างอะไร และ ทำไมมันถึงถูกต้อง ส่วนที่คุณมอบหมายไปคือการพิมพ์ นั่นคือการเปลี่ยนแปลงที่ยิ่งใหญ่กว่าที่ฟังดู

Vibe coding คืออะไรกันแน่

Vibe coding คือ การสร้างซอฟต์แวร์ด้วยการสื่อความตั้งใจ (intent) เป็นภาษาธรรมชาติ แล้วปล่อยให้ AI สร้างส่วนที่ทำงานจริง (implementation) ในขณะที่คุณคอยกำกับทิศทาง ตรวจสอบ และปรับแก้อย่างต่อเนื่องเป็นวงรอบแน่นๆ

มันไม่ใช่:

- การยอมรับโค้ดที่ถูกสร้างขึ้นโดยไม่เข้าใจมันเลย
- การแทนที่วิจารณ์ญาณทางวิศวกรรมด้วยความรู้สึกล้วนๆ
- ปุ่มพิเศษที่แปลงประโยคหนึ่งประโยคให้กลายเป็นผลิตภัณฑ์ที่พร้อมส่งมอบ

มัน คือ:

- การปฏิบัติต่อโมเดลเหมือนวิศวกรรุ่นน้องที่เร็ว รุ่ยอะ แต่มันใจในตัวเองมากไปหน่อย
- การใช้ความสนใจของคุณไปกับสถาปัตยกรรม ความถูกต้อง และรสนิยม (taste) แทนที่จะไปกับไวยากรณ์

- การทำซ้ำ (iterate) ในเวลาไม่กี่นาทีกับสิ่งที่เคยใช้เวลาหลายชั่วโมง

โมเดลความคิดที่ใช้ได้ผลที่สุดคือ: คุณคือหัวหน้าทีมเทคนิค (tech lead) ส่วน AI คือทีมของคุณ หัวหน้าทีมเทคนิคที่ดีไม่ได้เขียนโค้ดทุกบรรทัดเอง พวกเขากำหนดทิศทาง วางข้อจำกัด ตรวจสอบที่ทีมส่งมาขออนุมัติ (ในวงการซอฟต์แวร์ชุดการเปลี่ยนแปลงที่เสนอมานี้เรียกว่า pull request) และจับโอเคเดียวๆ ก่อนที่มันจะถูกปล่อยออกไป คุณกำลังทำแบบเดียวกัน — แคด้วยความเร็วระดับบทสนทนา

สองเส้นทางนี้เริ่มต้นจากโค้ดที่ถูกสร้างขึ้นชุดเดียวกัน แต่จบลงที่จุดที่ต่างกันมาก:



มีคำหนึ่งในคำนิยามข้างต้นที่แบกน้ำหนักไว้ทั้งหมด นั่นคือ เข้าใจ (understand) แนวทางทั้งหมดนี้ตั้งอยู่บนกฎเพียงข้อเดียว — เข้าใจสิ่งที่คุณส่งมอบ คุณไม่จำเป็นต้องพิมพ์โค้ดเอง แต่คุณต้องอธิบายมันได้ ปกป้องมันได้ในการรีวิว และแก้ไขมันได้ตอนที่สองเมื่อมันปลุกคุณขึ้นมาด้วยการแจ้งเตือน ถ้าคุณทำไม่ได้ แปลว่าคุณไม่ได้ทำ vibe code จริงๆ คุณแค่พนันแล้วโชคดี ช่องว่างระหว่างสองสิ่งนี้คือหัวข้อทั้งหมดของหนังสือเล่มนี้

การเปลี่ยนกรอบความคิด

ถ้าคุณเคยชินกับการพิมพ์ทุกตัวอักษรด้วยตัวเอง ส่วนที่ยากที่สุดไม่ใช่การเรียนรู้เครื่องมือใหม่ๆ แต่คือการปล่อยมือจากคีย์บอร์ด

คอบวเดเดิมของคุณคือ การผลิต — คุณเขียนโค้ดที่ถูกต้องได้เร็วแค่ไหน คอบวเดใหม่ของคุณคือ การกำหนดสเปกและการตรวจสอบ — คุณอธิบายสิ่งที่ต้องการได้ชัดเจนแค่ไหน และคุณตัดสินใจสิ่งที่ได้กลับมาได้ดีแค่ไหน

สิ่งนี้เปลี่ยนสิ่งที่คุณควรเพิ่มประสิทธิภาพให้:

- ก่อนหน้านี้: ท่องจำ API (Application Programming Interface — เมนูคำสั่งที่เครื่องมือหนึ่งเปิดให้ใช้) สู้กับไวยากรณ์ พิมพ์ให้เร็ว
- ตอนนี้: อธิบายความตั้งใจให้แม่นยำ แยกปัญหาออกเป็นส่วยย่อย ประเมินผลลัพธ์อย่างมีวิจารณญาณ

วิศวกรที่รุ่งเรืองในยุคนี้ไม่ใช่คนที่พิมพ์เร็วที่สุด แต่เป็นคนที่มีความเห็นที่ชัดเจนว่า “ดี” หน้าที่เป็นอย่างไร และมีวินัยพอที่จะปฏิเสธผลลัพธ์ที่ไม่ถึงมาตรฐาน

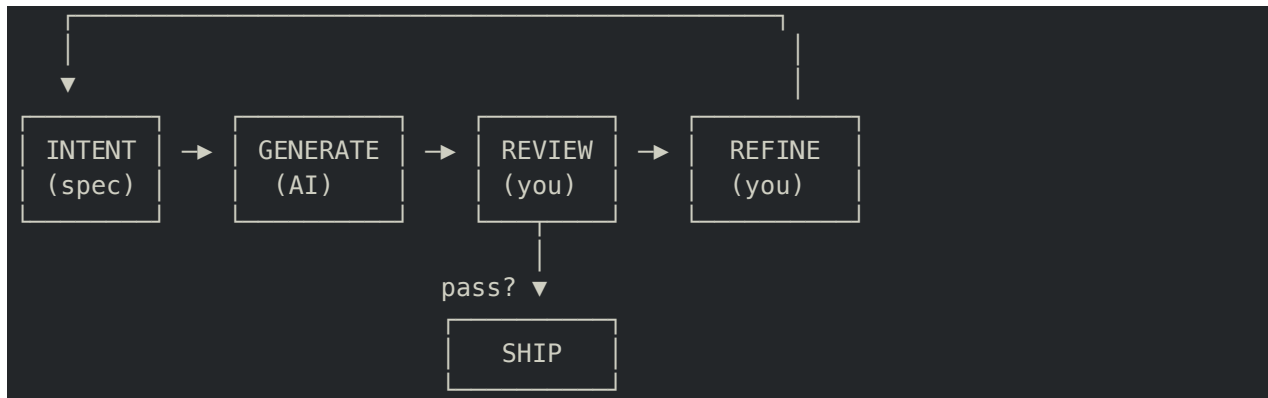
ยังมีมีการเปลี่ยนแปลงที่เจียวกว่าอีกอย่างหนึ่งที่ทำให้คนตั้งตัวไม่ทัน คือความสัมพันธ์ของคุณกับ การไม่รู้ สัญชาตญาณเดิมคือหยุดก่อนแล้วไปเรียนรู้ API ก่อนที่จะแตะมันจริง ตอนนี้คุณมักจะสร้างเวอร์ชันแรกในสแตกที่ไม่คุ้นเคยได้ แล้วเรียนรู้จากการอ่านมันย้อนกลับ นั่นคือพลังพิเศษที่แท้จริง — และก็เป็นที่รักด้วย การอ่านโค้ดที่คุณเขียนเองไม่ได้

ไม่เป็นไร แต่การส่งมอบโค้ดที่คุณ ประเมิน ไม่ได้มันไม่โอเค เส้นแบ่งอยู่ที่ว่าคุณบอกได้อย่างมั่นใจหรือไม่ว่าผลลัพธ์นั้นถูกต้อง พึ่งพาโมเดลให้เขียนได้เร็วกว่าที่คุณได้เสมอ แต่อย่าพึ่งพามันให้รู้มากกว่าที่คุณเต็มใจจะตรวจสอบ

วงรอบ: ความตั้งใจ → สร้าง → ตรวจสอบ → ปรับแก้

ทุกอย่างในหนังสือเล่มนี้วนกลับมาที่วงรอบเดี๋ยวนี จงจำมันให้ขึ้นใจ

วงรอบนี้จะหมุนไปเรื่อยๆ จนกว่าผลลัพธ์จะถึงมาตรฐานของคุณ — และการตรวจสอบคือประตูที่มันต้องผ่านทุกครั้ง:



1. ความตั้งใจ (Intent) ระบุสิ่งที่คุณต้องการและข้อจำกัดที่สำคัญ — อินพุต เอาต์พุต เคสขอบ (edge case) สแตกที่ใช้ สไลด์การเขียน ความตั้งใจที่คลุมเครือจะให้โค้ดที่คลุมเครือ
2. สร้าง (Generate) ปลอมให้โมเดลเขียนส่วนที่ทำงานจริง อย่าจับมือมันเขียนไวยากรณ์ที่ละบรรทัด แต่อธิบายพฤติกรรมที่ต้องการ
3. ตรวจสอบ (Review) อ่านสิ่งที่ได้กลับมา มันทำในสิ่งที่ควรทำจริงหรือไม่ มันจัดการกับลิสต์ว่าง ค่า null หรือการเรียกเครือข่ายที่ล้มเหลวหรือไม่ นี่คือการตรวจสอบขั้นสุดท้ายที่ใช้งานได้แล้วหรือยัง
4. ปรับแก้ (Refine) ชี้จุดที่ผิด ให้การแก้ไขที่เป็นรูปธรรม แล้วสร้างใหม่ ทำซ้ำจนกว่าจะถึงมาตรฐานของคุณ

ผู้เริ่มต้นส่วนใหญ่ข้ามขั้นตอนที่ 3 ไป พวกเขาสร้าง มันรัน แล้วก็ไปต่อ จากนั้นมันก็พังในโปรดักชันด้วยเคสที่ไม่เคยถูกระบุไว้เลย ขั้นตอนการตรวจสอบคือจุดที่งานวิศวกรรมเกิดขึ้นจริง

นี่คือตัวอย่างพรีวิวแรกที่ดีในวงรอบนี้:

```
Write a TypeScript function `parseDuration(input: string): number` that converts strings like "1h30m", "45s", "2d" into total seconds.
```

Requirements:

- Support units: d (days), h (hours), m (minutes), s (seconds).
- Multiple units can combine: "1h30m" = 5400.
- Reject invalid input (empty, unknown units, negative) by throwing an Error with a clear message.
- No external libraries. Include 5 unit tests covering the edge cases.

Show the function and tests only.

สังเกตโครงสร้าง: ลายเซ็นฟังก์ชันที่แม่นยำ กฎที่ชัดเจน เคสขอบที่ถูกระบุออกมา ข้อจำกัดที่ระบุไว้ และคำขอให้ไม่มีเทสต์ด้วย คุณไม่ได้หวังว่าโมเดลจะเดาถูก — คุณกำลังกำจัดความคลุมเครือที่ทำให้เกิดการเดาผิด

ขั้นตอนการปรับแก้เป็นทักษะพอๆ กับพรีวิวแรกเลย ฟีดแบ็กที่คลุมเครือจะได้การแก้ไขที่คลุมเครือ ลองเปรียบเทียบการแก้ไขสองแบบนี้:

Bad: "this is wrong, fix it"

```
Good: "parseDuration('1h30m') returns 90 instead of 5400 — you're summing the numbers but ignoring the unit multipliers. Multiply each value by its unit's seconds (d=86400, h=3600, m=60, s=1) before summing. Also add a test for the combined case '2d3h'."
```

เวอร์ชันที่ตีระบุงการ สาเหตุ และวิธีแก้ คุณกำลังตรวจสอบเหมือนกับที่คุณจะตรวจ pull request ของเพื่อนร่วมงาน — และยังคุณชี้ได้แม่นยำเท่าไร คุณก็ใช้รอบน้อยลงเท่านั้น แต่ละวงรอบควร บรรจบ เข้าหาคำตอบ ถ้าคุณพบว่าตัวเอง กำลังวนอยู่ในที่เดิม นั่นคือสัญญาณว่าความตั้งใจของคุณระบุไว้ไม่ครบ หยุดสร้างใหม่แล้วกลับไปเขียนสเปกใหม่แทน

จุดที่มันเปล่งประกาย

Vibe coding เปลี่ยนแปลงชีวิตได้จริงสำหรับ:

- การสร้างโครงร่างงานใหม่ทั้งหมด (Greenfield scaffolding) การเริ่มแอปใหม่ วางเส้นทาง (route) ตั้งค่าคอนฟิก — งานที่น่าเบื่อแต่มีแบบแผนที่ชัดเจนอยู่แล้ว
- โค้ดสำเร็จรูปและตัวเชื่อมต่อ (Boilerplate and glue) เอนด์พอยต์ CRUD การตรวจสอบฟอร์ม การแปลงข้อมูล ตัวปรับ (adapter) ระหว่าง API สองตัว
- ดินแดนที่ไม่คุ้นเคย ภาษาหรือเฟรมเวิร์กที่คุณไม่คุ้นเคย โมเดลรู้จักสำนวนของมัน คุณให้วิจารณ์ญาณ
- การรีแฟกเตอร์และการย้ายระบบ “แปลง promise พวกนี้เป็น async/await” “แยกไฟล์ 400 บรรทัดนี้ตามหน้าที่ ความรับผิดชอบ”
- การสำรวจแบบทิ้งขว้าง โปรโตไทป์ที่คุณจะเขียนใหม่อยู่ดี ความเร็วสำคัญกว่าความสวยงาม

ในโซนเหล่านี้ พรอมป์ที่ชัดเจนบวกกับการตรวจสอบอย่างระมัดระวังขณะการพิมพ์เองทุกครั้ง

จุดร่วมของสิ่งเหล่านี้คือมันเป็นงานที่มีแบบแผนชัดเจนและตรวจสอบได้ง่าย โมเดลเคยเห็นเอนด์พอยต์ CRUD มาแล้ว นับหมื่น มันจึงสร้างแบบที่เป็นสำนวนถูกต้อง และคุณก็บอกได้แค่มองผ่านๆ ว่าผลลัพธ์ถูกต้องหรือไม่ เมื่อทั้งสองอย่างนี้เป็นจริง — รูปแบบที่พบเจอบ่อย และตรวจสอบได้ถูก — vibe coding ก็เกือบจะเป็นข้อดีล้วนๆ

จุดที่มันลึ่มเหลว

จงชื้อสัตย์เกี่ยวกับข้อจำกัด เพราะนี่คือจุดที่ vibe coding แบบไร้การกำกับดูแลทำให้คนเจ็บตัว:

- ตรรกะที่สลับและแปลกใหม่ อัลกอริทึมที่ละเอียดอ่อน การทำงานพร้อมกัน (concurrency) หรืออะไรก็ตามที่ถูกต้อง 95% หมายถึงพัง โมเดลจะมั่นใจแม้ในตอนที่ไม่ผิด
- การเปลี่ยนแปลงข้ามไฟล์ที่กระจัดกระจาย มันหลุดร่องรอยของบริบทในโค้ดเบสขนาดใหญ่และแอบทำสิ่งที่มันมองไม่เห็นพังไปเจียบๆ
- ความปลอดภัยและเงิน การยืนยันตัวตน (auth) การชำระเงิน สิทธิการเข้าถึง หรืออะไรก็ตามที่บั๊กที่ดูสมเหตุสมผล จะมีผลลัพธ์จริง ตรวจสอบสิ่งเหล่านี้ราวกับว่างานของคุณขึ้นอยู่กับมัน เพราะมันอาจจะเป็นอย่างนั้นจริงๆ
- ปัญหาที่ระบุสเปกไม่ครบ ถ้า คุณ ไม่รู้ว่าความถูกต้องหน้าตาเป็นอย่างไร AI ก็อ่านใจคุณไม่ได้ สเปกขยะเข้า โค้ดขยะออก

รูปแบบความลึ่มเหลวเกือบทุกครั้งเหมือนกัน คือโค้ดที่ ดูเหมือน ถูกต้อง รันผ่านเส้นทางที่ราบรื่น (happy path) และผิดพลาดในแบบที่คุณคงจับได้ถ้าอ่านมัน ทางแก้คือขั้นตอนการตรวจสอบ ไม่มีทางข้ามมันไปได้

คำถามสองข้อนี้จะจัดวางงานใดๆ ก็ตามลงบนแผนที่นี้ — รูปแบบนั้นพบเจอบ่อยหรือไม่ และผลลัพธ์ตรวจสอบได้ง่ายหรือไม่:

	EASY TO VERIFY	HARD TO VERIFY
COMMON PATTERN	PURE UPSIDE (scaffold, boilerplate)	go slower, demand tests

RARE / NOVEL	read it, but doable	DANGER ZONE (novel logic, auth, money)
--------------	---------------------	--

สังเกตด้านตรงข้ามของกฎข้างต้น: นี่คือการที่ หายาก ละเอียดย่อย หรือมีต้นทุนสูงในการตรวจสอบ รูปแบบนั้นไม่ค่อยพบเจอ ดังนั้นโมเดลกำลังเดา หรือ ไม่ก็บ๊ักนั้นมองไม่เห็นบนเส้นทางที่ราบรื่น ดังนั้นการตรวจสอบเร็วๆ จะจับมันไม่ได้ นั่นไม่ได้แปลว่าคุณต้องหลีกเลี่ยง AI ในจุดนี้ — มันหมายความว่าที่คุณต้องช้าลง ย่อขั้นตอนให้เล็กลง เรียกห้องทดสอบ และอ่านทุกบรรทัดราวกับว่ามันเป็นศัตรู

กับดักที่พบบ่อย

มีกับดักไม่กี่อย่างที่ดักเกือบทุกคนในเดือนแรก:

- การยอมรับสิ่งแรกที่รันได้ “มันทำงานได้” กับ “มันถูกต้อง” เป็นคำกล่าวอ้างที่ต่างกัน อย่างแรกหมายความว่าเส้นทางที่ราบรื่นไม่พัง อย่างที่สองหมายความว่าที่คุณตรวจสอบเคลซอบแล้ว อย่าสับสนสองสิ่งนี้
- พรอมป์ขนาดยักษ์ (Mega-prompts) การขอพีเจอร์ทั้งหมดในครั้งเดียวให้พื้นที่โมเดลหลงทางได้มากเกินไป และให้คุณต้องตรวจสอบมากเกินไปในครั้งเดียว จงแต่งงานเป็นชิ้นเล็กๆ ส่งมอบชิ้นที่เล็กที่สุดที่ตรวจสอบได้ก่อน แล้วค่อยต่อยอดจากนั้น
- ปลอ่ยให้บริบทเน่า (context rot) เมื่อเซสชันยาวขึ้นเรื่อยๆ โมเดลจะเริ่มขัดแย้งกับการตัดสินใจก่อนหน้านี้ และนำบ๊ักที่คุณแก้ไปแล้วกลับมาใหม่ เมื่อสิ่งนี้เริ่มเกิดขึ้น จงสรุปสถานะปัจจุบันแล้วเริ่มต้นใหม่ แทนที่จะสู้กับเรดที่สับสน
- เชื่อ ความมั่นใจว่า เป็น สัญญาณ น้ำเสียง ของ โมเดล เหมือน กัน ทุก ประการ ไม่ว่า มัน จะ ถูก หรือ กำลัง หลอน (hallucinate) ความมั่นใจของมันบอกอะไรคุณไม่ได้เลย การตรวจสอบของคุณต่างหากที่บอกทุกอย่าง
- จ้างต่อการตัดสินใจ ไม่ใช่แค่การพิมพ์ การถามว่า “มีตัวเลือกอะไรบ้างสำหรับ X” นั้นไม่เป็นไร แต่การปล่อยให้คำตอบเลือกสถาปัตยกรรมแทนคุณโดยไม่มีการพิจารณาของคุณอยู่ในวงรอบนั้นเป็นความผิดพลาด

รสนิยม (Taste) คือทักษะหลักตัวใหม่

เมื่อใครก็สร้างโค้ดได้ ทักษะที่หายากคือการรู้ว่าโค้ดที่ถูกสร้างขึ้นชิ้นไหน ดี

รสนิยมคือสิ่งที่บอกคุณว่า:

- การแยกส่วน (abstraction) นี้เร็วเกินไป ขอเวอร์ชันที่ง่ายกว่านี้
- อันนี้จัดการกับเดโมได้ แต่จัดการกับอินพุตจริงไม่ได้
- อันนี้ถูกต้องในทางเทคนิค แต่เป็นฝันร้ายสำหรับการบำรุงรักษา
- อันนี้ เกือบ ถูกต้องแล้ว — และนี่คือสิ่งเจาะจงที่ต้องเปลี่ยน

คุณสร้างรสนิยมได้ด้วยวิธีแบบเก่า คือการอ่านโค้ดเยอะๆ ส่งมอบงานจริง แล้วดูว่าอะไรพัง AI ไม่ได้แทนที่ประสบการณ์นั้น — มันทำให้ประสบการณ์นั้นมีความมากขึ้น เพราะตอนนี้วิจารณ์ของคุณคือคอขวดและตัวสร้างความแตกต่าง คนที่มีรสนิยมส่งมอบซอฟต์แวร์ที่มั่นคงได้เร็ว คนที่ไม่มีมันส่งมอบเร็วแล้วก็จมอยู่กับบ๊ัก

นี่คือข้อสรุปที่ไม่ค่อยสบายใจนัก: AI ยกกระดับพื้นฐานให้ทุกคน ซึ่งหมายความว่ามัน ทำให้แบน ข้อได้เปรียบของการรู้ไวยากรณ์ และ ทำให้คมขึ้น ข้อได้เปรียบของการมีวิจารณ์ญาณ วิศกรรุ่นน้องที่พึ่งพามันเพื่อข้ามการเรียนรู้ไปถึงจุดอ้อมตัว เพราะพวกเขาไม่เคยสร้างรสนิยมที่จะบอกพวกเขาได้ว่าเมื่อไหร่ที่ผลลัพธ์ผิด คนที่เชื่อมั่นเพื่อส่งมอบมากขึ้นและอ่านมากขึ้นจะทบต้นขึ้นเรื่อยๆ เครื่องมือเดียวกัน ผลลัพธ์ตรงข้ามกัน — และความแตกต่างอยู่ที่ว่าคุณยังคงมีส่วนร่วมกับการที่ส่งกลับมาหรือไม่

ความคาดหวังที่สมจริง

จงตั้งความคาดหวังเหล่านี้ตั้งแต่ตอนนี้ เพื่อที่คุณจะได้ไม่ผิดหวังในภายหลัง:

- คุณจะต้องอ่านโค้ดเยอะมาก การตรวจสอบคืองานของคุณ ถ้าคุณไม่ยอมอ่านโค้ด นี่ไม่ใช่ทางลัดที่จะเสี่ยงมันได้
- มันจะไม่สำเร็จในครั้งเดียวสำหรับอะไรก็ตามที่ไม่ธรรมดา คาดหวังการวนรอบหลายๆ ครั้ง นั่นเป็นเรื่องปกติ ไม่ใช่ความล้มเหลว
- คุณยังคงรับผิดชอบอยู่ เมื่อโค้ดที่ AI เขียนพังในโปรดักชัน “โมเดลเป็นคนเขียน” ไม่ใช่ข้อแก้ตัว คุณเป็นคนส่งมอบมัน
- มันเร็วขึ้น ไม่ใช่ฟรี เครื่องมือเหล่านี้นำทั้งจริง แต่มันไม่ใช่เวทมนตร์ วิจารณ์ญาณยังคงเป็นของคุณเสมอ

และเพื่อให้ตรงไปตรงมาเกี่ยวกับสิ่งที่หนังสือเล่มนี้ ไม่ใช่: มันไม่ใช่เส้นทางสู่การส่งมอบซอฟต์แวร์ที่คุณไม่เข้าใจ และมันไม่ใช่แผนการรวดเร็ว ใครก็ตามที่ขาย vibe coding ว่าเป็น “สร้างแอปโดยไม่ต้องเรียนเขียนโค้ด” กำลังขายเวอร์ชันของเล่น — เวอร์ชันที่พังทันทีที่สัมผัสกับผู้ใช้จริง ทักษะนี้ทบทวนขึ้นสำหรับคนที่ปฏิบัติต่อมันในฐานะเครื่องทุ่นแรงบนพื้นฐานวิจารณ์ญาณทางวิศวกรรม ไม่ใช่ในฐานะสิ่งทดแทนมัน

คำสัญญาของ vibe coding เป็นเรื่องจริง คุณสร้างได้มากขึ้น เร็วขึ้น ในหลายโดเมนมากกว่าที่คุณจะทำคนเดียวได้ แต่มันให้รางวัลกับวิศวกรที่ยังคงมีส่วนร่วม — คนที่กำกับทิศทาง ตรวจสอบ และปรับแก้ — และลงโทษคนที่ถอนตัวออกไปและยอมรับทุกสิ่งที่ปรากฏขึ้น

จงเป็นหัวหน้าทีมเทคนิค ไม่ใช่ผู้ชม ส่วนที่เหลือของหนังสือเล่มนี้คือวิธีการ

สรุปและแบบฝึกหัด

ประเด็นสำคัญ

- Vibe coding คือการสั่งการให้ AI สร้างซอฟต์แวร์ในขณะที่คุณยังคงควบคุมว่า อะไร และ ทำไม — ไม่ใช่การส่งมอบโค้ดที่คุณอธิบายไม่ได้โดยไม่ผ่านการตรวจสอบ
- กฎที่ยืดทุกอย่างเข้าด้วยกันคือ เข้าใจสิ่งที่คุณส่งมอบ คุณไม่จำเป็นต้องพิมพ์มันเอง แต่คุณต้องปกป้องมันและแก้ไขมันได้
- คอขวดของคุณย้ายจากการผลิต (พิมพ์เร็ว) ไปสู่การกำหนดสเปกและการตรวจสอบ (อธิบายชัดเจน ตัดสินอย่างมีวิจารณ์ญาณ)
- ใช้ชีวิตอยู่ในวงรอบ — ความตั้งใจ → สร้าง → ตรวจสอบ → ปรับแก้ — และอย่าข้ามขั้นตอนการตรวจสอบเด็ดขาด เพราะนั่นคือจุดที่งานวิศวกรรมที่แท้จริงเกิดขึ้น
- ความมั่นใจของโมเดลไม่ใช่สัญญาณ การตรวจสอบของคุณต่างหากที่เป็นสัญญาณ รสนิยมและวิจารณ์ญาณคือทักษะที่หายากและทบทวนได้

ลองทำดู

เลือกฟังก์ชันเล็กๆ ที่คุณเข้าใจอย่างแท้จริง — อย่างเช่น “ตรวจสอบว่ารหัสผ่านตรงตามนโยบายหรือไม่” หรือ “จัดรูปแบบจำนวนไบต์ให้เป็นข้อความที่มนุษย์อ่านได้” เขียนพรมที่แม่นยำสำหรับมัน (ลายเซ็นฟังก์ชัน กฎ เคสขอบ ค่าขอให้มีเทสต์) สร้างมันขึ้นมา แล้วจงตั้งใจทำขั้นตอน ตรวจสอบ จริงๆ คือไล่ตามเคสขอบอย่างน้อยหนึ่งเคสด้วยมือและยืนยันว่าโค้ดจัดการมันได้ ถ้ามันจัดการไม่ได้ จงเขียนพรมที่ปรับแก้ที่แม่นยำ (อาการ + สาเหตุ + วิธีแก้) แทนที่จะพูดแค่ “แก้ไขหน่อย” เป้าหมายคือการรู้สึกถึงวงรอบทั้งหมดครั้งหนึ่งอย่างครบถ้วน บนโค้ดที่คุณสามารถประเมินได้อย่างเต็มที่

พรมปีประจำบท

```
Act as my pair-programming partner. I'll describe a small, self-contained function. Before you write any code, restate the requirements and list the edge cases you'll handle so I can confirm them. Then write the function plus unit tests covering those edge cases.
```

```
Function: <one-line description>
```

```
Inputs / outputs: <types and shapes>
```

```
Constraints: <stack, libraries to use or avoid, performance limits>
```

```
After the code, point out any assumption you made that I should verify.
```

```
Keep it to the simplest version that satisfies the requirements.
```


การตั้งค่าพื้นที่ทำงาน AI ของคุณ



การเขียนโค้ดแบบ vibe coding คือความร่วมมือ คุณนำเจตนาและวิจารณ์ญาณมา ส่วน AI นำความเร็วและการพิมพ์ที่ไม่รู้จักเหน็ดเหนื่อย แต่เหมือนความร่วมมือใด ๆ มันจะทำงานได้ดีที่สุดเมื่อมีสภาพแวดล้อมร่วมกัน กฎเกณฑ์ที่ชัดเจน และพื้นที่ปลอดภัยสำหรับทำผิดพลาด บทนี้ว่าด้วยการสร้างสภาพแวดล้อมนั้นก่อนที่คุณจะเขียนพีเออร์แม้นแต่ตัวเดียว

พื้นที่ทำงานที่ดีทำสามอย่าง คือให้บริบทที่ AI ต้องใช้ในการเปลี่ยนแปลงอย่างชาญฉลาด ให้คุณเห็นผลลัพธ์ของการเปลี่ยนแปลงนั้นได้อย่างรวดเร็ว และทำให้ทุกการเปลี่ยนแปลงยกเลิกได้ง่าย ทำสิ่งเหล่านี้ให้ถูกต้อง แล้ว AI จะรู้สึกเหมือนเพื่อนร่วมงานอาวุโส ทำผิด แล้วมันจะรู้สึกเหมือนเครื่องสล็อต ความแตกต่างแทบไม่ใช่ที่ตัวโมเดล แต่เป็นห้องที่คุณวางมันไว้ต่างหาก เอเจนต์ที่มีความสามารถแต่อยู่ในรีโพที่วุ่นวาย ไม่มีเทสต์ ไม่มีโฟลว์บริบท และไม่มีการควบคุมเวอร์ชัน จะดันรนไปมา ในขณะที่เอเจนต์ธรรมดา ๆ ในโปรเจกต์ที่สะอาดและมีเครื่องมือวัดผลครบ กลับสร้างความประหลาดใจให้คุณได้ การตั้งค่าคือตัวคุณ

ไม่มีสิ่งใดในนี้เป็นงานที่ทำครั้งเดียวแล้วลืม จงปฏิบัติต่อพื้นที่ทำงานของคุณเหมือนสิ่งที่ต้องปรับจูนอย่างต่อเนื่อง ทุกครั้งที่ AI สร้างความประหลาดใจให้คุณในทางที่ไม่ดี ให้ถามตัวเองว่าเป็นเพราะขาดบริบทบางส่วน วงจรป้อนกลับที่ซ้ำหรือค่าเริ่มต้นที่ไม่ปลอดภัยหรือเปล่า แล้วแก้ที่ห้อง ไม่ใช่แค่แก้บรรทัดโค้ด

เลือกเอดิเตอร์หรือผู้ช่วยที่ออกแบบมาเพื่อ AI

การตัดสินใจแรกของคุณคือ AI จะอยู่ที่ไหน ปัจจุบันมีอยู่ราวสามหมวดหมู่ และคุณไม่จำเป็นต้องผูกมัดตลอดไป

- เอดิเตอร์ที่ออกแบบมาเพื่อ AI โดยเฉพาะ (Cursor, Windsurf) คือเอดิเตอร์โค้ดที่คุ้นเคยซึ่งเชื่อมต่อ AI เข้าไปอย่างลึกซึ้ง เหมาะที่สุดหากคุณอยากเห็นและแตะต้องโค้ดขณะที่ AI ทำงานเคียงข้างคุณ

- เอเจนต์แบบเทอร์มินัล/CLI (Claude Code, Codex CLI, Gemini CLI, Aider) ลองนึกถึงการพิมพ์คำสั่งที่พร้อมดีแทนการคลิกปุ่ม แผงควบคุมแบบข้อความล้วนนั้นคือ command line หรือ CLI (Command-Line Interface — อินเทอร์เฟซบรรทัดคำสั่ง) AI ทำงานเป็นเอเจนต์บรรทัดคำสั่งที่อ่านไฟล์ แก้ไขไฟล์ และรันคำสั่งต่าง ๆ เหมาะที่สุดสำหรับปล่อยให้ AI ทำงานอย่างอิสระมากขึ้นในหลายไฟล์พร้อมกัน
- ปลั๊กอินสำหรับ IDE (GitHub Copilot, Continue, JetBrains AI) IDE (Integrated Development Environment — สภาพแวดล้อมพัฒนาแบบครบวงจร) คือแอปแบบครบวงจรที่โปรแกรมเมอร์ส่วนใหญ่ใช้เขียนโค้ด ปลั๊กอินเหล่านี้คือตัวช่วยที่ติดเสริมเข้าไปในเอดิเตอร์ที่คุณใช้อยู่แล้ว

สำหรับการส่งมอบซอฟต์แวร์จริง ให้เลือกเครื่องมือที่สามารถ อ่านทั้งโปรเจกต์ของคุณ แก้ไขหลายไฟล์ และรันคำสั่ง (เทสต์ บิลด์ ลินเตอร์) ได้ด้วยตัวเอง ความสามารถในการรันและตรวจสอบงานของตัวเองคือสิ่งที่แยกผู้ช่วยที่มีประโยชน์จริงออกจากของเล่นเติมคำอัตโนมัติ เครื่องมือที่ทำได้แค่เสนอบรรทัดถัดไปทำให้คุณต้องตรวจสอบทุกอย่างด้วยมือ ส่วนเครื่องมือที่รันเทสต์ที่มันเพ่งแก้ไขได้เองจะปิดวงจรให้เสร็จสมบูรณ์ในตัว

นอกเหนือจากความสามารถหลักนี้ ยังมีเรื่องอื่น ๆ ที่ควรพิจารณาก่อนตัดสินใจเลือก

- หน้าต่างบริบทและความรับรู้เกี่ยวกับโปรเจกต์ — มันโหลดไฟล์ที่เกี่ยวข้องได้จริงหรือไม่ หรือมันลืมนจุดเริ่มต้นของบทสนทนาไปเสียก่อนที่งานจะเสร็จครึ่งทาง
- โมเดลสิทธิการอนุญาต — มันถามก่อนรันคำสั่งและแก้ไขไฟล์หรือไม่ หรือลงมือทำก่อนแล้วค่อยว่า ทั้งสองแบบต่างก็มีที่ทางของมัน ขอแค่รู้ว่าตอนนี้คุณอยู่ในโหมดไหน
- ต้นทุนและขีดจำกัด — เอเจนต์ที่ทำงานอัตโนมัติสามารถใช้โควตาหมดไปอย่างรวดเร็ว ทำความเข้าใจโครงสร้างราคาก่อนที่จะปล่อยให้มันทำงานเป็นชั่วโมง

อย่ากังวลกับการเลือกมากเกินไป ทักษะในหนังสือเล่มนี้ใช้ข้ามเครื่องมือได้ ผู้ที่เขียนโค้ดแบบ vibe coding มาอย่างช่ำชองส่วนใหญ่มักลงเอยด้วยการติดตั้งเครื่องมือสองแบบไว้ คือเอเจนต์เทอร์มินัลที่รวดเร็วสำหรับงานปริมาณมาก และเอดิเตอร์แบบ GUI สำหรับตรวจทาน GUI (Graphical User Interface — อินเทอร์เฟซผู้ใช้แบบกราฟิก) คือโปรแกรมแบบคลิกหน้าต่างและปุ่มที่คุ้นเคย ส่วน TUI (Text User Interface — อินเทอร์เฟซผู้ใช้แบบข้อความ) คือลูกพี่ลูกน้องที่เป็นข้อความล้วนซึ่งอยู่ในเทอร์มินัล ส่วนปิดท้ายของบทนี้ว่าด้วย TUI กับ GUI จะกลับมาพูดถึงการจับคู่แบบนี้อีกครั้ง

จัดโครงสร้างโปรเจกต์ของคุณให้ AI สามารถทำได้

โมเดล AI ให้เหตุผลเกี่ยวกับโค้ดของคุณโดยการอ่านมัน โปรเจกต์ที่ยิ่งใหญ่ ขยายไม่มีทิศทาง และไม่สม่ำเสมอ นั้นยากสำหรับมันพอ ๆ กับที่ยากสำหรับคุณ คุณไม่จำเป็นต้องมีสถาปัตยกรรมที่สมบูรณ์แบบ แต่นิสัยไม่กี่อย่างจะให้ผลตอบแทนทันที

- รักษาผังโครงสร้างที่คาดเดาได้ ซอร์สโค้ดอยู่ทีเดียว เทสต์อยู่ข้าง ๆ หรือสะท้อนโครงสร้างของซอร์ส คอนฟิกรูที่รูท
- ใช้ชื่อที่ชัดเจนและสื่อความหมาย `calculateMonthlyInvoiceTotal` ดีกว่า `calc2` AI ใช้ชื่อเป็นเบาะแส
- เลือกไฟล์ที่เล็กกว่า ไฟล์ 200 บรรทัดแก้ไขได้ถูกต้องง่ายกว่าไฟล์ 2,000 บรรทัด ทั้งสำหรับคุณและสำหรับมัน
- จัดวางสิ่งที่เกี่ยวข้องกันไว้ด้วยกัน เมื่อโค้ด สไตล์ และเทสต์ของฟีเจอร์หนึ่งอยู่ด้วยกัน AI จะรวบรวมบริบทได้ในการอ่านครั้งเดียว
- มีจุดเริ่มต้นที่ชัดเจน จุดเดียวที่ชัดเจนซึ่งแอปเริ่มทำงาน จะให้ AI มีเส้นด้ายให้ดึงเมื่อมันกำลังทำแผนที่โปรเจกต์

ผังโครงสร้างทั่วไปที่เป็นมิตรกับ AI อาจมีลักษณะดังนี้

```
my-app/
├── AGENTS.md           # project rules & context for the AI
├── README.md           # what the project is, how to run it
├── .env.example        # documents needed secrets (no real values)
├── .gitignore          # excludes .env, build output, node_modules
├── package.json        # scripts: dev, test, lint, build
├── src/
│   ├── features/
│   │   └── invoices/   # code + tests for one feature, together
```

```

├── lib/           # shared helpers
├── index.ts       # entry point
└── tests/         # cross-cutting / integration tests

```

ผลตอบแทนจะทวีคูณ ยิ่งโปรเจกต์ของคุณให้ AI สืบหาได้ง่ายเท่าไร การเปลี่ยนแปลงของมันก็จะยิ่งเล็กและแม่นยำมากขึ้นเท่านั้น ริโพที่จัดระเบียบดีทำให้ AI แตะเพียงสามไฟล์ที่เกี่ยวข้องแทนที่จะเดาสุ่มในสามสิบไฟล์ ซึ่งหมายถึง diff ที่เล็กลง อุบัติเหตุน้อยลง และการรีวิวที่คุณอ่านได้จริง หากเคยเจอ AI แก้ไขแบบกว้างขวางและไม่โฟกัส บ่อยครั้งโครงสร้างต่างหากคือตัวการที่แท้จริง ไม่ใช่พรอมต์

เขียนไฟล์บริบท (AGENTS.md)

สิ่งเดียวที่ให้ผลตอบแทนสูงสุดที่คุณทำได้คือการเขียน ไฟล์บริบท ที่ AI อ่านโดยอัตโนมัติ เครื่องมือต่างกันมองหาชื่อไฟล์ต่างกัน — AGENTS.md, CLAUDE.md, .cursorrules หรือ .github/copilot-instructions.md — แต่แนวคิดเหมือนกันทุกประการ นั่นคือเอกสารที่มีชีวิตซึ่งบอก AI ว่าโปรเจกต์ ของคุณ ทำงานอย่างไร และคุณอยากให้มันประพฤติตัวอย่างไร

หากไม่มีไฟล์นี้ AI จะต้องเดา หากมี AI จะเลือกจัดรูปแบบโค้ดของคุณใหม่ เลิกคิดค้นไลบรารีที่คุณไม่ได้ใช้ และเลิกอธิบายซ้ำถึงการตัดสินใจที่คุณทำไปแล้ว ลองนึกภาพมันเป็นเอกสารปฐมนิเทศสำหรับเพื่อนร่วมทีมใหม่ที่ฉลาดล้ำแต่ความจำเสื่อมโดยสิ้นเชิงระหว่างแต่ละเซสชัน — ทุกสิ่งที่พวกเขาต้องรู้เพื่อทำงานได้ผลต้องอยู่ในที่ที่พวกเขาจะอ่านทุกครั้ง

ตัวอย่างไฟล์เริ่มต้นที่ใช้งานได้จริง

```

# Project: Invoice Manager

## What this is
A small web app for freelancers to create and track invoices.

## Tech stack
- TypeScript + React (Vite)
- Tailwind for styling
- SQLite via Prisma
- Vitest for tests

## Commands
- Install: `npm install`
- Run dev server: `npm run dev`
- Run tests: `npm test`
- Lint: `npm run lint`

## Conventions
- Use functional components and hooks; no class components.
- Keep components small; one component per file.
- Write a test for every new function in `src/lib`.
- Never edit files in `src/generated/` — they are auto-generated.

## Don'ts
- Don't add new dependencies without flagging it first.
- Don't commit secrets; use environment variables.

```

รักษาไฟล์ให้สั้นและทันสมัยอยู่เสมอ ไฟล์บริบทที่ยาวขึ้นจนถึงพันบรรทัดจะไม่ถูกอ่านอย่างละเอียดอีกต่อไป ไม่ว่าจะโดยคุณหรือโดยโมเดล ดังนั้นจงตัดทอนมันอย่างระมัดระวังมากขึ้น ๆ กับที่คุณเพิ่มเติมมันเข้าไป เมื่อ AI สันนิษฐานผิดซ้ำ ๆ เรื่องเดิม นั่นคือสัญญาณให้คุณเพิ่มบรรทัดหนึ่ง เมื่อกฎข้อหนึ่งไม่ตรงกับความเป็นจริงอีกต่อไป จงลบมันก่อนที่มันจะเริ่ม

ชักนำเอเจนต์ให้เข้าใจผิดอย่างจริงจัง

มีรูปแบบไม่ก๊อปปี้ที่แยกไฟล์บริบทที่ใช้งานได้ผลออกจากไฟล์ที่ถูกเมีน

- เจาะจงและเป็นรูปธรรม “ใช้ตามธรรมเนียมของเรา” นั้นไร้ประโยชน์ “ใช้ zod สำหรับการตรวจสอบข้อมูล เราไม่ใช่ yup” คือกฎที่ AI ทำตามได้จริง
- ระบุสิ่งที่ห้ามทำให้ชัดเจนพอ ๆ กับสิ่งที่ควรทำ โมเดลกระตือรือร้นที่จะช่วยเหลือ ซึ่งหมายความว่ามันจะเพิ่มดีเพนเดนซีหรือเขียนคอนฟิกใหม่อย่างเต็มใจ เว้นแต่คุณจะบอกไม่ให้ทำ
- ชี้ไปยังตัวอย่าง “ทำตามสไตล์ของ src/features/invoices” ให้โมเดลที่เป็นรูปธรรมให้ AI ลอกเลียนแทนที่จะเป็นคำสั่งกำกวมที่ต้องตีความเอง

จงปฏิบัติต่อไฟล์บริบทเหมือนบันทึกของบทเรียนทุกอย่างที่คุณไม่อยากพูดซ้ำ มันคือการลงทุนที่ถูกที่สุดและยั่งยืนที่สุดในทั้งบทนี้

ทำให้การควบคุมเวอร์ชันเป็นปฏิกิริยาสะท้อน

การควบคุมเวอร์ชันคือตาข่ายนิรภัยของคุณ และเมื่อมี AI มันกลายเป็นสิ่งจำเป็น AI สามารถเปลี่ยนแปลงไฟล์สลิปไฟล์ในหัววินาที คุณต้องการวิธีที่สะอาดในการตรวจสอบ เก็บไว้ หรือโยนทิ้งงานนั้น

- คอมมิตบ่อย ๆ เป็นก้อนเล็ก ๆ ก่อนที่คุณจะขอการเปลี่ยนแปลงใหญ่ ให้คอมมิตสิ่งที่ใช้งานได้อยู่ก่อน working tree ที่สะอาดหมายความว่า คุณสามารถกลับไปทีนั้นได้เสมอ
- อ่าน diff ก่อนคอมมิต นี่คือการเตือนรีวิวที่แท้จริงของคุณ คุณไม่จำเป็นต้องเข้าใจทุกบรรทัด แต่คุณควรเข้าใจว่าอะไรเปลี่ยนไปและทำไม
- ใช้ branch สำหรับงานที่มีความเสี่ยง สร้าง branch ใหม่ ปลอຍให้ AI ทำงานอย่างเต็มที่ แล้วค่อย merge ก็ต่อเมื่อคุณพอใจผลลัพธ์ ถ้าไม่ ก็ลบทิ้งไป ไม่มีความเสียหายใด ๆ
- เขียนข้อความคอมมิตที่ตรงไปตรงมา ทั้งตัวคุณในอนาคตและ AI ต่างก็อ่านประวัติเพื่อเข้าใจเจตนา

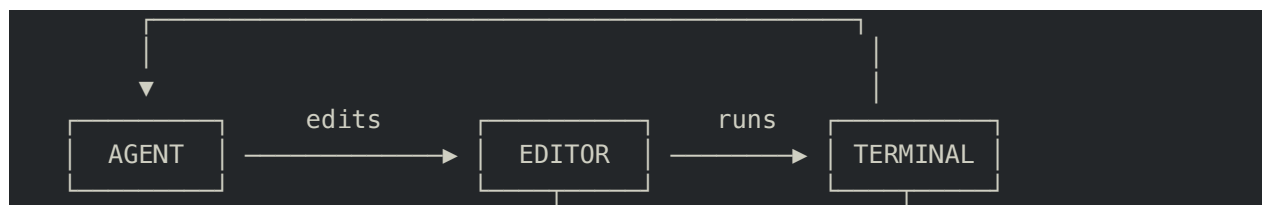
การเปลี่ยนมุมมองสำคัญคือ คอมมิตไม่ใช่งานเอกสารราชการ แต่เป็นจุดเชกในเกม เชกก่อนสับออส และเรียนรู้ทางออกฉุกเฉินสองทางที่ทำให้ประวัติที่สะอาดคุ้มค่าที่จะรักษาไว้ — git checkout . (หรือ git restore .) เพื่อโยนทิ้งการทดลองที่ยังไม่ได้คอมมิต และ git revert เพื่อยกเลิกคอมมิตที่คุณทำไปแล้ว การรู้ว่าคุณสามารถกลับไปยังสถานะที่รู้ว่าดีได้เสมอ คือสิ่งที่ทำให้คุณมอมส่ายจนที่ยาวให้ AI ได้โดยไม่วิตกกังวล

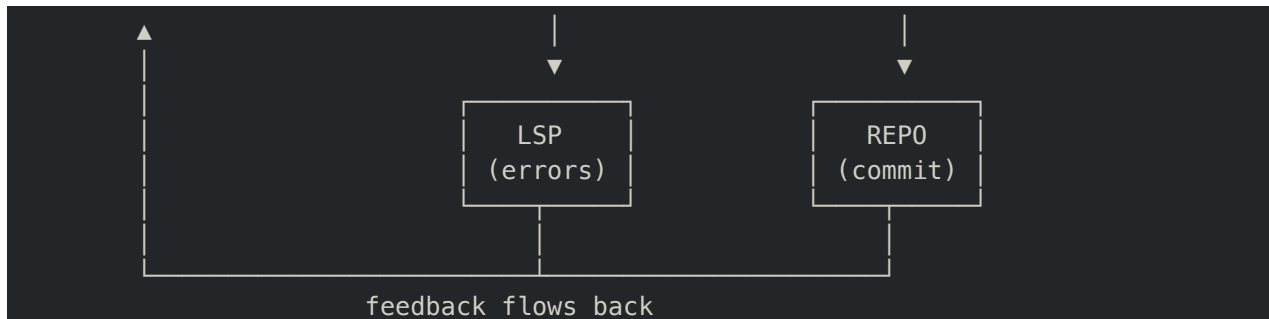
รักษาวงจรป้อนกลับให้กระชับ

ยิ่งคุณเห็นได้เร็วว่าการเปลี่ยนแปลงใช้งานได้หรือไม่ คุณและ AI ก็ยิ่งบรรจบกันสู่สิ่งที่ถูกต้องได้เร็วขึ้น วงจรที่ช้า — ที่คุณต้องรอเป็นนาทีเพื่อรู้ว่ามียะไรพังหรือเปล่า — จะทวีคูณความผิดพลาดทุกครั้งอย่างเจ็บปวด ๆ

- ทำให้ “รันมัน” เป็นคำสั่งเดียว npm run dev, make test หรืออะไรก็ตาม จงบันทึกไว้ในไฟล์บริบทของคุณเพื่อให้ AI รันมันได้เช่นกัน
- พึ่งพาเทสต์ที่รวดเร็ว แม้แต่เทสต์ยูนิตเพียงหยิบมือก็ให้ AI มีวิธีตรวจสอบงานของตัวเองโดยที่คุณไม่ต้องคอยดูแล
- ใช้ linter และ formatter พวกมันจับความผิดพลาดทั้งกลุ่มได้ในทันที และป้องกันไม่ให้ข้อโต้แย้งเรื่องสไตล์เข้ามาปนในรีวิวของคุณ
- แสดงข้อผิดพลาดกลับไปให้ AI เมื่อมียะไรพัง ให้วางข้อความ error ที่แท้จริง ปิดแบ็กที่เจาะจงจะให้การแก้ไขที่เจาะจง ส่วน “มันไม่ทำงาน” จะให้แค่การเดา

ชิ้นส่วนต่าง ๆ ของพื้นที่ทำงาน AI ประกอบกันเป็นวงจรเดียวที่เชื่อมโยงกัน เอเจนต์แก้ไขรีโพ เอดิเตอร์และ language server รายงานปัญหา เทอร์มินัลรันโค้ด และผลลัพธ์ทุกอย่างไหลกลับไปเอเจนต์





เวอร์ชันที่ลึกที่สุดของแนวคิดนี้คือการปล่อยให้ AI ปิดวงจรด้วยตัวมันเอง แทนที่คุณจะเป็นคนรันคำสั่ง อ่านความล้มเหลว และส่งต่อ ให้สิทธิ์เอเจนต์ในการรันเทสและ typechecker หลังจากทุกการเปลี่ยนแปลง แล้วให้มันตอบสนองต่อสิ่งที่เห็น วงจรจะกลายเป็น แก๊ซ รัน อ่านความล้มเหลว แก๊ซ รันอีกครั้ง ทั้งหมดนี้โดยไม่มีคนอยู่ตรงกลาง งานของคุณเปลี่ยนจากคนส่งสารเป็นผู้กำกับ คุณกำหนดเป้าหมายและรู้กัน แล้วเฝ้าดูวงจรหมุนไป เมื่อ AI สามารถแก๊ซ รัน เห็นความล้มเหลว และแก๊ซมันได้ด้วยตัวเอง คุณก็ได้สร้างวงจรที่ทำงานจริงในขณะที่คุณยังคงนั่งอยู่ในเก้าอี้ผู้กำกับ

จัดการความลับอย่างระมัดระวัง

เครื่องมือ AI อ่านไฟล์ของคุณ และบางครั้งก็แชร์บริบทกับบริการระยะไกล นั่นทำให้การรั่วไหลของความลับเป็นความเสี่ยงจริง สร้างนิสัยที่ดีตั้งแต่วันแรก

- อย่าใส่คีย์จริงในโค้ดเด็ดขาด ใช้ environment variables และไฟล์ `.env`
- ใส่ `.env` ใน gitignore เสมอ คอมมิต `.env.example` พร้อมคำว่างเปล่า เพื่อให้ AI รู้ว่าต้องใช้อะไรบ้างโดยไม่เห็นค่าจริง
- จำกัดขอบเขตของคีย์ให้แคบ และหมุนเวียนคีย์ใด ๆ ที่อาจถูกเปิดเผยไปแล้ว
- บอก AI ในไฟล์บริบทของคุณ ว่าห้าม hardcode หรือคอมมิตความลับเด็ดขาด

กฎที่ประกาศเสียงดังอย่าง Never read or print the contents of `.env` ใน AGENTS.md ของคุณคือประกันภัยราคาถูก เช่นเดียวกับ pre-commit hook หรือเครื่องมือสแกนความลับที่ปฏิเสธคอมมิตที่มีสิ่งที่ดูเหมือนคีย์ — การป้องกันหลายชั้นสำคัญมากในเรื่องนี้ เพราะต้นทุนของการรั่วไหลไม่สมมาตร คีย์ที่ถูกพืชมขึ้นไปยังรีโพสาธารณะจะถูกเจาะทันทีที่มันไปถึง แม้คุณจะลบมันในอีกไม่กี่วินาทีต่อมาก็ตาม เพราะบอตจะสแกนคอมมิตแบบเรียลไทม์ เมื่อไม่แน่ใจ ให้สมมติว่าคีย์ที่ถูกเปิดเผยนั้นถูกเผาไปแล้วและหมุนเวียนมันใหม่ การหมุนเวียนคีย์ใช้เวลาแค่ไม่กี่วินาที ในขณะที่การเก็บกวาดหลังจากคีย์รั่วไหลอาจใช้เวลาเป็นวัน

การตั้งค่าโลคัลที่มีเหตุผลทำให้การเปลี่ยนแปลงของ AI ปลอดภัย

รวมทุกอย่างเข้าด้วยกันด้วยสภาพแวดล้อมที่ความผิดพลาดมีต้นทุนต่ำและกู้คืนได้

- ทำงานบน branch ไม่ใช่บน main โดยตรง
- รักษา working tree ให้ สะอาดก่อน การเปลี่ยนแปลงใหญ่ เพื่อให้ diff มีความหมาย
- มีเทสต์และ dev server พร้อมรันได้ ด้วยคำสั่งเดียว
- เก็บความลับให้ อยู่นอกรีโพ และไฟล์บริบทให้ ทันท้ายอยู่เสมอ

ด้วยรากฐานนี้ กรณีที่แย่ที่สุดสำหรับการเปลี่ยนแปลงใด ๆ ของ AI ก็คือ git checkout . แล้วเริ่มใหม่ ความปลอดภัยนั่นเองที่ทำให้คุณเคลื่อนที่ได้เร็วและกำกับได้อย่างกล้าหาญ โดยรู้ว่าไม่มีอะไรที่ AI ทำนั้นย้อนกลับไม่ได้จริง ๆ ในบทถัดไป เราจะเริ่มใช้พื้นที่ทำงานนี้เพื่อลงมือสร้างจริง ๆ

การวินิจฉัยแบบสด (LSP)

วงจรป้อนกลับที่กระชับที่สุดไม่ต้องรอให้คุณรันโค้ด — มันจับความผิดพลาดในทันทีที่ถูกพิมพ์ นั่นคือสิ่งที่ Language Server Protocol (LSP) มอบให้ ความฉลาดแบบสดเบื้องหลังเอดิเตอร์ของคุณ รวมถึง ข้อผิดพลาด ข้อมูลชนิดตัวแปร

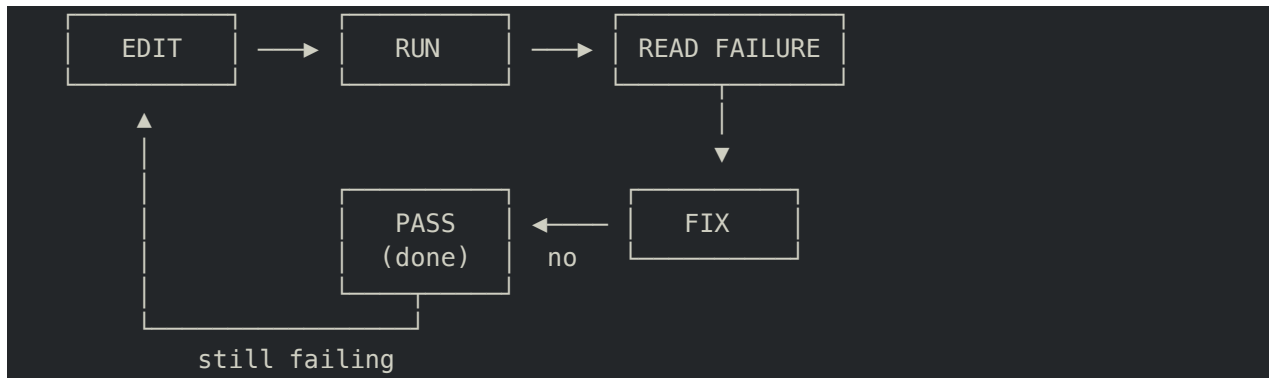
การไปยังคำนิยาม และ การเติมคำอัตโนมัติ แบบเรียลไทม์ language server อ่านโค้ดของคุณขณะที่มันเปลี่ยนแปลง และรายงานปัญหาก่อนที่จะรันจริง

ผลตอบแทนสำหรับ vibe coding คือการ ป้อนการวินิจฉัยเหล่านั้นกลับไปที่ AI หากไม่มีสิ่งนี้ วงจรจะซ้ำ AI เขียนโค้ด คุณรันมัน มันพังเพราะพิมพ์ผิดหรือ type error แล้วคุณต้องวางข้อความกลับไปด้วยมือ เมื่อเชื่อมต่อการวินิจฉัยเข้าไป แล้ว เอเจนต์จะเห็นเส้นหยักสีแดงเดียวกับที่คุณเห็นทันทีที่มันเขียนบรรทัดนั้น และแก้ไข type หรือ lint error ไปพร้อม ๆ กับที่มันทำงาน ก่อนที่โค้ดจะได้รันเลยด้วยซ้ำ

คุณสามารถให้เอเจนต์มีสายตานี้ได้หลายวิธี

- รัน language server สำหรับสแตคของคุณ (tsserver, rust-analyzer, pyright และอื่น ๆ) — เอดิเตอร์ที่ออกแบบ มาเพื่อ AI ส่วนใหญ่ทำสิ่งนี้ให้คุณอยู่แล้ว
- ให้เอเจนต์รันคำสั่ง typecheck หรือ lint แล้วอ่านผลลัพธ์ หรือเปิดเผยการวินิจฉัยผ่าน MCP server
- ปฏิบัติต่อรายงานการวินิจฉัยที่สะอาดเป็นส่วนหนึ่งของ “เสร็จแล้ว” ไม่ใช่สิ่งที่คิดขึ้นทีหลัง

เมื่อเชื่อมต่อในลักษณะนี้ เอเจนต์จะรันวงจรแก้ไขตัวเองแบบเดียวกันด้วยตัวเอง — แก้ไข รัน อ่านความล้มเหลว แก้ไข และรันอีกครั้งจนกว่าการวินิจฉัยจะกลับมาสะอาด



นี่คือเหตุผลที่ภาษาแบบ typed สามารถเป็นพลังพิเศษเจียน ๆ สำหรับ vibe coding ทุก type annotation คือข้อกำหนด เล็ก ๆ ที่เครื่องตรวจสอบได้ ซึ่งจับความผิดพลาดของ AI ก่อนที่มันจะได้รันเลยด้วยซ้ำ วงจรแบบรัน-พัง-วาง กลายเป็น วงจรที่แก้ไขตัวเองอย่างเจียน ๆ AI บดเวลาน้อยลงและบรรจบสู่คำตอบได้เร็วขึ้น

การขยายขีดความสามารถของเอเจนต์

จากกล่อง ผู้ช่วย AI ยังเป็นแบบทั่วไป โมเดลฐานเดียวกันจะกลายเป็นผู้ร่วมงานที่ดีกว่ามากเมื่อคุณห่อเครื่องมือรอบตัว มัน เปลี่ยนแซทเทรลธรรมดาคือกลายเป็น harness (สายรัดนิรภัย) ที่ปรับแต่งให้เข้ากับโปรเจกต์ของคุณ จุดขยายหลัก ๆ ที่ควรรู้มีดังนี้

- Skills — ความสามารถสำเร็จรูปที่น่ากลับมาใช้ใหม่ได้ (คำสั่งบวกลบคูณหาร) ที่เอเจนต์โหลดตามความต้องการ ทำให้ความรู้เฉพาะด้านของคุณพร้อมใช้งานโดยไม่ต้องอธิบายซ้ำทุกเซสชัน
- Plugins — ชุดรวมที่ส่งมอบ skills คำสั่ง hooks และตัวเชื่อมต่อรวมกันเป็นหน่วยเดียวที่ติดตั้งได้
- MCP (Model Context Protocol) — มาตรฐานเปิดสำหรับการเชื่อมต่อเอเจนต์กับเครื่องมือและข้อมูลภายนอก (ฐานข้อมูล, API (Application Programming Interfaces — มาตรฐานของคำสั่งที่ซอฟต์แวร์อื่นเปิดให้ใช้), ระบบไฟล์ของคุณ) ผ่านอินเทอร์เฟซร่วมเดียว
- Subagents — เอเจนต์ผู้ช่วยเฉพาะทางที่เอเจนต์หลักสร้างขึ้นสำหรับงานที่โฟกัส (การค้นหา การรีวิว การกิจเสริมที่ยาวนาน) แต่ละตัวมีหน้าตาบริบทของตัวเอง
- Hooks — สคริปต์ที่ทำงานโดยอัตโนมัติเมื่อเกิดเหตุการณ์ในวงจรชีวิต (ก่อนคำสั่งรัน หลังไฟล์ถูกแก้ไข) ทำให้คุณสามารถบังคับใช้กฎหรือฉันทาได้อย่างแน่นอน
- Slash commands — เวอร์กโฟลว์ที่มีชื่อสั้น ๆ ที่คุณเรียกใช้ด้วย /name ยับคำขอหลายขั้นตอนที่ทำให้ซ้ำ ๆ ให้เหลือเพียงการกดปุ่มเดียว

คุณไม่จำเป็นต้องมีสิ่งเหล่านี้ทั้งหมดตั้งแต่วันแรก เริ่มต้นแบบเรียบง่าย แล้วเพิ่ม skill หรือ slash command ในทันทีที่คุณสังเกตว่าตัวเองทำการตั้งค่าเดิมซ้ำ ๆ ง่ายๆ ๆ ที่ใช้ได้ผลคือ ครั้งที่สามที่คุณพิมพ์คำสั่งหลายขั้นตอนแบบเดียวกัน ให้เปลี่ยนมันเป็น slash command ครั้งที่แรกที่คุณอยากให้เอเจนต์เข้าถึงเครื่องมือที่มันเข้าไม่ถึง ให้มองหา MCP server สำหรับสิ่งนั้น โดยเฉพาะ hooks จะกลับมาอีกครั้งในบทความด้วยรั้วกันความปลอดภัย ในฐานะวิธี บังคับใช้ กฎของคุณ แทนที่จะแค่หวังว่า AI จะจำมันได้

macOS กับ Windows

เครื่องมือ AI ส่วนใหญ่ — เอเจนต์แบบ CLI, สคริปต์, บทเรียน — ต่างสันนิษฐานอย่างเจียบ ๆ ว่าใช้ Unix shell นั้น กำหนดรูปแบบการเลือกสภาพแวดล้อมพัฒนาในทางปฏิบัติ

- macOS เป็น Unix-native มันมาพร้อม POSIX shell แท้ ๆ และ git, ssh, curl ตัวเดียวกับที่โลกนักพัฒนาที่เหลื่อใช้ เอเจนต์และสคริปต์ส่วนใหญ่ “ใช้งานได้เลย”
- Windows มีความสามารถครบถ้วน แต่เส้นทางที่ราบรื่นที่สุดคือ WSL2 (Windows Subsystem for Linux) สภาพแวดล้อม Linux แท้ ๆ ภายใน Windows ให้ทำการเขียนโค้ดด้วย AI ของคุณ ภายใน WSL2 แทนที่จะต่อสู้กับความแปลกประหลาดของ PowerShell แบบเนทีฟ
- Linux คือสมาชิกลำดับที่สามของครอบครัวนี้ และมีพฤติกรรมเกือบเหมือนกับ macOS ทุกประการในจุดประสงค์เหล่านี้ — มันคือ Unix โดยตรง ดังนั้นเอเจนต์และสคริปต์เดียวกันจึงรันได้โดยไม่ต้องแก้ไข หากคุณใช้มันอยู่แล้ว คุณก็พร้อมแล้ว

มีข้อควรระวังไม่กี่อย่างเพื่อให้งานของคุณพกพาข้ามระบบเหล่านี้ได้

- เส้นทางไฟล์ — /home/you/project เทียบกับ C:\Users\you\project เบ็กสแลชและอักษรโดรฟ์ทำให้สคริปต์พัง WSL2 หลีกเลี่ยงปัญหานี้ได้ส่วนใหญ่
- การขึ้นบรรทัดใหม่ — Windows ใช้ CRLF, Unix ใช้ LF ความไม่ตรงกันทำให้ diff รกและสคริปต์เชลล์พัง ให้ตั้งค่า core.autocrlf และ .gitattributes เพื่อทำให้เป็นมาตรฐาน LF
- การแยกตัวพิมพ์เล็ก-ใหญ่ — macOS และ Windows มักถือว่า File.ts และ file.ts เป็นไฟล์เดียวกัน แต่ Linux และ CI ไม่ถือเช่นนั้น ดังนั้นโค้ดที่บิลด์ได้บนเครื่องอาจล้มเหลวบนคลาวด์ จงสม่ำเสมอตัวพิมพ์เล็ก-ใหญ่ในการ import และชื่อไฟล์
- สิทธิ์การเข้าถึง — โหมดไฟล์ของ Unix (chmod +x) ไม่สามารถแมปเข้ากับ Windows ได้อย่างสะอาด

ไม่ว่าคุณใช้ระบบไหน ให้ขอให้ AI รักษาคำสั่งให้พกพาได้ — สแลชไปข้างหน้า, การขึ้นบรรทัดแบบ LF, ตัวพิมพ์เล็ก-ใหญ่ที่สม่ำเสมอ — เพื่อให้สคริปต์ของมันรันเหมือนกันทั้งบนแล็ปท็อปของคุณและใน CI เมื่อ AI ส่งคำสั่งบรรทัดเดียวที่สันนิษฐานว่าคุณมีเครื่องมือที่คุณไม่มี วิธีแก้ที่เร็วที่สุดมักจะเป็นการบอก OS และ shell ของคุณในไฟล์บริบท เพื่อให้มันเล็กเดา

TUI เทียบกับ GUI

เครื่องมือเขียนโค้ดด้วย AI มีอยู่สองรูปแบบ และนักเขียนโค้ดแบบ vibe coding ที่มีประสิทธิภาพจะใช้ทั้งสองแบบ เอเจนต์แบบเทอร์มินัล/TUI (Claude Code, Codex CLI, Aider) รันในเทอร์มินัล รวดเร็ว เขียนสคริปต์ได้ และประกอบกันได้ เหมาะกับงานอัตโนมัติและปล่อยให้ AI ทำงานอิสระในหลายไฟล์ เครื่องมือแบบ GUI/เอดิเตอร์ (Cursor, Windsurf, ส่วนขยาย IDE) อยู่ในเอดิเตอร์แบบภาพ ค้นพบได้ง่าย มี diff แบบอินไลน์ คลิกเพื่อยอมรับ และทางลาดขึ้นที่นุ่มนวล เหมาะกว่าสำหรับการรีวิวกการเปลี่ยนแปลง

	TUI / เทอร์มินัล	GUI / เอดิเตอร์
ความเร็ว	เร็วมาก โอเวอร์เฮดต่ำ	ช้ากว่า UI มากกว่า
เขียนสคริปต์ได้	ได้ — pipe, chain, automate	จำกัด
Diff แบบภาพ	ข้อความล้วน	สมบูรณ์ อินไลน์
ค้นพบได้ง่าย	ต่ำ (ต้องรู้คำสั่ง)	สูง (คลิกได้)
เหมาะกับ	อัตโนมัติ ความอิสระ ผู้ใช้ระดับสูง	การรีวิว การปฐมนิเทศ งานเชิงภาพ

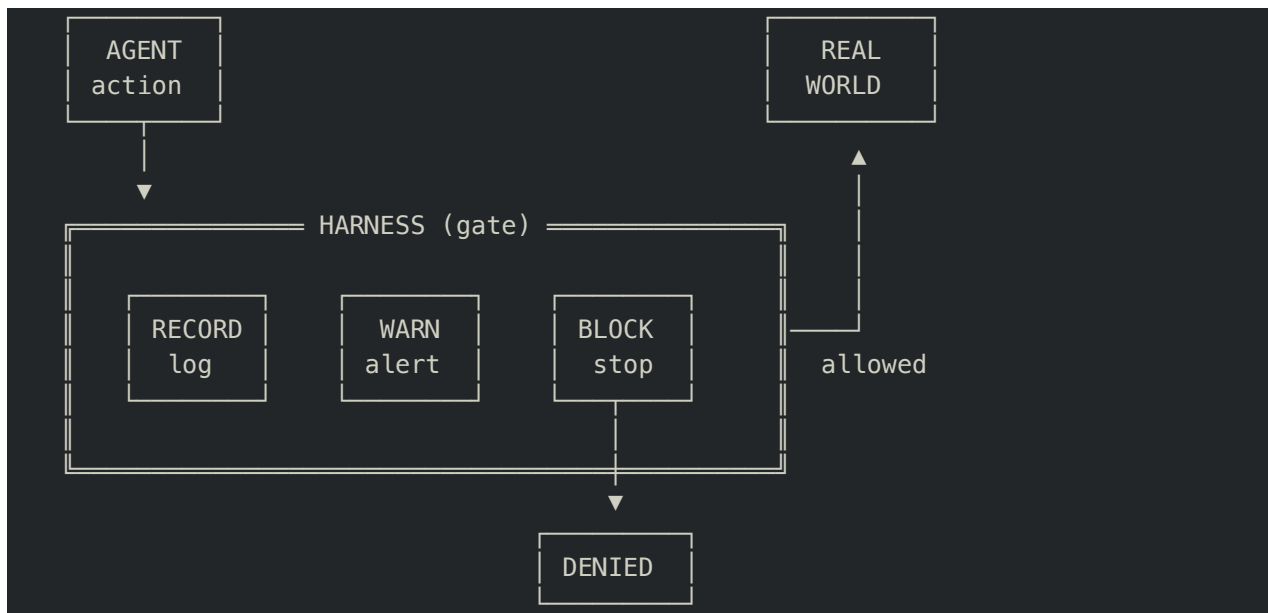
คุณไม่จำเป็นต้องเลือกอันใดอันหนึ่งตลอดไป การผสมผสานคือประเด็นสำคัญ ขับเคลื่อนงานส่วนใหญ่ผ่านเอเจนต์ TUI ที่รวดเร็ว จากนั้นเข้าไปที่ GUI สำหรับการรีวิวเชิงภาพหรือการแก้ไขแบบละเอียด เวิร์กโฟลว์ทั่วไปคือให้เอเจนต์เทอร์มินัลทำการเปลี่ยนแปลงกว้าง ๆ ในหลายไฟล์ แล้วเปิด diff แบบเคียงข้างของ GUI เพื่ออ่านและอนุมัติสิ่งที่มันทำจริง ๆ ความเร็วสำหรับการลงมือทำ ความชัดเจนสำหรับการตรวจสอบ เลือกตัวหลักที่คุณสบายใจ และจำไว้ว่าทักษะในหนังสือเล่มนี้ใช้ข้ามเครื่องมือได้

วิศวกรรมสายรัดนิรภัย (Harness)

นิสัยคือสิ่งที่คุณต้องจำเอาไว้ทำ ชันมัดไปในการตั้งค่าพื้นที่ทำงานของคุณคือการทำให้การตรวจสอบเหล่านั้นเป็นสิ่งที่ AI ข้ามไม่ได้ นั่นคือ harness

ลองนึกภาพ harness เป็นด่านตรวจความปลอดภัยที่อยู่ระหว่างเอเจนต์กับโลกจริง ในทุกขั้นตอนที่มีความเสี่ยง มันจะหาหนึ่งในสามอย่างนี้ คือบันทึกสิ่งที่เกิดขึ้น เตือนคุณ หรือบล็อกการกระทำนั้นทั้งหมด ก่อนคำสั่งจะรัน ก่อนไฟล์จะถูกเขียน เมื่อพร้อมต์ถูกส่ง ตอนคอมมิต ตอน-push — มีด่านอยู่ทุกที่ เอเจนต์ไม่สามารถเดินอ้อมมันได้ แบบที่คุณอาจลืมรันเทสต์ตอนดีสโง

ทุกการกระทำที่มีความเสี่ยงที่เอเจนต์จะทำจะผ่าน harness ก่อน ที่ซึ่งแต่ละด่านสามารถบันทึกมัน เตือนคุณ หรือบล็อกมัน ก่อนที่มันจะไปถึงโลกจริง



ทำไมต้องทำให้เป็นข้อบังคับ การพนันทั้งหมดของ vibe coding คือ “ปล่อยให้ AI รุ่งเต็มที่ คุณเป็นคนบังคับทิศทาง” มันจะได้ผลตอบแทนก็ต่อเมื่อ AI ไม่สามารถขัดขวางหน้าผาได้ในขณะที่คุณกะพริบตา harness ย้ายความปลอดภัยของคุณจากความระแวงระวัง (การอ่านทุก diff การจำทุกกฎ) ไปสู่ ไร้ขีด (กฎที่บังคับใช้โดยอัตโนมัติ) สำหรับสิ่งใดก็ตามที่คุณจะนำเสนอต่อผู้ใช้งานจริง นั่นไม่ใช่การขัดเกลาที่เลือกได้ แต่คือสิ่งที่ทำให้ความอิสระนั้นปลอดภัย

harness ที่ดีปฏิบัติตามหลักการไม่กี่ข้อที่ควรรู้

- สำเร็จเจียบ ล้มเหลวดัง — มันไม่พูดอะไรเมื่อทุกอย่างปกติ ดังนั้นคำเตือนจึงยังคงมีความหมาย
- ไม่แก้ไขอัตโนมัติเด็ดขาด — มันเสนอแนะ บล็อก หรือเตือน มันไม่ “แก้ไข” เอเจนต์อย่างเจียบ ๆ แล้วซ่อนปัญหาไว้
- ปลดระวางกฎที่ไม่มีการใช้งานก่อนเพิ่มกฎใหม่ — รักษาความกระชับและความน่าเชื่อถือ ไม่ใช่กองการตรวจสอบที่ล้าสมัยที่เพิ่มขึ้นเรื่อย ๆ
- ขับเคลื่อนด้วยคอนฟิก — เอนจินเดียว การตั้งค่าเฉพาะโปรเจกต์ เพื่อให้ใช้งานได้กับรีโพอใดก็ได้
- บันทึกแบบ append-only — บันทึกที่ปลอมแปลงได้ยากของสิ่งที่เอเจนต์ทำจริง ๆ

คุณไม่จำเป็นต้องสร้างสิ่งนี้ขึ้นเองตั้งแต่ต้น มี harness สำเร็จรูปที่ติดตั้งใช้ได้ทันที — dancinlab/harness เป็นเอน

เงินหนึ่งที่ไม่ผูกติดกับโปรเจกต์เฉพาะ ซึ่งคุณสามารถเชื่อมต่อเข้ากับวงจรชีวิตของเครื่องมือของเอเจนต์เขียนโค้ดได้ พุดตรง ๆ harness ไม่ได้ทำให้ AI ถูกต้อง แต่มันทำให้ความผิดพลาดของ AI จับได้ในราคาถูก ซึ่งเมื่อคุณกำลังเคลื่อนที่เร็ว นั่นคือส่วนใหญ่ของสมรภูมิ

ความปลอดภัยด้านต้นทุน: CLI เทียบกับ API

บิลใบเดียวสามารถทำลายโปรเจกต์ได้ และมันมักไม่ได้มาจากการโฮสติ้ง แต่มาจาก AI API ที่คิดค่าใช้จ่ายตามการใช้งาน เครื่องมือที่คุณตั้งค่าไว้เขียนโค้ดในชีวิตประจำวัน — Claude Code, Cursor และอื่น ๆ — มักเป็น การสมัครสมาชิกแบบเหมาจ่าย ต้นทุนของอีกหนึ่งพรอมต์นั้นแทบเป็นศูนย์ แต่ในทันทีที่ แอป ของคุณเรียก AI API หรือ SDK ดิบ ๆ — SDK (Software Development Kit — ชุดเครื่องมือพัฒนาซอฟต์แวร์) คือไลบรารีโค้ดสำเร็จรูปที่ผู้ให้บริการมอบให้คุณเพื่อเรียกใช้บริการของพวกเขา — เศรษฐศาสตร์จะพลิกกลับ คุณจะถูกเรียกเก็บเงิน ต่อโทเคน ต่อคำขอ โดยไม่มีเพดาน การรักรายที่มีบั๊ก งานแบบ batch ที่ไม่มีขอบเขต หรือเอเจนต์ที่วนซ้ำไม่หยุด สามารถเปลี่ยนบิลว่างเปล่าเป็นความประหลาดใจสีหลักได้อย่างเจ็บ ๆ ในชั่วข้ามคืน นั่นคือ “bill shock” (ตกใจกับบิล) และมันคือวิธีที่นักสร้างเดี่ยวส่วนใหญ่มักโดนเผา

จงสร้างกรอบ API ที่มีค่าใช้จ่ายก่อนที่จะปล่อยใช้งานจริง

- ตั้งเพดานการใช้จ่ายอย่างเข้มงวด ในแดชบอร์ดของผู้ให้บริการ — เพดานจริง ไม่ใช่แค่แจ้งเตือนทางอีเมล
- เลือกใช้ CLI แบบเหมาจ่ายสำหรับการพัฒนา เก็บ API แบบวัดปริมาณไว้สำหรับพีเอจการผลิตที่ต้องใช้จริง ๆ เท่านั้น
- จำกัดอัตราการเรียกของคุณเอง — จำกัดความพร้อมกันและจำนวนคำขอต่อนาที — และเฝ้าดูแดชบอร์ดการใช้งานในช่วงวันแรก ๆ
- ใช้โมเดลที่ราคาถูก/เล็กสำหรับงานปริมาณมาก และ ทดสอบกับข้อมูลนำเข้าเล็ก ๆ ก่อน (หนึ่งเรคคอร์ด ไม่ใช่ทั้งตาราง)
- อย่าปล่อยลุบที่ไม่มีการควบคุมที่เรียก API ที่มีค่าใช้จ่ายเด็ดขาด จำกัดขอบเขตของทุกลุบ

จงตั้งเป็นกฎตั้งแต่ต้น ทุก API ที่มีค่าใช้จ่ายต้องมีเพดานที่เข้มงวดและการจำกัดอัตรา

สรุปและฝึกฝน

ประเด็นสำคัญ

- พื้นที่ทำงานที่ดีให้บริบทแก่ AI วงจรป้อนกลับที่รวดเร็ว และการยกเลิกที่ราคาถูก — การตั้งค่าคือตัวคุณ ป่อยครั้ง ยิ่งกว่าตัวโมเดลเองเสียอีก
- ไฟล์บริบท (AGENTS.md / CLAUDE.md) คือสิ่งที่ให้ผลตอบแทนสูงสุดที่คุณเขียนได้ กฎที่เป็นรูปธรรม สิ่งที่ทำอย่างชัดเจน และตัวชี้ไปยังตัวอย่าง
- ทำให้การควบคุมเวอร์ชันเป็นปฏิกิริยาสะท้อน — คอมมิตเล็ก ๆ อ่าน diff แยก branch สำหรับงานเสี่ยง — เพื่อไม่ให้สิ่งใดที่ AI ทำนั้นย้อนกลับไม่ได้จริง ๆ
- รักษาวงจรให้กระชับและปล่อยให้เอเจนต์ปัดวงจรเอง แก๊ว รั้น อ่านความล้มเหลว แก๊ว ป้อนการวินิจฉัย (LSP, lint, typecheck) กลับไปให้ AI
- สร้างกรอบทุก API ที่มีค่าใช้จ่ายด้วยเพดานและการจำกัดอัตรา เก็บความลับให้อยู่นอกรีโพ และใช้ harness เพื่อทำให้การตรวจสอบความปลอดภัยเป็นสิ่งที่ AI ซ้ำไม่ได้

ลองทำดู

เปิดโปรเจกต์ที่คุณกำลังทำอยู่ (หรือสร้างโปรเจกต์ว่างขึ้นมาใหม่) และเขียนร่างแรกของ AGENTS.md ด้วยมือของคุณเอง ว่าโปรเจกต์นี้คืออะไร เทสแดกที่ใช้ คำสั่งที่แน่นอนสำหรับรัน dev/test/lint ธรรมเนียมจริงสามข้อ และสิ่งที่ห้ามทำอย่างชัดเจนอย่างน้อยสองข้อ (เช่น “ห้ามอ่านหรือแสดงเนื้อหาของ .env” “ห้ามเพิ่มดีเพนเดนซีโดยไม่แจ้งก่อน”) จากนั้นขอให้เครื่องมือ AI ของคุณทำการเปลี่ยนแปลงเล็กน้อย แล้วดูว่ามันเคารพไฟล์นั้นหรือไม่ ทุกครั้งที่มันทำสิ่งที่คุณไม่ต้องการ ให้เพิ่มหรือปรับบรรทัดหนึ่งให้คมชัดขึ้น คุณกำลังปรับปรุงห้อง ไม่ใช่แค่ปรับโค้ด

พรอมต์ประจำบท

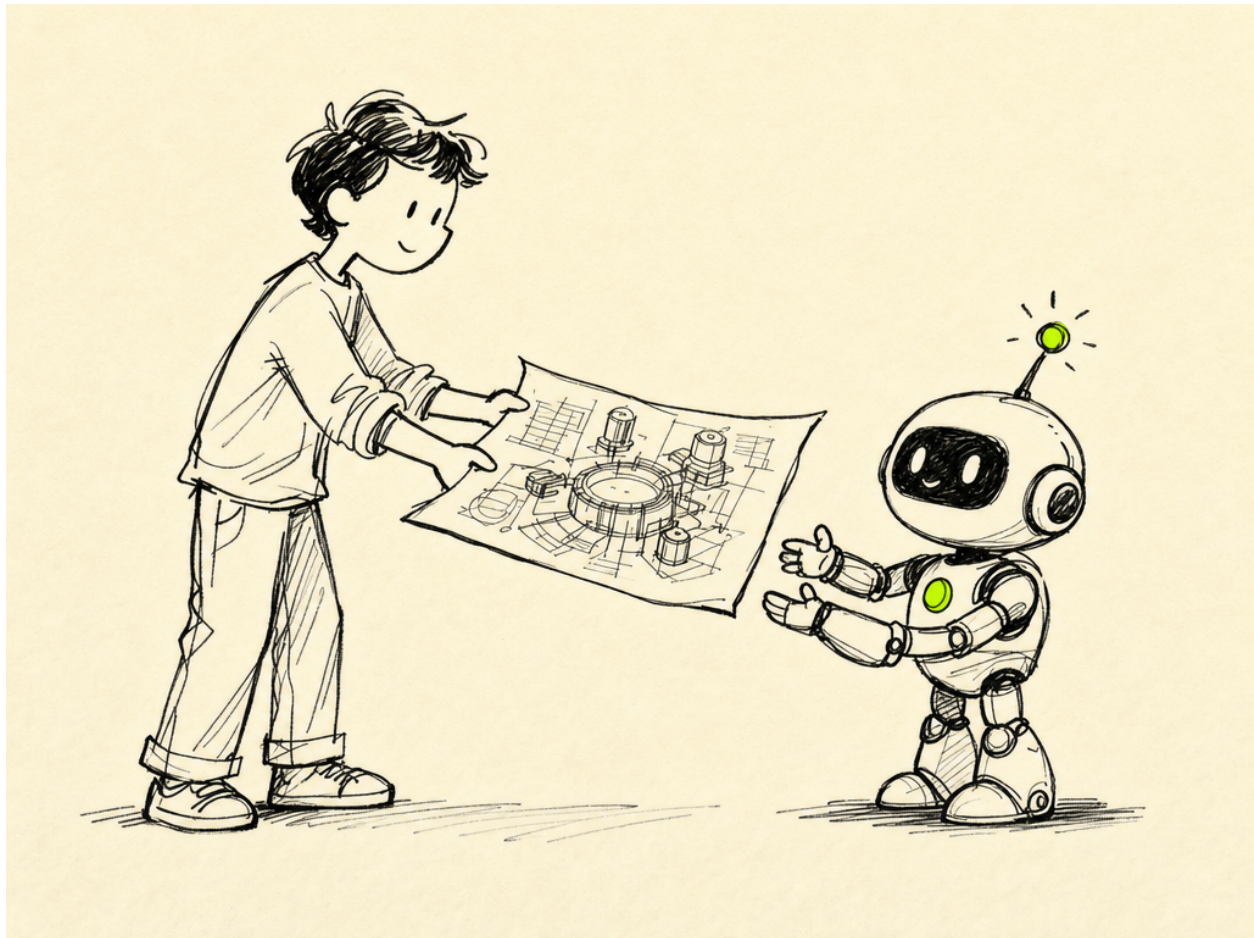
Read my project and help me write a concise AGENTS.md context file for it.

First, inspect the repo and tell me:

- the tech stack and versions you detect
- the commands for install / dev / test / lint / build
- the layout and where the entry point is

Then draft an AGENTS.md with these sections: What this is, Tech stack, Commands, Conventions, and Don'ts. Be specific and concrete (name the exact libraries and rules, not "follow our conventions"). State the don'ts as loudly as the do's, and point to one existing file as a style example. Keep it under ~40 lines — I'll prune from there.

การเขียนพรอมต์แบบวิศวกร



ความแตกต่างระหว่างเชสชัน AI ที่น่าหงุดหงิดกับเชสชันที่ได้ผลจริง แทบไม่เคยอยู่ที่ตัวโมเดลเลย แต่อยู่ที่พรอมต์ต่างหาก มีไอหม่ส่วนใหญ่ปฏิบัติกับ AI เหมือนเสิร์ชเอนจิน — พิมพ์ความต้องการคลุมเครือแล้วหวังว่าจะได้สิ่งที่ดี วิศวกรมือเก่าจะปฏิบัติกับมันเหมือนเพื่อนร่วมทีมรุ่นน้องที่ฉลาดแต่ตรงไปตรงมา — คนที่ทำงานเร็วและเก่ง แต่จะสร้างสิ่งที่คุณอธิบายไว้เป๊ะ ๆ รวมถึงความผิดพลาดของคุณด้วย

บทนี้ว่าด้วยการปิดช่องว่างนั้น คุณไม่จำเป็นต้องท่องจำเทคนิคหรือสะสมโฟลเดอร์เทมเพลตพรอมต์ “มหัศจรรย์” คุณแค่ต้องสื่อสารแบบที่วิศวกรฝีมือดีสื่อสารงานให้คนอื่นเข้าใจ: ด้วยบริบท (context) ข้อจำกัด (constraint) และค่านิยม

ชัดเจนว่า “เสร็จ” คืออะไร ทุกอย่างด้านล่างนี้เป็นแค่รูปแบบต่าง ๆ ของทักษะเดียวนั้น

ให้บริบทก่อนให้โจทย์

AI มองไม่เห็นโปรเจกต์ของคุณแบบที่คุณมองเห็น มันไม่รู้เวอร์ชันเฟรมเวิร์กที่คุณใช้ ไม่รู้โครงสร้างไฟล์ ไม่รู้ธรรมเนียม (convention) ที่คุณวางไว้แล้ว เมื่อคุณข้ามส่วนนี้ไป มันจะเดา — และเดาผิดด้วย ที่แย่กว่านั้นคือมันเดาอย่าง มั่นใจ จนผลลัพธ์ดูสมเหตุสมผลไปเรื่อย ๆ จนกว่าจะไปชนกับโค้ดจริงของคุณ

ลองเทียบพรมตีสองแบบสำหรับคำขอเดียวกันนี้

```
Add a function to validate emails.
```

```
This is a TypeScript backend using Express and Zod for validation.
We already validate inputs with Zod schemas in src/schemas/.
Add an email validation schema in that style. Emails must be
lowercase, max 254 chars, and reject disposable domains from
the existing BLOCKED_DOMAINS list in src/config.ts.
```

พรมตีสองจะได้โค้ดทั่ว ๆ ไปที่อาจใช้ regex แบบที่คุณไม่ต้องการ และเขียนในสไตล์ที่ไม่ตรงกับโค้ดเบสของคุณ ส่วนพรมตีสองจะได้โค้ดที่วางลงไปใช้ได้เลย กฎคือ บอกสแตกเทคโนโลยี ธรรมเนียมที่ใช้ และชิ้นส่วนที่มีอยู่แล้วที่ AI ควรนำมาใช้ซ้ำ

พรมตีสองจะเรียงชิ้นส่วนเหล่านี้เป็นลำดับ จากฐานขึ้นไป — บริบทอยู่ล่างสุด คำขอจริงอยู่บนสุด:

EXAMPLES	one input → output
CONSTRAINTS	what NOT to do
EDGE CASES	empty, invalid, errors
INPUTS / OUTPUTS	types and shapes
GOAL	one clear task
CONTEXT	stack, files, patterns
foundation first ▲	

รายการที่ควรเช็คก่อนกดส่ง: ภาษาและเฟรมเวิร์ก (พร้อมเวอร์ชันถ้ามีผล) ไฟล์หรือโมดูลที่เกี่ยวข้อง แพตเทิร์นที่มีอยู่แล้วในโปรเจกต์ และไลบรารีที่ AI ควรเลือกใช้หรือหลีกเลี่ยง คุณไม่จำเป็นต้องใส่ทุกอย่างทุกครั้ง — แต่ถ้าคำตอบจะเปลี่ยนไปตามข้อใดข้อหนึ่ง ให้ระบุมันออกมา เมื่อไม่แน่ใจ ให้วาง signature จริง type จริง หรือ config จริงที่โค้ดใหม่ต้องเข้ากันได้ โค้ดที่มีอยู่แล้วเพียงไม่กี่บรรทัดสอน AI เรื่องธรรมเนียมของคุณได้มากกว่าคำบรรยายทั้งย่อหน้าเสียอีก

ระบุ Input, Output และกรณีขอบ (Edge Case)

งานหนึ่งที่ยังไม่ถูกนิยามจนกว่าขอบเขตของมันจะถูกนิยาม วิศวกรมือเก๋าคิดในรูปแบบ “อะไรเข้า อะไรออก และเกิดอะไรขึ้นเมื่อสิ่งต่าง ๆ ผิดพลาด” ให้ใส่สิ่งนี้ลงในพรมตีสองด้วย

```
Write a function to parse a date string.
```

```
Write a function parseDate(input: string): Date | null.
```

```
Inputs: ISO 8601 strings ("2026-06-14", "2026-06-14T10:30:00Z").
```

```
Output: a Date object, or null if the string is invalid.
```

Edge cases to handle:

- empty string or whitespace -> null
- valid format but impossible date (2026-02-30) -> null
- trailing garbage ("2026-06-14xyz") -> null

Do not throw. Return null for all invalid input.

เมื่อคุณระบุกรณีขอบไว้ AI จะจัดการมันให้ เมื่อคุณไม่ระบุ มันจะเขียนแค่เส้นทางที่ราบรื่นที่สุด (happy path) แล้วคุณจะไปพบช่องโหว่เอาตอนใช้งานจริง (production) หลักการเดียวกันนี้ใช้ได้กว้างกว่าแค่ฟังก์ชันล้วน ๆ: สำหรับ endpoint หนึ่งตัว ให้ระบุทั้งการตอบกลับเมื่อสำเร็จ และ การตอบกลับเมื่อผิดพลาด (จะให้ status code อะไรเมื่ออีเมลซ้ำ? เมื่อ body ผิดรูปแบบ?) สำหรับคอมโพเนนต์ UI ให้ระบุสถานะว่างเปล่าและสถานะกำลังโหลดด้วย ไม่ใช่แค่ตอนมีข้อมูลเต็ม พฤติกรรมที่น่าสนใจที่สุดของซอฟต์แวร์ส่วนใหญ่อยู่ที่ขอบของมัน ดังนั้นนั่นคือจุดที่พรมตของคุณควรใช้คำมากที่สุด

ถ้าคุณไม่แน่ใจว่ากรณีขอบมีอะไรบ้าง ให้ถาม AI ก่อน: “ตัว parser วันที่ควรจัดการกรณีขอบอะไรบ้าง?” — แล้วนำคำตอบนั้นมาป้อนกลับเป็นข้อจำกัด วิธีนี้เปลี่ยน AI จากตัวสร้างโค้ดให้กลายเป็นตัวสร้างเช็กลิสต์ และการตรวจสอบเช็กลิสต์นั้นเร็วกว่าการตีบั๊กกรณีที่คุณลืมนึกถึง

ทำงานเป็นขั้นตอนเล็ก ๆ

ความผิดพลาดที่พบบ่อยที่สุดคือการขอมากเกินไปในครั้งเดียว “สร้างแดชบอร์ดผู้ใช้ที่มีทั้งระบบล็อกอิน กราฟ และหน้าตั้งค่าให้หน่อย” จะได้โค้ดกองมืมหาที่คุณตรวจสอบไม่ไหว พร้อมบักกระจายอยู่ในไฟล์ที่คุณไม่ได้อ่าน

ขั้นตอนเล็ก ๆ ช่วยให้คุณควบคุมสถานการณ์ได้:

- ส่งมอบทีละหนึ่งฟังก์ชัน หนึ่งคอมโพเนนต์ หรือหนึ่ง endpoint ต่อหนึ่งพรมต
- ตรวจสอบว่าแต่ละขั้นทำงานได้ก่อนไปต่อ
- ต่อยอดจากโค้ดที่ยืนยันแล้วว่าใช้งานได้ ไม่ใช่กองโค้ดที่ยังไม่ได้ตรวจสอบ

Build the whole authentication system with login, signup, password reset, sessions, and email verification.

Step 1 of the auth flow: write just the signup endpoint.
POST /signup taking { email, password }. Hash the password with bcrypt, store the user, return 201 with the user id. Assume the User model and db client already exist (I'll show you their shapes). We'll handle login next.

พรมตที่เล็กลงหมายถึง diff ที่เล็กลง และ diff ที่เล็กลงหมายถึงคุณอ่านสิ่งที่เปลี่ยนแปลงจริง ๆ ได้ นั่นคือหัวใจทั้งหมดของเกมนี้ ยังมีผลตอบแทนอีกอย่างหนึ่งด้วย: เมื่อมีอะไรพัง ขั้นตอนเล็ก ๆ ทำให้หาสาเหตุ (bisect) ได้ง่ายมาก ถ้าคุณสร้างและตรวจสอบ endpoint สมัครง่ายแล้ว ตามด้วยตรรกะเซสชัน แล้วตามด้วยการรีเซตรหัสผ่าน บั๊กที่ปรากฏหลังขั้นตอนที่สามแทบจะแน่นอนว่าอยู่ในขั้นตอนที่สาม การสร้างโค้ดทีเดียวใหญ่ ๆ ทำลายสัญญาณนี้ทิ้งไป — บั๊กอาจอยู่ตรงไหนก็ได้ในโค้ดสัร้อยบรรทัดที่คุณไม่เคยอ่าน

ขอแผนก่อนขอโค้ด

สำหรับงานที่ไม่ธรรมดา ให้แยก การคิด ออกจาก การพิมพ์ ขอให้ AI ร่างแนวทางก่อน วิธีนี้ช่วยจับสมมติฐานที่ผิดได้ ก่อนที่มันจะกลายเป็นโค้ดผืดยาว 200 บรรทัด และการแก้แผนย่อมถูกกว่าการแก้ pull request มาก

I want to add rate limiting to my API. Before writing any code, give me a short plan: where the limiter should live, what storage it needs, what happens when a limit is hit, and any tradeoffs. Don't write code yet — I'll approve the approach first.

เมื่อแผนถูกต้องแล้ว บอกว่า “ดี ทำขั้นตอนที่ 1 เลย” เมื่อแผนไม่ถูกต้อง คุณใช้เวลาแค่สามสิบวินาทีในการแก้ทิศทาง แทนที่จะเสียเวลาเป็นชั่วโมงในการคลี่คลายโค้ดที่ยุ่งเหยิง การวางแผนก่อนคือนิสัยที่แยกการเขียนพารามิเตอร์แบบวิศวกร ออกจากการเขียนพารามิเตอร์แบบขอไปที มากที่สุด

ขั้นตอนวางแผนยังเป็นจุดที่คุณจะจับได้ว่า AI เลือกทางแก้ที่หนักเกินความจำเป็น — หยิบ Redis มาใช้ทั้งที่แค่ in-memory map ก็เพียงพอ หรือเขียนสิ่งที่โลบราปริรบรรทัดเดียวแก้ได้อยู่แล้วขึ้นมาเอง อ่านแผนโดยตั้งคำถามเดียวไว้ในใจ: นี่คือนิสัยที่ง่ายที่สุดที่ยังใช้งานได้หรือเปล่า? ถ้ากลืนของแนวทางนั้นดูซับซ้อนเกินจำเป็น ให้หักท้วงตรง ๆ (“นี่เป็นโปรเจกต์งานอดิเรกที่มีเซิร์ฟเวอร์เดียว ข้าม distributed store ได้ไหม?”) ก่อนที่จะมีการเขียนโค้ดสักบรรทัดเดียว

แก้ทีละส่วน (Diff) ไม่ใช่เขียนใหม่ทั้งหมด

เมื่อมีโค้ดอยู่แล้ว จงอดใจไม่ขอให้เขียนทั้งหมดใหม่อีกครั้ง คำขอตามหลังที่คลุมเครืออย่าง “ทำให้ดีขึ้นหน่อย” จะทิ้งโค้ดที่ใช้งานได้แล้วไปและนำบั๊กที่คุณแก้ไปแล้วกลับมาใหม่ ให้ชี้เฉพาะจุดที่คุณต้องการเปลี่ยนแปลง

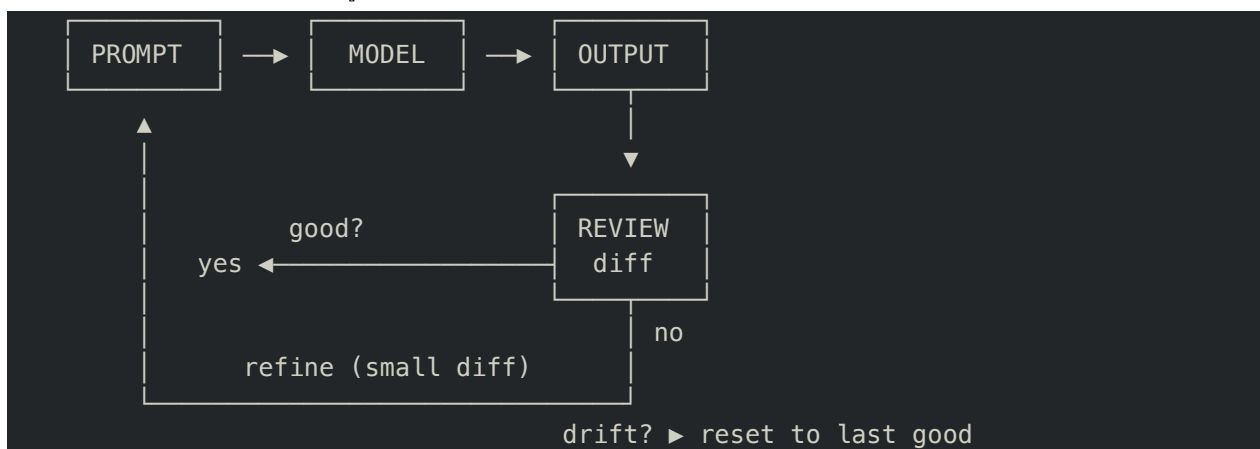
```
That's not right, rewrite the whole thing.
```

```
The function works, but two issues:
```

1. The error on line with `throw new Error` should return a Result type instead of throwing – match the pattern in the `validateUser` function above.
2. The loop re-reads `items.length` each iteration; hoist it.

```
Show me just the diff for those two changes.
```

วงจรที่ดีต่อสุขภาพคือลูปแคบ ๆ นี้: ป้อนพารามิเตอร์ให้โมเดล อ่านผลลัพธ์ แล้วปรับแก้ด้วย diff เล็ก ๆ ที่ตรงจุด — ไม่ใช่เขียนใหม่ทั้งหมด — จนกว่าจะถูกต้อง:



การขอ diff (แทนการเขียนใหม่ทั้งหมด) ทำให้การเปลี่ยนแปลงยังตรวจสอบได้ และรักษาสวนที่ทำงานได้อยู่แล้วเอาไว้ ปฏิบัติกับผลลัพธ์ของ AI เหมือนกับ pull request ของเพื่อนร่วมทีม: คอมเมนต์ในบรรทัดที่เจาะจง ขอให้แก้ไขเฉพาะจุด เมื่อการเปลี่ยนแปลงเริ่มเบี่ยงเบนไปเรื่อย ๆ และโค้ดห่างจากสิ่งที่คุณต้องการมากขึ้นทุกครั้งที่ตอบกลับ อย่าพยายามปะแก้ต่อไป — ให้ย้อนกลับไปยังเวอร์ชันสุดท้ายที่คุณเชื่อถือได้แล้วเริ่มพารามิเตอร์ใหม่จากตรงนั้นด้วยคำอธิบายที่คมชัดกว่าเดิม การเดินทางต่อบนฐานที่พังแล้วมีแต่จะทำให้ความยุ่งเหยิงทับถมมากขึ้น

ให้ตัวอย่าง

ตัวอย่างที่เป็นรูปธรรมหนึ่งตัวอย่างมีค่ามากกว่าคำบรรยายทั้งย่อหน้า ถ้าคุณต้องการผลลัพธ์ในรูปแบบใดรูปแบบหนึ่ง จงแสดงรูปแบบนั้นให้เห็น นี่คือนิสัยที่เร็วที่สุดในการขจัดความคลุมเครือ

```
Format the data nicely.
```

```
Transform this array of users into a summary object.
```

```
Input:
```

```
[{ name: "Ann", role: "admin" }, { name: "Bo", role: "user" }]
```

```
Desired output:
```

```
{ total: 2, byRole: { admin: ["Ann"], user: ["Bo"] } }
```

```
Write the transform function and a test matching this example.
```

ตัวอย่างช่วยตรึงเรื่องการตั้งชื่อ โครงสร้าง และตัวพิมพ์เล็กใหญ่ที่คำพูดเพียงอย่างเดียวยังคลุมเครืออยู่ เมื่อเป็นไปได้ ให้ใส่ตัวอย่างในรูปแบบเทสต์ (test) — แล้ว “ทำให้เทสต์ผ่าน” จะกลายเป็นคำสั่งที่ไม่คลุมเครืออีกต่อไป เทคนิคนี้ใช้ได้ ในวงกว้าง: ตัวอย่างการตอบกลับจาก API (Application Programming Interface — วิธีที่โปรแกรมหนึ่งขอข้อมูลจาก อีกโปรแกรมหนึ่ง) ช่วยตรึง parser คู่ตัวอย่างก่อน/หลังช่วยตรึงการรีแฟกเตอร์ ภาพหน้าจอหรือภาพร่าง ASCII ช่วย ตรึงเลย์เอาต์ เมื่อไรก็ตามที่คุณจับได้ว่ากำลังเขียนคำบรรยายรูปแบบยาว ๆ ให้หยุดแล้วแสดงตัวอย่างจริงหนึ่งตัวอย่าง แทน

บอก AI ว่าอะไรที่ไม่ควรทำ

ข้อจำกัดไม่ได้มีไว้แค่เรื่องผลลัพธ์ที่คุณต้องการเท่านั้น — แต่ยังเป็นเรื่องการเคลื่อนไหวที่คุณต้องการให้ AI หลีกเลี่ยง ด้วย หากปล่อยไว้โดยไม่มีข้อจำกัด โมเดลมักเอนเอียงไปทางการเพิ่ม dependency การรีแฟกเตอร์โค้ด “อย่างช่วย เหลือ” ที่คุณไม่ได้ขอให้แต่ละ และการสร้างพีเจอรืเพิ่มเติมรอบ ๆ ขอบคำขอของคุณ ข้อจำกัดเชิงลบไม่ก็ข้อช่วยให้ ผลลัพธ์กระชับ

```
Add a debounce to the search input.
```

```
Add a debounce to the search input.
```

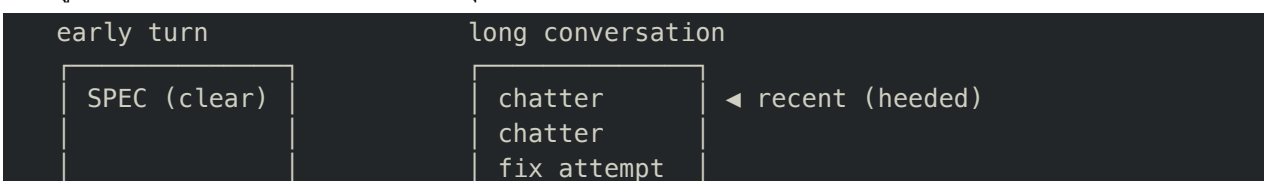
```
Constraints:
```

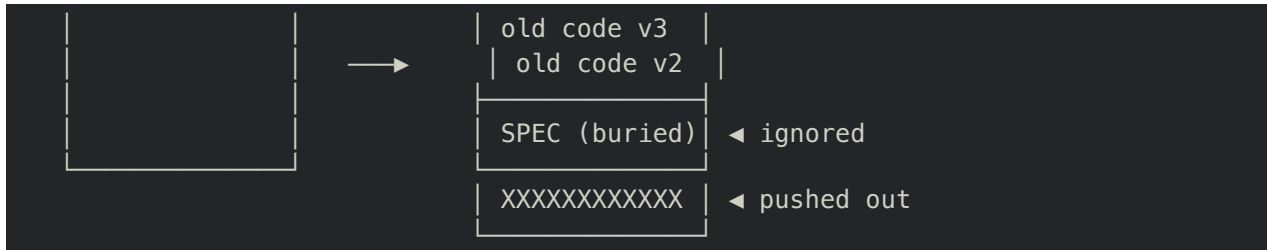
- No new dependencies — we already have lodash, use that.
- Touch only SearchBar.tsx. Do not refactor the parent.
- Keep the existing prop names and types unchanged.
- 300ms delay.

พรอมต์ที่สองปิดกั้นวิธีที่พรอมต์แรกจะพลาดได้นับสิบทาง ข้อจำกัดเชิงลบมีค่ามากเป็นพิเศษในโค้ดเบสที่มีอยู่แล้ว เพราะ ต้นทุนของการเปลี่ยนแปลงที่ไม่ได้ร้องขอไม่ได้มีแค่การตรวจทาน — แต่ยังมี ความขัดแย้งในการรวมโค้ด (merge conflict) เพื่อนร่วมทีมที่ประหลาดใจ และการถดถอย (regression) ในโค้ดที่เคยทำงานได้ดีอยู่แล้ว ถ้ามีเส้นที่คุณไม่ ต้องการให้ล้ำเข้าไป ให้พูดออกมาตรง ๆ AI จะไม่เดาขอบเขตของคุณจากความสุภาพ

จัดการบริบทเหมือนทรัพยากรอย่างหนึ่ง

บทสนทนาที่ยาวไม่ได้ไม่มีต้นทุน เมื่อเธรดยาวขึ้นเรื่อย ๆ คำสั่งก่อนหน้าจะถูกฝังลึกลงไป โมเดลจะพึ่งพาข้อความล่าสุด มากขึ้น และรายละเอียดที่คุณพูดไว้เมื่อยี่สิบรอบก่อนก็จะค่อย ๆ หมดความสำคัญไปอย่างเจิบ ๆ ลองนึกภาพหน้าต่าง บริบท (context window) เป็นกล่องขนาดคงที่: เมื่อบทสนทนาโตขึ้น กล่องก็เต็มขึ้น และคำสั่งแรกสุดของคุณก็จะถูกดัน ไปอยู่ก้นกล่องที่ไม่เดลิให้ความสนใจน้อยที่สุด:





ปฏิบัติกับหน้าตาเว็บที่เหมือนพื้นที่ทำงานที่คุณมีหน้าที่ดูแลให้สะอาด ไม่ใช่หน่วยความจำที่ไม่มีวันหมด

นิสัยไม่ก็อย่างที่ช่วยได้:

- ทวนสเปกใหม่เมื่อคุณเปลี่ยนทิศทาง อย่าสมมติว่าข้อกำหนดจากสืบทอดความก่อนหน้ายังใช้ได้ อยู่สรุปเป้าหมายปัจจุบันเป็นย่อหน้าสั้น ๆ แล้วให้ AI ทำงานจากตรงนั้น
- เริ่มบทสนทนาใหม่สำหรับปัญหาใหม่ เมื่อคุณเปลี่ยนจากโค้ดระบบล็อกอินไปเป็นบั๊ก CSS (Cascading Style Sheets — ภาษาที่ควบคุมหน้าตาของเว็บไซต์) ให้เปิดบทสนทนาใหม่ การลากบริบทที่ไม่เกี่ยวข้องติดไปด้วยจะทำให้โมเดลแย่ง ไม่ใช่อะไร
- วางโค้ดปัจจุบัน ไม่ใช่ประวัติศาสตร์ ถ้าฟังก์ชันหนึ่งเปลี่ยนไปแล้วห้าครั้ง AI ไม่ต้องการเวอร์ชันเก่าทั้งห้า — มันต้องการเวอร์ชันที่มีอยู่ตอนนี้ ให้สถานะปัจจุบันและการเปลี่ยนแปลงถัดไปเท่านั้น
- เก็บแหล่งความจริงสั้น ๆ ไว้นอกแชท สำหรับงานที่ยาวขึ้น ไฟล์เพียงไม่กี่บรรทัด (“สแต็ก การตัดสินใจสำคัญอะไรเสร็จแล้ว อะไรต้องทำต่อ”) ช่วยให้คุณเริ่มบทสนทนาใหม่ได้ในไม่กี่วินาที แทนที่จะต้องไล่สร้างมันขึ้นมาใหม่จากการเลื่อนย้อนกลับ

เป้าหมายคือสิ่งเดียวกับที่วิ่งอยู่ตลอดทั้งบทนี้: ทุกขณะ AI ควรทำงานจากภาพที่ถูกต้องและกระชับของงาน — ไม่ใช่บทสนทนายาวเหยียดที่ต้องเดาทางเอาเอง

รูปแบบที่ควรหลีกเลี่ยง (Anti-Pattern ที่พบบ่อย)

นี่คือนิสัยการเขียนพรอมต์ที่ค่อย ๆ ผลาญเวลาของคุณไปอย่างเงียบ ๆ คอยสังเกตพวกมันในแชตของคุณเอง

- ความปรารถนาคลุมเครือ “สร้างหน้าล็อกอินที่ดีให้หน่อย” ดีตามมาตรฐานของใคร? แทนที่ค่าคุณศัพท์ด้วยความเจาะจง: ฟิลต์ไหนบ้าง การตรวจสอบแบบไหน เฟรมเวิร์กอะไร
- พรอมต์ยักรั้ว โยนทั้งพีเจอรลงในข้อความเดียว แบ่งเป็นขั้นตอนที่คุณตรวจสอบได้ที่ละขั้น
- บริบทที่เจียวยาว สมมติว่า AI จำได้หรือรู้จักโปรเจกต์ของคุณ ทวนไฟล์ ประเภทข้อมูล และธรรมเนียมที่เกี่ยวข้องอีกครั้ง โดยเฉพาะในบทสนทนาใหม่
- การยอมรับแบบไม่มองไม่ตรวจ วางโค้ดที่สร้างขึ้นโดยไม่อ่านมัน AI มั่นใจแม้ในตอนที่คุณยังสงสัย คุณยังคงเป็นวิศวกรผู้รับผิดชอบ
- การลองซ้ำไม่รู้จบ ส่ง “ลองอีกครั้ง” ซ้ำ ๆ หวังผลลัพธ์ที่ต่างออกไป ถ้าลองสองครั้งแล้วยังพลาด ปัญหาอยู่ที่พรอมต์ ให้เพิ่มบริบทหรือข้อจำกัด อย่าแค่กดปุ่มใหม่
- เป้าหมายที่เคลื่อนที่ เปลี่ยนความต้องการกลางเรดโดยไม่ทวนใหม่ สรุปสเปกปัจจุบันเมื่อคุณเปลี่ยนทิศทาง เพื่อให้ AI ทำงานจากภาพเก่าที่ล้าสมัย
- วงจรคำยืนยัน ยอมรับ “คุณพูดถูกต้องเลย!” ว่าเป็นหลักฐานว่าการแก้ไขนั้นถูกต้อง การเห็นด้วยไม่ใช่การตรวจสอบ รันโค้ดและตรวจสอบพฤติกรรมด้วยตัวคุณเอง

แก่นความคิดหลัก

เทคนิคทุกอย่างในที่นี้ย่อลงมาเหลือความคิดเดียว: พุดในสิ่งที่你必须พูดกับเพื่อนร่วมทีมที่มีความสามารถแต่อ่านใจคุณไม่ได้ บริบทมาก่อน อินพุตและเอาต์พุตที่ชัดเจน ขั้นตอนเล็ก ๆ ที่ตรวจสอบได้ แผนก่อนโค้ด การแก้ไขเฉพาะจุด แทนการเขียนใหม่ ตัวอย่างในจุดที่คำพูดไปไม่ถึง ข้อจำกัดชัดเจนว่าอะไรไม่ควรแตะ ภาพที่สะอาดและทันสมัยของงานอยู่ตลอดเวลา

คุณไม่ได้กำลังเขียนคาถาวิเศษ คุณกำลังเขียนสเปก สั้นและแม่นยำ แล้วจึงตรวจสอบผลลัพธ์ ทำแบบนี้สม่ำเสมอ

แล้ว AI จะเลิกให้ความรู้สึกเหมือนเครื่องสล็อต และเริ่มให้ความรู้สึกเหมือนวิศวกรที่รวดเร็วและไม่รู้จักเหน็ดเหนื่อยตาม ที่มันเป็นได้จริง ๆ — และที่สำคัญไม่แพ้กัน คุณยังคงเป็นคนที่เข้าใจว่าอะไรถูกส่งมอบออกไป

สรุปและแบบฝึกหัด

ประเด็นสำคัญ

- ให้อ่านบริบทก่อนให้โจทย์: ระบุสแต็กเทคโนโลยี ธรรมเนียม และชิ้นส่วนที่มีอยู่แล้วที่ควรนำมาใช้ซ้ำ — หรือวาง signature จริงที่โค้ดต้องเข้ากันได้
- นิยามขอบเขต: อินพุต เอาต์พุต และกรณีขอบ พหุติกรรมที่น่าสนใจของซอฟต์แวร์อยู่ที่ขอบของมัน ดังนั้นจงใช้ คำของคุณตรงนั้น
- ทำงานเป็นขั้นตอนเล็ก ๆ ที่ตรวจสอบได้ และขอแผนก่อนขอโค้ด เพราะการแก้แผนถูกกว่าการแก้ pull request มาก
- แก่ทีละส่วน (diff) ที่ตรงจุด ไม่ใช่เขียนใหม่ทั้งหมด และใช้ตัวอย่างที่เป็นรูปธรรม (ในรูปแบบเทสต์ยิงดี) เพื่อตรง สิ่งที่คำพูดยังคลุมเครือ
- จัดการหน้าตาบริบทเหมือนทรัพยากรอย่างหนึ่ง และบอกว่าอะไรไม่ควรทำ — ข้อจำกัดเชิงลบที่ชัดเจนช่วยให้ ผลลัพธ์กระชับ

ลองทำดู

หยิบคำขอสั้น ๆ คลุมเครือที่คุณอาจพิมพ์ตามปกติ (“เพิ่มการค้นหาให้ลิสต์หน่อย”) แล้วเขียนมันใหม่เป็นพรมต์แบบ วิศวกรก่อนส่งอะไรออกไป เพิ่มบริบท (สแต็ก ไฟล์ที่เกี่ยวข้อง) อินพุต/เอาต์พุต กรณีขอบสองสามข้อ ตัวอย่างผลลัพธ์ที่ ต้องการหนึ่งตัวอย่าง และข้อจำกัดเชิงลบหนึ่งข้อ (อะไรที่ไม่ควรแตะ) ส่งเวอร์ชันที่เขียนใหม่แล้วเปรียบเทียบกับสิ่งที่ เวอร์ชันชี้แจงจะให้ผลลัพธ์ออกมา ทำแบบนี้กับคำขอจริงสามครั้ง แล้วการอัปเดตนี้จะกลายเป็นเรื่องอัตโนมัติ

พรมต์ปราคาบท

```
Context: <language + framework + version>, working in <file/module>.  
We already <relevant existing pattern or library to reuse>.
```

```
Task: <one clear thing to build>.
```

```
Inputs / outputs: <types and shapes>.
```

```
Edge cases to handle: <list the ones that matter>.
```

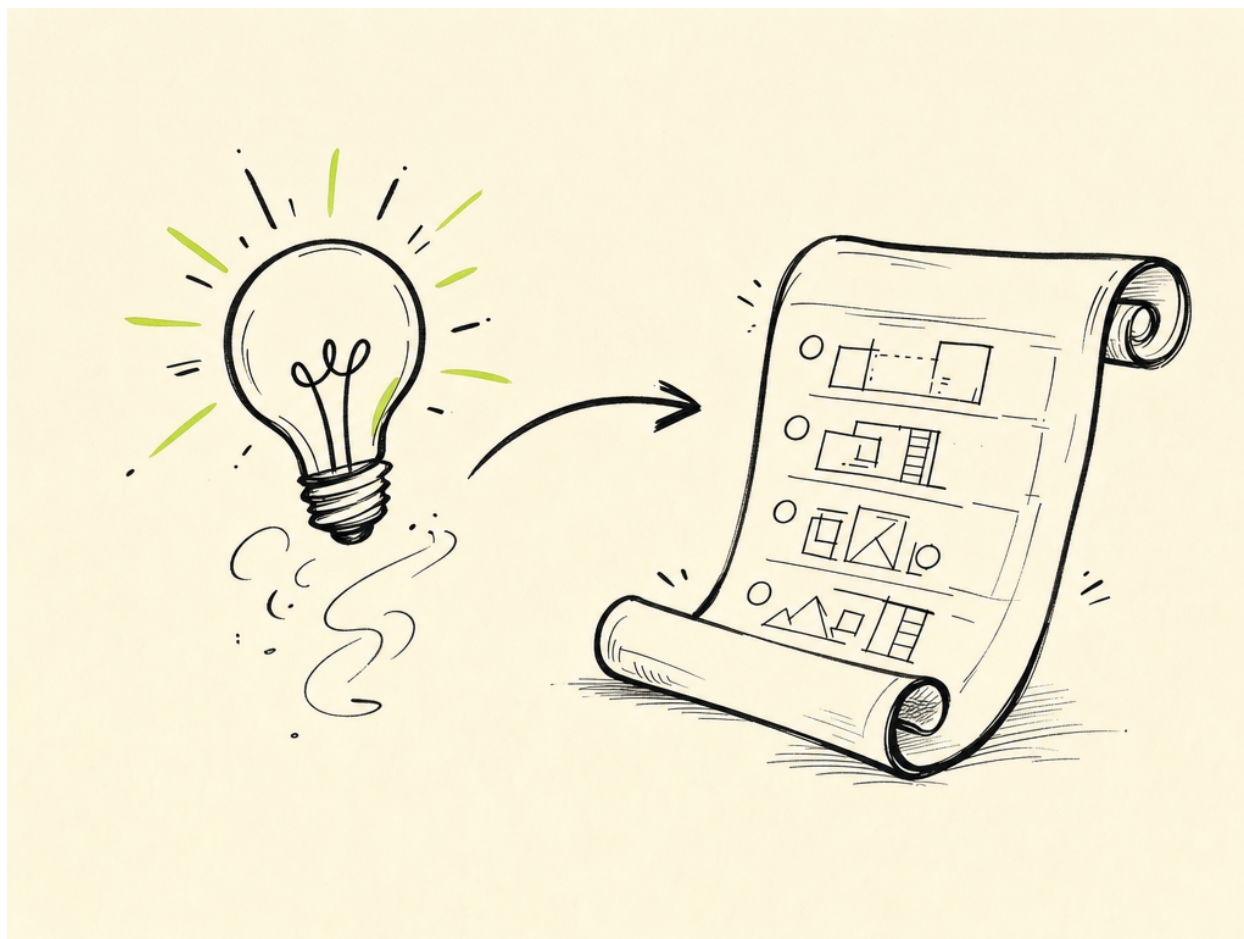
```
Example: <one concrete input -> expected output>.
```

Constraints:

- No new dependencies – use <what we already have>.
- Touch only <this file>; don't refactor anything else.
- Keep existing names and signatures unchanged.

```
Before coding, give me a 3-4 line plan and wait for my "go".
```

จากไอเดียสู่สเปกในไม่กี่นาที



ทุกโปรเจกต์เริ่มต้นจากความคิดที่ยังคลุมเครือ: “ฉันอยากได้แอปที่ช่วยติดตามหนังสือที่กำลังอ่าน” นั่นเป็นเมล็ดพันธุ์ที่ดี แต่คุณสร้างงานจากมันตรง ๆ ไม่ได้ — และ AI ของคุณก็ทำไม่ได้เหมือนกัน ไอเดียที่คลุมเครือเมื่อส่งให้ AI จะได้ซอฟต์แวร์ที่คลุมเครือกลับมา: ฟังก์ชันที่คุณไม่ได้ขอ ส่วนที่คุณต้องการกลับหายไป และโค้ดกองใหญ่ที่คุณไม่เข้าใจ

ทางแก้คือ สเปก (spec) — เอกสารสั้น ๆ เขียนด้วยภาษาธรรมดา ที่บอกว่าคุณกำลังสร้างอะไร และ ไม่ กำลังสร้างอะไร ขาวดีคือ เมื่อมี AI เป็นคู่คิด คุณสามารถเดินทางจากไอเดียบรรทัดเดียวไปสู่สเปกที่พร้อมสร้างได้จริงภายในไม่กี่นาที ไม่ใช่หลายวัน บทนี้จะแสดงวิธีการให้ดู

เพื่อให้ชัดเจนว่าสเปกคืออะไรไม่ใช่: มันไม่ใช่สัญญา ไม่ใช่เอกสารออกแบบที่คุณต้องปกป้องในที่ประชุม และไม่ใช่สิ่งที่เขียนครั้งเดียวแล้วตึงไว้ตายตัว มันคือบันทึกการทำงานสำหรับตัวคุณเองและ AI ของคุณ คุณจะแก้ไขมันไปเรื่อย ๆ ตามที่เรียนรู้ เป้าหมายไม่ใช่เอกสารที่สมบูรณ์แบบ — แต่คือเอกสารที่ ดีพอ จะทำให้ชั่วโมงถัดไปของการสร้างงานชัดเจนกว่าชั่วโมงที่แล้ว

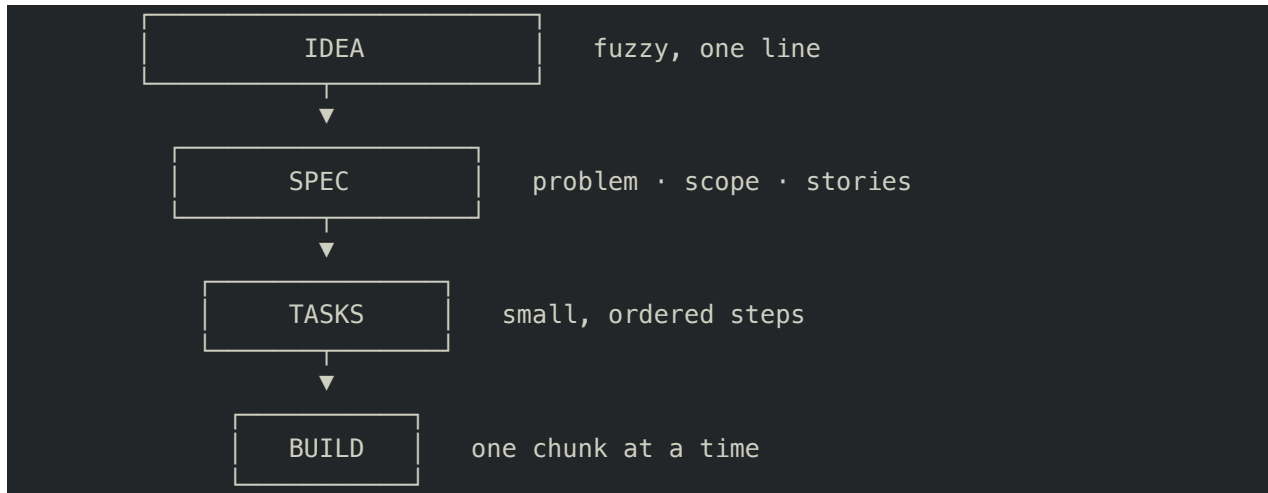
ทำไมต้องมีสเปก ไม่ใช่แค่ใช้สัญชาตญาณ

Vibe coding ไม่ได้แปลว่าข้ามการวางแผน มันหมายถึงการวางแผนแบบ เร็ว และ เบา แล้วสั่งให้ AI ลงมือทำ สเปกคือพวงมาลัยบังคับทิศทาง ถ้าไม่มีมัน AI จะเดาเจตนาของคุณในทุกพรอมต์ และความเข้าใจผิดเล็ก ๆ น้อย ๆ จะสะสมกลายเป็นความยุ่งเหยิง

สเปกให้สิ่งสำคัญสามอย่างแก่คุณ:

- บริบทรวม — คุณสามารถวางมันลงในเซสชัน AI ไหนก็ได้ และสร้างความเข้าใจใหม่ได้ทันทีที่กำลังสร้างอะไรอยู่
- ขอบเขตที่ชัดเจน — เส้นแบ่งที่เขียนไว้ระหว่าง “อยู่ในขอบเขต” กับ “อยู่นอกขอบเขต” ที่หยุดยั้งการขยายฟีเจอรแบบไม่รู้จบ
- รายการงาน — ชิ้นงานย่อยที่สร้างได้ทีละขั้น ให้ AI ทำเสร็จได้ทีละอย่าง

สเปกทำงานเหมือนกรวย: มันรับไอเดียที่กว้างและคลุมเครือ แล้วค่อย ๆ บีบให้แคบลงทีละขั้น จนกลายเป็นชิ้นงานเล็ก ๆ ที่ AI สร้างได้จริง



มันไม่จำเป็นต้องยาว หนึ่งหน้าก็เพียงพอสำหรับโปรเจกต์ส่วนใหญ่ แท้จริงแล้ว สเปกที่ยาวเกินสองหน้ามักเป็นสัญญาณเตือน: คุณกำลังวางแผนล่วงหน้าไกลเกินไป หรือกำลังขยายขอบเขตงานอย่างเจียน ๆ เมื่อถึงเวลาให้ตัดออก ส่วนที่คุณตัดทิ้งมักจะไม่มีใครคิดถึง และส่วนที่คุณเก็บไว้จะยิ่งคมชัดขึ้น

ขั้นตอนที่ 1: ทำความเข้าใจปัญหาให้ชัด

ก่อนพูดถึงฟีเจอร ให้ระบุปัญหาและคนที่ประสบปัญหานั้นก่อน อย่ารีบกระโดดไปหาทางแก้ คำอธิบายปัญหาที่ชัดเจนจะทำให้การตัดสินใจทุกอย่างต่อจากนี้ตรงประเด็น

ถามตัวเอง (หรือให้ AI สัมภาษณ์คุณ):

- นี่มีไว้เพื่อใคร? (ให้เจาะจง — “ฉัน” “นักพัฒนาอิสระ” “ชมรมวิ่งของฉัน”)
- ทุกวันนี้พวกเขาต้องทำอะไรที่ลำบากโดยไม่มีสิ่งนี้?
- ความสำเร็จหน้าตาเป็นอย่างไรในหนึ่งประโยค?

คุณให้ AI ช่วยขุดคุ้ยได้ ลองพรมตนี้ดู:

```
I have a rough idea: a web app to track books I'm reading.
Act as a product thinking partner. Ask me 5 sharp questions,
one at a time, to clarify the problem, the target user, and what
"done" looks like. Don't propose features yet – just interrogate
the idea until it's concrete.
```

การตอบคำถามดี ๆ ห้าข้อมักจะช่วยลบไอเดียคลุมเครือให้กลายเป็นสิ่งที่จับต้องได้ เคล็ดลับคือปล่อยให้ AI โต้แย้งกลับแทนที่จะเห็นด้วยกับทุกอย่าง ถ้ามันถามว่า “คุณต้องซิงก์ข้อมูลข้ามอุปกรณ์ไหม?” แล้วคำตอบที่ซื่อสัตย์ของคุณคือ “ไม่ใช่แค่แค่แล็ปท็อปของฉันเครื่องเดียว” คุณก็เพิ่งตัดแบ็กเอนด์ ระบบยืนยันตัวตน และงานหลายสัปดาห์ทิ้งไป — ก่อนจะเขียนโค้ดสักบรรทัด นั่นคือประเด็นทั้งหมด: จุดที่ถูกที่สุดในการตัดฟีเจอรคือในบทสนทนา ไม่ใช่ในโค้ดเบส

สังเกตช่วงเวลาที่คุณเปลี่ยนจาก “อืม คงจะใช้มั้ง” มาเป็น “ใช่ อย่างนั้นแหละ” การเปลี่ยนแปลงนั้นคือสัญญาณว่าคุณเจอปัญหาที่แท้จริงแล้ว และพร้อมเดินหน้าต่อไป

ขั้นตอนที่ 2: กำหนดขอบเขตและสิ่งที่ไม่ทำ

นี่คือขั้นตอนที่มีค่าที่สุดและมักถูกข้ามบ่อยที่สุด สิ่งที่ไม่ทำ (non-goals) — สิ่งที่คุณตั้งใจ ไม่ สร้าง — ปกป้องคุณจาก AI ที่ชอบเพิ่มของแถม “ที่น่าจะมีประโยชน์”

สำหรับแอปติดตามหนังสือของเรา ขอบเขตอาจเป็นแบบนี้:

- อยู่ในขอบเขต: เพิ่มหนังสือ ทำเครื่องหมายว่ากำลังอ่าน/อ่านจบ ดูรายการหนังสือ ให้คะแนนหนังสือที่อ่านจบแล้ว
- นอกขอบเขต (สิ่งที่ไม่ทำ): ฟีดเจอร์โซเชียล ระบบแนะนำหนังสือ การนำเข้าจาก Goodreads แอปมือถือ ระบบบัญชีผู้ใช้

ลองนึกภาพสองคอลัมน์: ทุกอย่างที่คุณตกลงว่าจะสร้างอยู่ทางซ้าย ทุกอย่างที่คุณตั้งใจข้าม ไปอยู่ทางขวา เส้นแบ่งระหว่างสองฝั่งนี้คือขอบเขตที่ทำให้เวอร์ชัน 1 มีขนาดเล็กพอที่จะเสร็จได้จริง

IN SCOPE	NON-GOALS
Add a book	Social features
Mark reading/done	Recommendations
See a list	Goodreads import
Rate finished books	Mobile app
	User accounts

build now ← → not now

การเขียนสิ่งที่ไม่ทำลงไปทำหน้าที่สองอย่าง มันทำให้เวอร์ชันแรกของคุณเล็กพอที่จะทำเสร็จได้จริง และมันให้ประโยคหนึ่งที่คุณสามารถแปะกลับไปได้เมื่อ AI เริ่มหลงทาง: “อย่าลืมว่าระบบบัญชีเป็นสิ่งที่ไม่ทำสำหรับเวอร์ชัน 1 — อย่าเพิ่มระบบล็อกอิน”

สิ่งที่ไม่ทำไม่ได้แปลว่า “ไม่มีวัน” — มันแปลว่า “ยังไม่ใช่ตอนนี้” การที่ระบบบัญชีผู้ใช้เป็นสิ่งที่ไม่ทำ ไม่ได้หมายความว่าแอปของคุณจะไม่มีมันตลอดไป มันแค่หมายความว่ามันไม่ใช่ส่วนหนึ่งของเวอร์ชันนี้ ดังนั้นคุณจึงไม่ต้องจ่ายต้นทุนของมันวันนี้ การมองมุมนี้ใหม่สำคัญมาก: มันช่วยให้คุณปฏิเสธฟีดเจอร์หนึ่งได้โดยไม่รู้สึกว่าการกำลังทิ้งไอเดียดี ๆ ไป คุณแค่กำลังจัดลำดับก่อนหลังเท่านั้น รายการส่วนใหญ่ในลิสต์สิ่งที่ไม่ทำของคุณ แท้จริงแล้วคือรายการงานค้างสำหรับเวอร์ชัน 2 ที่แฝงตัวอยู่ และนั่นก็โอเค — เขียนมันเก็บไว้ที่ไหนสักที่แล้วเดินหน้าต่อ

ขั้นตอนที่ 3: เขียน User Story

User story แปลงขอบเขตงานให้กลายเป็นพฤติกรรมที่ทดสอบได้ รูปแบบของมันเรียบง่าย:

ในฐานะ [ผู้ใช้] ฉันต้องการ [ทำสิ่งนี้] เพื่อที่จะ [ได้ประโยชน์นี้]

สำหรับแอปติดตามหนังสือ:

- ในฐานะผู้อ่าน ฉันต้องการเพิ่มหนังสือด้วยชื่อเรื่องและผู้เขียน เพื่อที่จะเริ่มติดตามมันได้
- ในฐานะผู้อ่าน ฉันต้องการทำเครื่องหมายหนังสือว่าอ่านจบแล้ว เพื่อที่จะเห็นความคืบหน้าของฉัน
- ในฐานะผู้อ่าน ฉันต้องการให้คะแนนหนังสือที่อ่านจบแล้ว 1-5 ดาว เพื่อที่จะจำได้ว่าฉันชอบเล่มไหน

แต่ละ story คือหน่วยงานที่คุณสร้างและตรวจสอบได้อย่างอิสระ ถ้าคุณนึกภาพไม่ออกว่าจะทดสอบ story นั้นด้วยการคลิกไปมาในแอปได้อย่างไร แปลว่ามันคลุมเครือเกินไป — ให้แยกมันออกหรือเขียนใหม่

ส่วน □ □ □ □ □ □ □ □ คือส่วนที่คนมักตัดทิ้ง ทั้งที่มีค่าที่สุด “ในฐานะผู้อ่าน ฉันต้องการซ่อนค้นหา” บอก AI ว่า ต้องสร้างอะไร แต่ไม่บอก ทำไม จึงทำให้มันตัดสินใจแลกเปลี่ยน (trade-off) ที่ดีไม่ได้ “ในฐานะผู้อ่าน ฉันต้องการหาหนังสือในรายการของฉันโดยพิมพ์ชื่อเรื่องบางส่วน เพื่อที่จะไม่ต้องเลื่อนดูรายการหลายสิบรายการ” บอกงานที่แท้จริง — และตอนนี้ตัวกรองแบบพิมพ์แล้วกรองทันทีแบบง่าย ๆ อาจดีกว่าเสิร์ชเอนจินขนาดใหญ่ก็ได้ ประโยชน์คือสเปกของสเปกอีกที: มันอธิบายฟีดเจอร์ได้ดีพอที่ AI (และตัวคุณในอนาคต) จะแยกแยะได้ว่าการทำงานแบบไหนถูกต้องและแบบไหนผิด

ขั้นตอนที่ 4: ร่าง PRD แบบเบา ๆ

PRD (Product Requirements Document หรือเอกสารข้อกำหนดผลิตภัณฑ์) ก็แค่การรวมการตัดสินใจของคุณไว้ในที่เดียว มันมีโครงสร้างที่คาดเดาได้ — หมวดเต็ม ๆ ไม่ก็หมวดทุกครั้ง:

```
PRD
├─ Problem      why this exists
├─ User         who it's for
├─ Goal         what it does, in one line
├─ In Scope     what v1 includes
├─ Non-Goals    what v1 skips
├─ User Stories as a __, I want __, so that __
├─ Data Model   the fields you store
└─ Done When    the acceptance test
```

ให้มันเล็กเข้าไว้ นี่คือ PRD เริ่มต้นฉบับสมบูรณ์ที่คุณส่งให้ AI ได้เลย:

```
# Book Tracker – Mini PRD

## Problem
I read a lot but lose track of what I've finished and what I thought of it.

## User
Me (a single user). No accounts needed for v1.

## Goal
A simple web app to add books, track reading status, and rate finished ones.

## In Scope (v1)
- Add a book (title, author)
- Set status: To Read / Reading / Finished
- List all books, filterable by status
- Rate finished books 1–5 stars

## Non-Goals (v1)
- User accounts / login
- Social features, sharing, recommendations
- Importing from other services
- Mobile / native apps

## Tech (keep it simple)
- Single-page web app
- Local browser storage (no backend for v1)

## Done When
I can add a book, change its status, rate it, and see my list
persist after a page refresh.
```

สังเกตบรรทัด “Done When” นั่นคือแบบทดสอบยอมรับ (acceptance test) ของคุณ — วิธีที่จับต้องได้ในการรู้ว่าคุณส่งมอบเวอร์ชัน 1 สำเร็จแล้ว ต้องมีมันเสมอ “Done When” ที่ดีคือสิ่งที่คุณ ลงมือทำได้จริง: คลิกตรงนี้ พิมพ์นี้ รีเฟรช แล้วเห็นสิ่งนั้น ถ้าเงื่อนไขความสำเร็จของคุณอ่านเหมือนความรู้สึก (“เมื่อมันดูชัดเกลาแล้ว” “เมื่อมันรู้สึกดี”) มันจะไม่มีวันเสร็จ เพราะไม่มีอะไรให้ตรวจสอบเทียบ ทำให้มันเป็นสคริปต์ที่คุณส่งให้คนแปลกหน้าทำตามได้

คุณไม่จำเป็นต้องเขียน PRD ด้วยมือเอง วิธีที่มีประโยชน์คือป้อนคำตอบจากขั้นตอนก่อนหน้ากลับไปให้ AI แล้วให้มัน

ประกอบร่างขึ้นมา จากนั้นคุณค่อยแก้ไข จุดสำคัญของการแก้ไขแทนที่จะรับมาทั้งหมดคือ คุณ ต้องเข้าใจและเห็นด้วยกับทุกบรรทัด — เอกสารนี้คือแหล่งความจริงที่คุณจะแปะกลับไปเรื่อย ๆ ดังนั้นถ้ามีข้อผิดพลาดตรงนี้ มันจะถูกคัดลอกไปทุกเซสชันที่ตามมา

ขั้นตอนที่ 5: ร่างโมเดลข้อมูล

แอปส่วนใหญ่แท้จริงแล้วเกี่ยวกับ ข้อมูล: คุณเก็บอะไรและชิ้นส่วนต่าง ๆ เชื่อมโยงกันอย่างไร การร่างโมเดลข้อมูลอย่างรวดเร็วช่วยให้คุณจากโค้ดที่สับสนและไม่สอดคล้องกันในภายหลัง คุณไม่จำเป็นต้องมีไดอะแกรมฐานข้อมูล — แค่รายการฟิลด์ก็เพียงพอ

สำหรับแอปติดตามหนังสือ มีสิ่งหลักอยู่หนึ่งอย่าง — Book:

```
Book
- id: unique string
- title: string
- author: string
- status: "to_read" | "reading" | "finished"
- rating: number 1-5 (only if finished)
- addedAt: date
```

แม้แต่ภาพร่างเล็ก ๆ นี้ก็ทำให้การตัดสินใจชัดเจนขึ้น: คะแนนมีอยู่เฉพาะหนังสือที่อ่านจบแล้วเท่านั้น สถานะเป็นหนึ่งในสามค่าที่กำหนดไว้ตายตัว และหนังสือทุกเล่มมี id เมื่อคุณส่งสิ่งนี้ให้ AI มันจะหยุดเดาชื่อฟิลด์และสร้างงานได้อย่างสอดคล้อง

เหตุผลที่ต้องปึกหมุดเรื่องนี้ตั้งแต่เนิ่น ๆ คือชื่อฟิลด์รั่วไหลไปทุกที่ เมื่อ AI เขียนโค้ดที่อ่าน book.rating แล้ว ชื่อนั้นจะปรากฏในฟอร์ม หน้ารายการ ขั้นตอนการจัดเก็บข้อมูล และพีเจอาร์ใด ๆ ที่คุณเพิ่มในภายหลัง ถ้าคุณปล่อยให้มันเลือนไหล — rating ในที่หนึ่ง stars ในอีกที่ score ในอีกที่ — คุณจะไต่บันไดที่ตามแก้ยากลำบาก โมเดลข้อมูลหกบรรทัดคือประกันภัยที่ถูกต้องที่สุดต่อปัญหานี้

บล็อกสเปกสั้น ๆ ที่คัดลอกวางได้ มักได้ผลดีกว่าข้อความบรรยายสำหรับ AI นี่คือโมเดลเดียวกันเขียนเป็นพรอมต์ที่คุณส่งได้จริง:

```
Use this exact data shape for a book. Don't add fields I didn't list,
and don't rename these:
```

```
Book {
  id      - unique string, generated on creation
  title   - string, required
  author  - string, required
  status  - one of: "to_read" | "reading" | "finished"
  rating  - integer 1-5, present ONLY when status is "finished"
  addedAt - ISO date string, set when the book is added
}
```

If you think a field is missing, ask me before adding it.

บรรทัดสุดท้ายนั้น — ถามฉันก่อนเพิ่ม — กำลังทำหน้าที่สำคัญจริง ๆ มันเปลี่ยน AI จากคนเดาสุ่มให้กลายเป็นผู้ร่วมงานและทำให้โมเดลข้อมูลยังเป็นของคุณอยู่

ขั้นตอนที่ 6: แบ่งสเปกออกเป็นงานขนาดพอดีสำหรับ Vibe Coding

ก้าวสุดท้ายคือการแบ่งสเปกของคุณออกเป็นงานที่ AI ทำได้ ที่ละอย่าง งานขนาดพอดีสำหรับ vibe coding คืองานที่เล็กพอให้คุณตรวจสอบผลลัพธ์ได้ในหนึ่งนาทีและรู้ว่ามันใช้ได้หรือไม่ ถ้างานหนึ่งชิ้นต้องแก้หลายไฟล์และคุณบอกไม่ได้ว่ามันถูกต้องหรือไม่ แปลว่ามันใหญ่เกินไป

งานที่ดีสำหรับแอปติดตามหนังสือ:

1. ตั้งค่าโครงหน้าพื้นฐานพร้อมส่วนหัวและรายการหนังสือที่ยังว่างเปล่า
2. สร้างฟอร์ม “เพิ่มหนังสือ” (ชื่อเรื่อง + ผู้เขียน) ที่เพิ่มรายการเข้าไปในหน่วยความจำ
3. เพิ่มสถานะ (To Read / Reading / Finished) พร้อมดรอปดาวน์สำหรับเปลี่ยนสถานะ
4. บันทึกรายการลงในที่เก็บข้อมูลท้องถิ่นของเบราว์เซอร์ เพื่อให้ข้อมูลคงอยู่หลังรีเฟรช
5. เพิ่มตัวควบคุมการให้คะแนนที่ปรากฏเฉพาะหนังสือที่อ่านจบแล้ว
6. เพิ่มปุ่มกรองสถานะไว้เหนือรายการ

แต่ละงานสอดคล้องกับ user story หรือบรรทัดใน PRD หนึ่งบรรทัด สร้างต่อยอดจากงานก่อนหน้านี้ และจบลงด้วยสิ่งที่คุณ เห็น ว่าใช้งานได้ นั่นคือจังหวะของ vibe coding: งานเล็ก รันมัน ตรวจสอบ แล้วต่อไป

คุณให้ AI สร้างการแบ่งงานนี้ให้คุณได้ด้วยซ้ำ:

```
Here is my mini PRD and data model [paste them].  
Break this into 6–8 small, ordered build tasks. Each task should  
be independently testable and build on the previous one. Keep each  
task to something I can verify by clicking around the app.
```

รักษาสเปกให้มีชีวิตอยู่

สเปกไม่ใช่อนุสาวรีย์ที่คุณสร้างครั้งเดียวแล้วชื่นชม มันคือเอกสารมีชีวิต และความผิดพลาดที่พบบ่อยที่สุดคือการปฏิบัติต่อมันเหมือนสิ่งที่ตายตัว ระหว่างที่คุณสร้างงาน คุณจะเรียนรู้สิ่งที่ขั้นตอนวางแผนบอกคุณไม่ได้: ฟีเจอร์หนึ่งยากกว่าที่คิด สิ่งที่ไม่ทำอย่างหนึ่งกลับกลายเป็นสิ่งจำเป็น user story หนึ่งกลับเป็นสอง story ที่ซ่อนอยู่ เมื่อสิ่งนั้นเกิดขึ้น ให้แก่สเปก ก่อน แล้วค่อยบอก AI ให้ทำตามฉบับใหม่ สเปกนำ โค้ดตาม

นิสัยไม่ก็อย่างช่วยให้มันยังคงชัดเจน:

- อัปเดตทันทีที่การตัดสินใจเปลี่ยนไป สเปกที่พูดอย่างหนึ่งในขณะที่โค้ดของคุณทำอีกอย่างหนึ่งนั้นแย่งยิ่งกว่าไม่มีสเปกเลย — มันหลอกลวงเซสชัน AI ครั้งต่อไปอย่างเจ็บ ๆ
- ชัดขวางงานออกทันทีที่ทำเสร็จ รายการงานทำหน้าที่เป็นตัวติดตามความคืบหน้าไปในตัว การเห็นหกในแปดรายการถูกติ๊กแล้วก็เป็นแรงผลักดันในตัวของมันเอง
- ย้ายสิ่งที่ไม่คาดคิดไปไว้ในสิ่งที่ไม่ทำหรือลิสต์ “ไว้ทีหลัง” แทนที่จะลงมือทำมันเลย ถ้าฟีเจอร์ที่น่าดึงดูดปรากฏขึ้นกลางทาง ให้จดมันไว้แล้วเดินหน้าต่อ คุณตัดสินใจได้ทีหลังว่าสมควรมีที่ยืนในเวอร์ชัน 1 หรือไม่

นี่คือวินัยที่แยก “ฉันสร้างบางอย่างขึ้นมา” ออกจาก “ฉันสร้างสิ่งที่ฉันตั้งใจจะสร้าง” สเปกคือวิธีที่ทำให้คุณยังคงเป็นผู้เขียนโปรเจกต์ของตัวเอง แทนที่จะเป็นผู้โดยสาร

กับดีกที่พบบ่อย: ระบุทางแก้ ไม่ใช่ระบุปัญหา

ความล้มเหลวรูปแบบหนึ่งควรถูกพูดถึงแยกออกมา เพราะมันตกหลุมได้ง่ายมาก เมื่อคุณนั่งลงเพื่อเขียนสเปก สิ่งที่ยั่ววนใจคือการอธิบาย ทางแก้ ที่คุณจินตนาการไว้แล้ว — “แถบด้านข้างที่มีแท็บ” “หน้าต่างป๊อปอัพที่โผล่ขึ้นมา” “ตาราง Postgres” แต่สเปกที่เต็มไปด้วยทางแก้กลับล่อการตัดสินใจที่คุณยังไม่ได้พิสูจน์ว่าคุณค่าไว้อย่างเจ็บ ๆ

ลองเปรียบเทียบสองประโยคนี้:

```
Bad: Add a left sidebar with collapsible sections for each book status.  
Good: Let me see my books grouped by status without losing the full list.
```

ประโยคแรกได้เลือกแถบด้านข้าง ส่วนที่พับเก็บได้ และเลย์เอาต์ไปแล้ว — ก่อนที่ใครจะยืนยันด้วยซ้ำว่านั่นคือการตัดสินใจที่ถูกต้อง ประโยคที่สองระบุ งานที่ต้องทำให้สำเร็จ และปล่อยให้ AI (และตัวคุณ) เป็นอิสระในการหาทางแก้ที่ง่ายที่สุดที่ตอบโจทย์นั้น ซึ่งอาจเป็นแค่ปุ่มกรองสามปุ่มก็ได้ อธิบายผลลัพธ์ที่คุณต้องการ ปล่อยให้วิธีการยังคงต่อรองได้นานที่สุดเท่าที่จะทำได้ ทางแก้เปลี่ยนแปลงได้ง่ายในสเปก แต่แพงเมื่อต้องเปลี่ยนในโค้ด

ผลตอบแทน

ตอนนี้คุณมีคำอธิบายปัญหา ขอบเขตงานและสิ่งที่ไม่ทำ user story, PRD หนึ่งหน้า โมเดลข้อมูล และรายการงานเรียงลำดับ — รากฐานทั้งหมดของโปรเจกต์จริง ร่างขึ้นภายในไม่กี่นาทีโดยมี AI เป็นกระดานระดมความคิด

ทั้งหมดนี้ไม่จำเป็นต้องให้คุณรู้วิธีเขียนโค้ด สิ่งที่ต้องมีคือการรู้ว่าต้องการอะไร และคิดให้ชัดเจนสักไม่กี่นาที — ส่วนที่เหลือ AI จัดการคิดออกมาดัง ๆ ร่วมกับคุณ นั่นคือการเปลี่ยนแปลงที่แท้จริง: คุณยังไม่ได้เขียนซอฟต์แวร์ คุณกำลังเขียน เจตนา และเจตนาที่ดีคือชัยชนะไปกว่าครึ่งแล้ว

เก็บสเปกนี้ไว้ในไฟล์ในโปรเจกต์ของคุณ (หลายคนใช้ SPEC.md หรือ PRD.md) มันจะกลายเป็นเอกสารที่คุณแปะลงในเซสชัน AI ใหม่เพื่อยึดโยงบริบทกลับมา และเป็นเช็คลิสต์ที่คุณไล่ทำที่ละงานขนาดพอดีสำหรับ vibe coding ในบทถัดไป เราจะหยิบงานแรกขึ้นมาและลงมือสร้างมันจริง ๆ

สรุปและฝึกปฏิบัติ

สิ่งสำคัญที่ควรจำ

- สเปกคือบันทึกทำงานสั้น ๆ เขียนด้วยภาษาธรรมดา — สิ่งที่คุณกำลังสร้างและไม่ได้กำลังสร้าง — ไม่ใช่สัญญาที่ตรึงไว้ตายตัว หนึ่งหน้าก็เพียงพอแล้ว
- ทำความเข้าใจปัญหาและตัวคนก่อนพูดถึงฟีเจอร์ และเขียนสิ่งที่ไม่ทำลงไป: มันคือเกราะป้องกันที่คมที่สุดของคุณจาก AI ที่ชอบเพิ่มของแถม “ที่น่าจะมีประโยชน์”
- User story ที่มีส่วน □ □ □ □ □ □ □ □ และ PRD เล็ก ๆ ที่มีบรรทัด “Done When” ที่จับต้องได้ จะเปลี่ยนเจตนาให้กลายเป็นพฤติกรรมที่ทดสอบได้และสร้างได้จริง
- ปักหมุดโมเดลข้อมูลไว้ตั้งแต่เนิ่น ๆ — ชื่อฟิลด์รั่วไหลไปทุกที่ — และแบ่งสเปกเป็นงานขนาดพอดีสำหรับ vibe coding ที่คุณตรวจสอบได้ที่ละอย่าง
- อธิบายปัญหา ไม่ใช่ทางแก้ ผลลัพธ์ยังคงเปลี่ยนแปลงได้ง่ายในสเปกแต่แพงเมื่อต้องเปลี่ยนในโค้ด รักษาสเปกให้มีชีวิตอยู่ในขณะที่คุณเรียนรู้

ลองทำดู

หยิบไอเดียที่คุณเก็บไว้เฉย ๆ มาสักอันแล้วแปลงมันเป็นสเปกหนึ่งหน้าในเซสชัน AI เดียว ให้ AI สัมภาษณ์คุณด้วยคำถามคมคาย 5 ข้อก่อน จากนั้นเขียน — ด้วยคำพูดของคุณเอง — ปัญหา ผู้ใช้ user story สามข้อ (แต่ละข้อมีส่วน □ □ □ □ □ □ □ □) ลิสต์สิ่งที่ไม่ทำอย่างชัดเจน โมเดลข้อมูลหกบรรทัด และบรรทัด “Done When” หนึ่งบรรทัดที่คุณลงมือทำได้จริง บันทึกไว้เป็น SPEC.md แบบทดสอบสเปกที่ดี: คุณควรแปะมันลงในแชทใหม่เอี่ยมได้ในวันพรุ่งนี้ และ AI จะรู้แน่ชัดว่าต้องสร้างอะไร

พร้อมต์ประจำบท

```
I have a rough idea: <one-line description>.
```

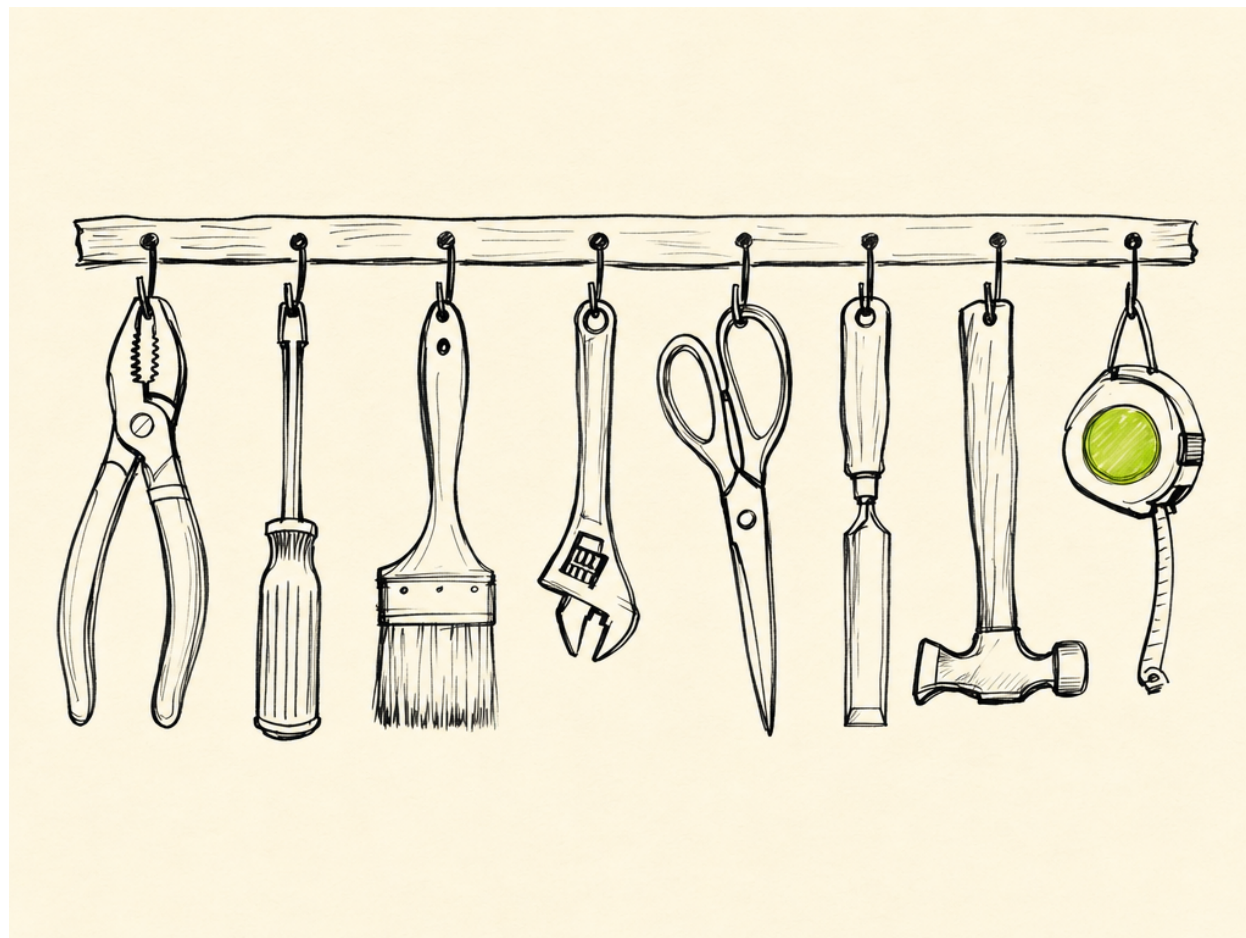
```
Act as my product thinking partner. First, ask me 5 sharp questions, one at a time, to clarify the problem, the target user, and what "done" looks like. Don't propose features yet.
```

```
After I answer, draft a one-page spec with these sections:
```

- Problem – User – Goal
- In Scope (v1) – Non-Goals (v1)
- User stories (each "As a __, I want __, so that __")
- Data model (a short list of fields)
- Done When (a concrete, physically-performable acceptance test)

```
Keep it tight – under one page. If something is ambiguous, ask before assuming. I'll edit the draft until I agree with every line.
```

เลือกสแตกของคุณ: ภาษาต่าง ๆ แบบเห็นภาพรวม



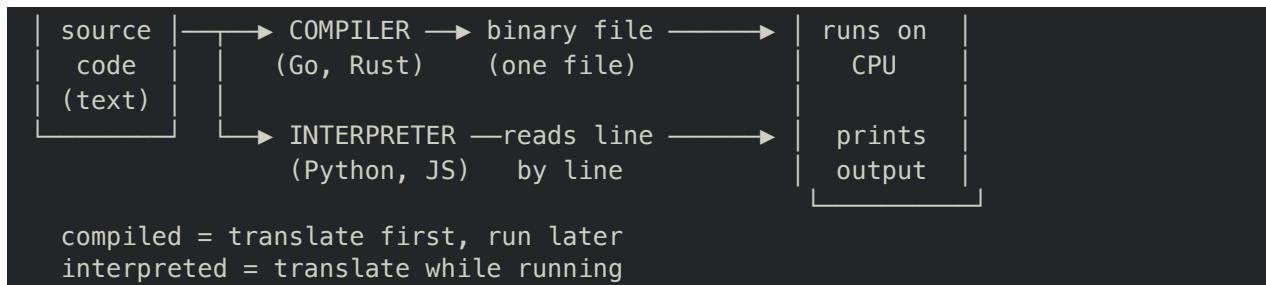
เวลาคุณสั่ง AI ให้สร้างซอฟต์แวร์ ภาษาที่คุณเลือกยังคงสำคัญมาก — AI เป็นคนเขียนโค้ด แต่คุณคือคนที่ต้องอยู่กับผลลัพธ์: คุณจะส่งมอบงานได้เร็วแค่ไหน ดีพลอยง่ายแค่ไหน และเจอทางตันบ่อยแค่ไหน ชาวดีคือคุณไม่จำเป็นต้องเชี่ยวชาญภาษาเหล่านี้เลย แค่รู้พอที่จะเลือกได้อย่างมีเหตุผล และรู้ทันเวลาที่ AI กำลังพาคุณเข้าไปในเส้นทางที่เจ็บปวด

บทนี้จะพาไปดูภาษาที่คุ้มค่าแก่การใส่ใจ แต่ละภาษาเก่งเรื่องอะไรจริง ๆ มีจุดอ่อนตรงไหน และ — สำคัญมากสำหรับ vibe coding — AI ผู้ช่วยจัดการภาษานั้นได้ดีแค่ไหน ปิดท้ายด้วยโค้ดแบบง่ายสุด ๆ สำหรับเลือกภาษาตามเป้าหมายของคุณ

ก่อนเริ่ม มีข้อสังเกตหนึ่งอย่าง: AI ผู้ช่วยเก่งกว่ามาก ๆ กับภาษายอดนิยมที่มีโค้ดสาธารณะกองเป็นภูเขา เมื่อเทียบกับภาษาเฉพาะกลุ่ม (niche) นี่ไม่ใช่ผลกระทบเล็ก ๆ มันมักเป็นปัจจัยเดียวที่ใหญ่ที่สุดว่าโปรเจกต์ของคุณจะราบรื่นหรือไม่ ความนิยมคือฟิเจอร์อย่างหนึ่ง เมื่อภาษาและเฟรมเวิร์กหลักของมันปรากฏนับล้านครั้งในโค้ดสาธารณะ AI ก็เคยเห็นแพตเทิร์นทั่วไปทุกแบบ ข้อผิดพลาดทั่วไปทุกแบบ และวิธีแก้ที่เป็นธรรมชาติของภาษานั้นทุกแบบ แต่เมื่อมันไม่ได้เห็น คุณจะต้องจ่ายค่าความขาดแคลนนั่นด้วยความพยายามที่ล้มเหลวและคำตอบที่ฟังดูมั่นใจแต่ไม่จริง

ก่อนจะเข้าสู่รายชื่อภาษา มีภาพหนึ่งที่ควรจำไว้ในหัว: โค้ดที่คุณเขียนไม่ใช่สิ่งที่เครื่องรันจริง ๆ มีบางอย่างอยู่ตรงกลางที่แปลงข้อความของคุณให้กลายเป็นซอฟต์แวร์ที่ทำงานได้ — อาจเป็น คอมไพเลอร์ (compiler แปลทุกอย่างล่วงหน้าให้เป็นไฟล์ที่คุณรันได้) หรือ อินเทอร์พรีเตอร์ (interpreter อ่านและรันทีละบรรทัด) ภาษาที่คุณเลือกจะกำหนดว่าคุณอยู่บนเส้นทางไหน





JavaScript / TypeScript

นี่คือตัวเลือกเริ่มต้นสำหรับ vibe coding ส่วนใหญ่ และมีเหตุผลรองรับ มันรันได้ทุกที่ — เบร่าวเซอร์ เซิร์ฟเวอร์ (Node.js) ฟังก์ชันแบบ edge — และเป็นภาษาของฝั่ง frontend เว็บทั้งหมด TypeScript คือ JavaScript ที่เพิ่มระบบชนิดข้อมูล (type system) เข้าไป และคุณควรเลือก TypeScript แทบทุกครั้ง: ชนิดข้อมูลช่วยจับความผิดพลาดได้ตั้งแต่เนิ่น ๆ และที่สำคัญคือมันให้ราวกัน (guardrail) แก่ AI ที่ทำให้ผลลัพธ์ของมันน่าเชื่อถือขึ้น ลายเซ็นฟังก์ชันที่มีชนิดข้อมูลชัดเจนคือสัญญาณที่ผู้ช่วยอ่านได้ ถ้าไม่มีสิ่งนี้ AI ก็ต้องเดาว่าข้อมูลของคุณมีรูปร่างแบบไหน และการเดาคือจุดที่บั๊กมักซ่อนตัวอยู่

- เก่งด้าน: เว็บแอป (ทั้ง frontend และ backend), ทุกอย่างที่ใช้คลิกในเบร่าวเซอร์, โปรโตไทป์เร็ว ๆ, โปรเจกต์แบบ full-stack ในภาษาเดียว, การดีพลอยแบบ serverless/edge
- จุดที่เจ็บ: ระบบนิเวศเปลี่ยนแปลงเร็วและทำของเดิมพัง; ปัญหา dependency บวมพองมีจริง (แอปเล็ก ๆ อาจดึงแพ็คเกจมาเป็นร้อย); ความหลวมของ JavaScript ส่วน ๆ เปิดช่องให้บั๊กเล็ก ๆ หลุดรอดไปได้ — เป็นอีกเหตุผลให้ใช้ TypeScript
- กรณีใช้งานทั่วไป: แดชบอร์ด SaaS (Software as a Service — ซอฟต์แวร์ที่เช่าใช้ออนไลน์แทนการติดตั้ง), แลนด์เพจ, อีคอมเมิร์ซ, แอปแบบเรียลไทม์, แอ็ชเชนชันเบร่าวเซอร์, ทุกอย่างที่คุณดีพลอยไปยัง Vercel, Netlify หรือ Cloudflare
- AI จัดการได้แค่ไหน: ยอดเยี่ยม นี่คือภาษาที่มีตัวแทนมากที่สุดในข้อมูลฝึกฝน ผู้ช่วยเขียน React, Next.js และ Node ได้อย่างมั่นใจและแทบไม่ติดขัด ถ้าไม่มีเหตุผลหนักแน่นให้เลือกอย่างอื่น ให้เริ่มที่นี่

มีข้อแลกเปลี่ยนที่เป็นรูปธรรมอันหนึ่งที่ควรรู้: ความเร็วที่ทำให้ JavaScript เยี่ยมสำหรับการทำโปรโตไทป์ ก็คือความเร็วเดียวกันที่ทำให้มันปั่นป่วนไม่หยุด บทเรียนจากสิบแปดเดือนก่อนอาจใช้ API ที่เลิกใช้แล้วไปเสียแล้ว และ AI — ซึ่งฝึกจากภาพนิ่งของอดีต — บางครั้งก็ยื่นแพ็คเกจที่ล้าสมัยมาให้คุณ เมื่อสิ่งที่มันเขียนใช้ไม่ได้ สาเหตุมักมาจากไลบรารีที่เปลี่ยนไปแล้ว วิธีแก้มักจะเป็นการบอกเวอร์ชันที่คุณใช้และขอให้มันตรวจสอบเอกสารล่าสุด นี่คือภาษาที่คุณต้องจ่ายเพื่อได้อยู่ในระบบนิเวศที่กระตือรือร้นที่สุดในโลก และสำหรับโปรเจกต์ส่วนใหญ่ก็คุ้มค่าที่จะจ่าย

Python

Python อ่านเกือบเหมือนภาษาอังกฤษ จึงเป็นภาษาที่เป็นมิตรที่สุดสำหรับผู้เริ่มต้น และเป็นภาษากลางของงานข้อมูล งานสคริปต์ และ AI เอง ถ้าโปรเจกต์ของคุณเกี่ยวข้องกับแมชชีนเลิร์นนิง การวิเคราะห์ข้อมูล ระบบอัตโนมัติ หรือการคำนวณเชิงวิทยาศาสตร์ Python มักเป็นคำตอบ

- เก่งด้าน: AI/ML, การประมวลผลข้อมูล, สคริปต์อัตโนมัติ, backend API (API หรือ Application Programming Interface คือประตูที่โปรแกรมหนึ่งใช้คุยกับอีกโปรแกรมหนึ่ง), โค้ดเชื่อมต่อ (glue code), ทุกอย่างที่มีไลบรารีอยู่แล้ว (และสำหรับ Python มักจะมีจริง ๆ)
- จุดที่เจ็บ: ซ้ำเมื่อต้องคำนวณตัวเลขดิบ ๆ เว้นแต่จะพึ่งพาไลบรารีที่เขียนด้วย C อยู่ข้างใน; “dependency hell” กับ virtual environment และเวอร์ชันแพ็คเกจทำให้มือใหม่งง; ไม่เหมาะกับ frontend ของเบร่าวเซอร์
- กรณีใช้งานทั่วไป: การสแครปข้อมูล, ไปป์ไลน์ข้อมูล, การทดลอง ML, บอท Discord/Telegram, เครื่องมือภายในองค์กร, REST (Representational State Transfer — สไตล์ทั่วไปของประตู API เหล่านั้น) API (FastAPI, Django, Flask)
- AI จัดการได้แค่ไหน: ยอดเยี่ยม เทียบเท่ากับ JavaScript ผู้ช่วยคล่องแคล่วมากกับ Python และไลบรารีหลักของมัน เป็นตัวเลือกเริ่มต้นที่ดีเยี่ยมสำหรับโปรเจกต์ที่ไม่ใช่เว็บ

จุดที่ Python มักสะดุดมือใหม่อยู่เสมอคือสภาพแวดล้อม (environment) ไม่ใช่ตัวภาษาเอง สคริปต์ที่รันได้ดีสำหรับ AI ในโพลเดอร์หนึ่งอาจล้มเหลวในอีกโพลเดอร์ เพราะแพ็คเกจอยู่ใน virtual environment ที่ไม่ได้เปิดใช้งาน หรือเพราะสองโปรเจกต์ต้องการไลบรารีเดียวกันคนละเวอร์ชัน ทั้งหมดนี้ไม่ใช่ความผิดของโค้ดคุณ และไม่ยากเลยเมื่อคุณเคยเจอมันแล้ว — แต่มันคือช่วงเวลา “ทำไมมันไม่รันสักที” ที่พบบ่อยที่สุดใน Python ให้ขอ AI ตั้งค่า virtual environment และไฟล์ requirements.txt (หรือใช้เครื่องมืออย่าง uv หรือ Poetry) ตั้งแต่เริ่มต้น ความปวดหัวส่วนใหญ่จะหายไป

Go

Go ถูกออกแบบที่ Google ให้เรียบง่าย รวดเร็ว และดีพลอยง่าย มันคอมไพล์เป็นไบนารีแบบครบในตัวไฟล์เดียวที่คุณวางลงบนเซิร์ฟเวอร์ได้เลยโดยไม่ต้องติดตั้งรันไทม์ใด ๆ มันตั้งใจตัดฟีเจอร์หรูหราออกไป ซึ่งทำให้โค้ดเบสน่าเบื่อในแบบที่ดีที่สุด — และความน่าเบื่อนั้นทั้งมนุษย์และ AI ให้เหตุผลตามได้ง่าย

- เก่งด้าน: บริการ backend ที่รวดเร็ว, เครื่องมือ command-line, งานเครือข่าย, ทุกอย่างที่ต้องรับมือกับการเชื่อมต่อพร้อมกันจำนวนมาก, การดีพลอยที่เรียบง่าย
- จุดที่เจ็บ: การจัดการข้อผิดพลาดที่ยืดยาว (คุณจะได้เห็น if err != nil อยู่ทุกที่); ไม่ใช่ภาษาสำหรับ frontend; ความเรียบง่ายที่ตั้งใจไว้อาจรู้สึกจำกัดถ้าคุณต้องการทางเลือกที่แสดงออกได้มาก
- กรณีใช้งานทั่วไป: API, ไมโครเซอร์วิส, CLI (CLI หรือ Command-Line Interface คือวิธีรันโปรแกรมแบบข้อความล้วนด้วยการพิมพ์คำสั่ง), เครื่องมือด้านโครงสร้างพื้นฐาน, backend ที่มีกราฟฟิกส์สูง
- AI จัดการได้แค่ไหน: ดีมาก ชุดฟีเจอร์ที่เล็กและสม่ำเสมอของ Go หมายความว่า “วิธีที่ถูกต้อง” สำหรับเกือบทุกอย่าง และผู้ช่วยก็ทำตามได้ดี มีข้อมูลฝึกฝนน้อยกว่า JS/Python แต่ความเรียบง่ายช่วยชดเชยได้

ฟีเจอร์เด็ดของ Go สำหรับผู้สร้างคนเดียวคือไบนารีเดียวนั่นเอง ไม่มีขั้นตอนแบบ “ติดตั้ง Node แล้วก็อีกสลิปแพ็คเกจบนเซิร์ฟเวอร์” ไม่มีปัญหาเวอร์ชันรันไทม์ไม่ตรงกันระหว่างเครื่องคุณกับโปรดักชัน คุณ build แล้วก็คัดลอกไฟล์เดียว แล้วก็รันมันเลย แค่นั้นก็ทำให้การดีพลอยมีข้อผิดพลาดน้อยลงอย่างเห็นได้ชัด — ซึ่งสำคัญมากเวลาคุณทำ vibe coding และไม่ยอมมานั่งตีบั๊กโครงสร้างพื้นฐาน ความยืดหยุ่นคือต้นทุนที่ต้องจ่าย: โค้ดที่ Python เขียนแค่สามบรรทัดอาจกลายเป็นแปดบรรทัดใน Go แต่บรรทัดที่เพิ่มมานั้นคาดเดาได้ และความคาดเดาได้นี้แหละคือสิ่งที่ AI จัดการได้ดี

Rust

Rust ให้ความเร็วระดับ C พร้อมกับคอมไพเลอร์ที่ปฏิเสธไม่ให้คุณเขียนบั๊กเกี่ยวกับหน่วยความจำได้ทั้งหมด หมู่ข้อแลกเปลี่ยนคือคอมไพเลอร์ที่เข้มงวดจนขึ้นชื่อและเส้นโค้งการเรียนรู้ที่สูงชัน สำหรับ vibe coding Rust เป็นดาบสองคม: AI เขียนมันได้ แต่ความเข้มงวดของ Rust หมายความว่าคอมไพเลอร์จะปฏิเสธโค้ดบ่อยกว่า นำไปสู่วงจรแก้ไขไปมาที่มากขึ้น

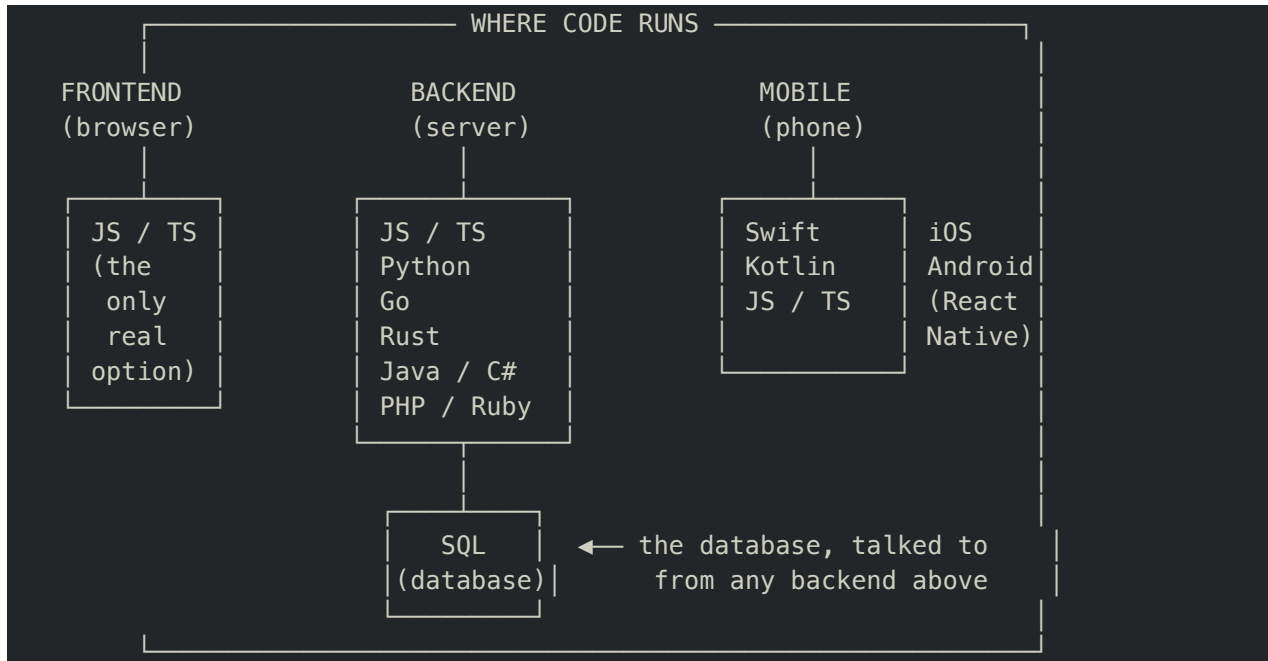
- เก่งด้าน: ซอฟต์แวร์ที่ต้องการประสิทธิภาพสูง, การเขียนโปรแกรมระบบ, เอนจินเกม, WebAssembly, เครื่องมือ command-line ที่ความเร็วสำคัญ, ทุกอย่างที่มีความถูกต้องและความปลอดภัยเป็นเรื่องสำคัญที่สุด
- จุดที่เจ็บ: เส้นโค้งการเรียนรู้สูง; เวลาคอมไพล์ช้า; “borrow checker” สร้างความหงุดหงิดให้มือใหม่; เกินความจำเป็นสำหรับเว็บแอปและสคริปต์ส่วนใหญ่
- กรณีใช้งานทั่วไป: backend ประสิทธิภาพสูง, เครื่องมือ CLI, โมดูล WASM (WebAssembly), ระบบกึ่งฝังตัว (embedded-ish), โครงสร้างพื้นฐานที่ทุกมิลลิวินาทีมีค่า
- AI จัดการได้แค่ไหน: ดีแต่ขรุขระกว่า ผู้ช่วยรู้จัก Rust ดี แต่คอมไพเลอร์ที่เข้มงวดจะเผยข้อผิดพลาดที่ต้องทำซ้ำแล้วซ้ำเล่า คาดหวังวงจร “แก้ error แล้วลองใหม่” มากกว่ากับ Python หรือ JS เลือก Rust เมื่อประสิทธิภาพหรือความปลอดภัยคุ้มค่ากับความเสียดทานจริง ๆ — ไม่ใช่เลือกโดยดีฟอลต์

มีข้อดีแอบแฝงของความเข้มงวดของ Rust ที่มองข้ามได้ง่าย: คอมไพเลอร์คือผู้ริ้วฟรีที่ไม่ปรานีใคร เมื่อ AI เขียน JavaScript แบบหยาบ ๆ บั๊กนั้นจะหลุดออกไปก่อดผู้ใช้งานจริง เมื่อ AI เขียน Rust แบบหยาบ ๆ คอมไพเลอร์มักจะจับได้ก่อนที่โปรแกรมจะรันเลยด้วยซ้ำ วงจรแก้ไขที่เพิ่มขึ้นมารู้สึกเหมือนความเสียดทาน แต่ส่วนหนึ่งของมันคือคอมไพเลอร์กำลังหยุดบั๊กจริงเอาไว้ คำถามที่ควรถามตัวเองอย่างตรงไปตรงมาก่อนจะเอื้อมมือไปหา Rust คือคุณต้องการความเร็วหรือความปลอดภัยนั้นจริง ๆ หรือแค่มีคนบอกว่าคุณควรอยากได้มัน สำหรับโปรเจกต์แรก คุณแทบไม่ต้องการมันหรอก

กล่าวถึงภาษาอื่น ๆ พอสังเขป

คุณจะเจอภาษาเหล่านี้ และบางครั้งภาษาใดภาษาหนึ่งก็เหมาะสมพอดี:

การมองภาพว่าแต่ละภาษามักอยู่ตรงไหนช่วยได้มาก ซอฟต์แวร์มีชั้น — สิ่งที่รันในเบราว์เซอร์ (frontend), สิ่งที่รันบนเซิร์ฟเวอร์ (backend), สิ่งที่รันบนโทรศัพท์ (mobile) — และภาษาส่วนใหญ่ก็มี “บ้านเกิด” ของตัวเอง:



- Java / C#: ม้างานของบริษัทขนาดใหญ่ ทรงตัว มีเครื่องมือเยอะมาก เหมาะกับ backend องค์กรขนาดใหญ่ และ Android (Java/Kotlin) หรือระบบนิเวศ Windows/.NET (C#) AI จัดการทั้งสองได้ดี ในฐานะ vibe coder เดียว คุณแทบไม่จำเป็นต้องใช้มัน เว้นแต่คุณจะทำงานในสแต็กองค์กรที่มีอยู่แล้ว หรือสร้างสำหรับ Android/Windows โดยเฉพาะ
- Swift / Kotlin: ภาษาดั้งเดิมสำหรับแอป iOS (Swift) และ Android (Kotlin) ถ้าคุณกำลังส่งแอปมือถือจริงเข้า App Store หรือ Play Store คุณมักจะต้องแตะภาษาเหล่านี้ AI รองรับได้ดีแต่อ่อนกว่าภาษาเว็บ และการพัฒนาแอปมือถือมีความเสียดทานเพิ่มขึ้น (อุปกรณ์, กฎของ app store, เครื่องมือ build)
- PHP / Ruby: ทหารเก่าของเว็บ และยังคงขับเคลื่อนเว็บไซต์จำนวนมากอย่างเงียบ ๆ PHP ขับเคลื่อน WordPress และด้วยเหตุนี้จึงขับเคลื่อนเว็บไซต์จำนวนมาก; Ruby on Rails ยังคงเป็นวิธีที่รวดเร็วและมีความเห็นชัดเจนในการสร้างแอปที่ผูกกับฐานข้อมูล ทั้งคู่มีข้อมูลฝึกฝนเยอะพอสมควร ดังนั้น AI จัดการได้ดี — แต่ไม่ค่อยเป็นตัวเลือกรายแรกที่มือใหม่จะเอื้อมไปหาในทุกวันนี้ และคุณมักจะเลือกมันเพราะกำลังทำงานในโค้ดเบสที่มีอยู่แล้วมากกว่าจะเริ่มใหม่
- SQL (Structured Query Language): ไม่ใช่ภาษาสำหรับใช้งานทั่วไป แต่คุณจะเจอมันทันทีที่แอปของคุณเก็บข้อมูล ให้นึกว่ามันเป็นภาษาที่คุณใช้ถามคำถามฐานข้อมูล — “ขอรายชื่อผู้ใช้ทุกคนที่สมัครสัปดาห์นี้” SQL ใช้ค้นหาฐานข้อมูล คุณไม่ได้เลือก SQL แทนที่ ภาษาอื่น — มันอยู่เคียงข้างสิ่งที่คุณเลือกเสมอ ขาวดีคือ AI เก่งมากในการเขียน SQL ดังนั้นคุณสามารถอธิบายข้อมูลที่ต้องการเป็นคำพูดธรรมดา แล้วให้มันสร้างคำสั่ง query ให้ได้

เปรียบเทียบแบบเห็นภาพรวม

ภาษา	เหมาะกับ	ระวิ้งเรื่อง	ความคล่องของ AI
JavaScript/TypeScript	เว็บแอป, full-stack, edge	dependency บวมพอง, เปลี่ยนแปลงเร็ว	ยอดเยี่ยม

ภาษา	เหมาะกับ	ระวังเรื่อง	ความคล่องของ AI
Python	AI/ML, ข้อมูล, สคริปต์, API	ช้าเมื่อคำนวณดิบ, การตั้งค่า environment	ยอดเยี่ยม
Go	backend เร็ว, CLI, บริการ	ยืดยาว, ไม่เหมาะกับ frontend	ดีมาก
Rust	ประสิทธิภาพ, ความปลอดภัย, WASM	เส้นโค้งชัน, วงจรแก้ไขมากขึ้น	ดี (ขรุขระกว่า)
Java/C#	องค์กร, Android, .NET	หนัก, มักเกินความจำเป็นสำหรับคนเดียว	ดีมาก
Swift/Kotlin	แอป iOS / Android แบบดั้งเดิม	ความเสียดทานของมือถือ, AI อ่อนกว่า	ดี
PHP/Ruby	WordPress, Rails, เว็บแอปที่มีอยู่แล้ว	ค่าเริ่มต้นที่ล้าสมัย, ภาระของเก่า	ดีมาก
SQL	คุยกับฐานข้อมูล	อยู่เคียงข้าง ไม่ใช่แทนที่	ยอดเยี่ยม

ต้นทุนที่แท้จริงของทางเลือกแปลกใหม่

มันน่าดึงดูดที่จะเลือกภาษาเพราะมันดูสง่างาม หรือเร็ว หรือเพราะคนที่คุณนับถือยกย่องมัน โดยเฉพาะสำหรับ vibe coding สัญชาตญาณนั้นอาจจมโปรเจกต์ของคุณลงอย่างเงียบ ๆ เหตุผลคือ: ทุกชั่วโมงที่คุณใช้ต่อสู้กับเครื่องมือของภาษาเฉพาะกลุ่ม หรือคลายปมผลลัพธ์ของ AI ที่ผิดพลาดอย่างแนบเนียนเพราะโมเดลเห็นตัวอย่างน้อยเกินไป คือชั่วโมงที่คุณไม่ได้ใช้ส่งมอบงาน ทางเลือกแปลกใหม่แทบไม่เคยล้มเหลวแบบดั่งโครม มันล้มเหลวแบบหยุดนิ่ง ๆ ของวงจรแก้ไขที่เพิ่มขึ้น ไลบรารีที่ขาดหาย และคำตอบใน Stack Overflow ที่ไม่มีอยู่จริง

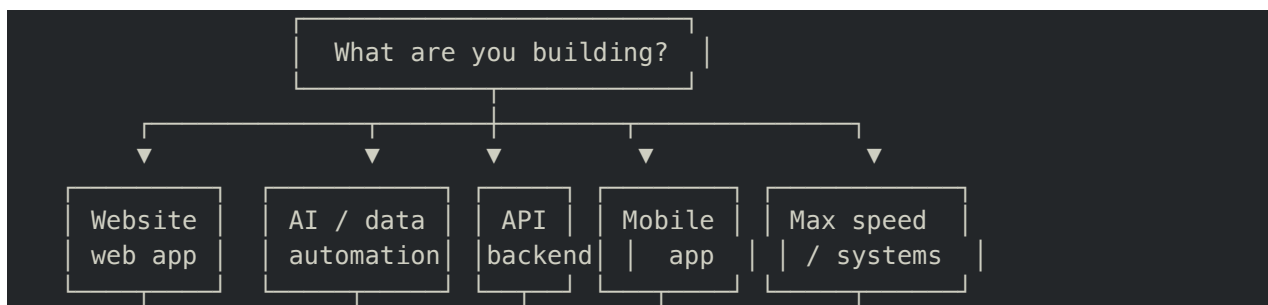
โดยเฉพาะเจาะจง บทลงโทษนี้ปรากฏใน 3 จุด อย่างแรก AI คล่องแคล่วน้อยลง จึงทำผิดพลาดมากขึ้นและแก้ไขได้แยกลง — และคุณเองที่ไม่รู้ภาษานั้น ก็แยกไม่ออกว่าโค้ดไหนดีไหนแย่ อย่างที่สอง ระบบนิเวศบางเบากว่า ไลบรารีที่คุณต้องการอาจไม่มีอยู่ อาจถูกทิ้งร้าง หรือไม่มีตัวอย่างให้ลอกเลียน อย่างที่สาม เมื่อคุณติดขัด เว็บก็เงียบกว่า: บทเรียนน้อยลง คำถามที่ได้รับคำตอบน้อยลง คนที่เคยเจอกำแพงเดียวกับคุณน้อยลง ในขณะที่ Python หรือ TypeScript ทั้งสามข้อนี้แทบไม่ใช่ปัญหาเลย

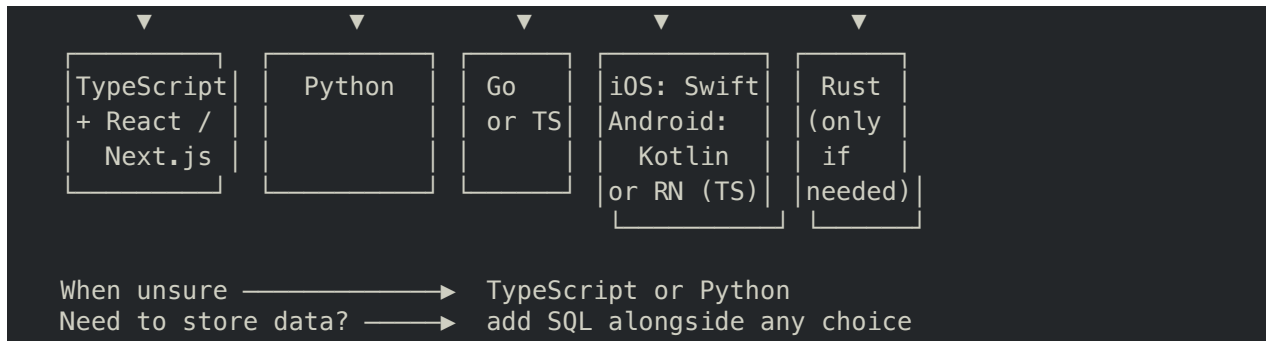
ไม่ได้หมายความว่าภาษาแปลกใหม่นั้นแย่ มันหมายความว่าภาระการพิสูจน์ตกอยู่ที่ทางเลือกแปลกใหม่นั้น ถ้าคุณมีเหตุผลที่เป็นรูปธรรมและชัดเจน — คุณต้องการความเร็วของ Rust จริง ๆ, คุณถูกบังคับให้ใช้ Swift เพราะ App Store, คุณกำลังต่อยอดบริการ Go ที่มีอยู่แล้ว — ก็จ่ายต้นทุนนั้นด้วยตาที่เปิดกว้าง แต่ถ้าเหตุผลคือ “มันดูเทดี” ให้เลือกตัวเลือกยอดนิยมแล้วส่งมอบงานออกไป คุณเขียนใหม่ทีหลังได้เสมอ และแทบจะไม่มีวันได้ทำจริง เพราะตัวเลือกยอดนิยมจะทำหน้าที่ของมันได้สำเร็จอยู่แล้ว

เลือกตามเป้าหมาย

ข้ามความทรมานในการตัดสินใจไปเลย จับคู่เป้าหมายของคุณกับแถวหนึ่ง แล้วเดินหน้าต่อ — คุณเปลี่ยนเส้นทางได้เสมอในภายหลัง และสำหรับโปรเจกต์แรก โมเมนตัมสำคัญกว่าการปรับให้เหมาะสมที่สุด

นี่คือการตัดสินใจทั้งหมดในรูปแบบต้นไม้เดียว เริ่มจากด้านบน ตอบแต่ละคำถาม แล้วตามกิ่งไปยังสแตกของคุณ:





- ฉันอยากสร้างเว็บไซต์หรือเว็บแอป → TypeScript (กับ React/Next.js) ตัวเลือกที่ชัดเจนและรองรับดีเยี่ยม
- ฉันอยากทำ AI ข้อมูล หรือระบบอัตโนมัติ → Python นี่คือที่ที่เครื่องมือต่าง ๆ อาศัยอยู่
- ฉันต้องการ API หรือบริการ backend ที่รวดเร็ว → Go (เรียนง่ายและเร็ว) หรือ TypeScript (ถ้าเป็นส่วนหนึ่งของสแต็กเว็บ) FastAPI ของ Python ก็ใช้ได้เช่นกัน
- ฉันกำลังสร้างแอปมือถือ → Swift สำหรับ iOS, Kotlin สำหรับ Android — หรือพิจารณาแนวทางข้ามแพลตฟอร์มแบบเว็บ (React Native) เพื่ออยู่ในโลกของ TypeScript ต่อไป
- ฉันต้องการความเร็วดิบหรือการควบคุมระดับระบบ → Rust แต่เฉพาะเมื่อคุณยืนยันแล้วว่าต้องการมันจริง ๆ คนส่วนใหญ่ไม่ได้ต้องการ
- ฉันกำลังเพิ่มเติมให้เว็บไซต์ที่มีอยู่แล้ว → ใช้ภาษาเดียวกับที่มันเขียนไว้อยู่แล้ว ปลั๊กอิน WordPress หมายถึง PHP; แอป Rails เก่าหมายถึง Ruby จับคู่กับโค้ดเบสที่มีอยู่ อย่าฝืนมัน
- ฉันแค่อยากเรียนรู้และส่งมอบงานให้เร็ว → Python หรือ TypeScript ทั้งคู่ให้อภัยง่าย ทั้งคู่เป็นภาษาที่ AI รู้จักดีที่สุด และทั้งคู่มีบทเรียนและตัวอย่างไม่รู้จบ
- แอปของฉันต้องเก็บข้อมูล → ไม่ว่าคุณ sẽเลือกอะไรข้างต้น บวกกับ SQL สำหรับฐานข้อมูล คุณไม่ต้องเลือกอย่างใดอย่างหนึ่งระหว่างมัน

บทเรียนแฝง: เมื่อไม่แน่ใจ ให้เลือกตัวเลือกยอดนิยม TypeScript และ Python ครอบคลุมสิ่งที่ vibe coder เดี่ยวส่วนใหญ่จะสร้างขึ้นเกือบทั้งหมด และเป็นภาษาที่ AI ผู้ช่วยของคุณจัดการได้เชื่อถือได้ที่สุด ภาษาที่ “ดีที่สุด” สำหรับ vibe coding มักเป็นภาษาที่ AI รู้จักดีที่สุด — และนั่นหมายถึงภาษาที่ได้รับความนิยม เก็บทางเลือกแปลกใหม่ไว้สำหรับตอนที่คุณมีเหตุผลที่เป็นรูปธรรม และปล่อยให้ AI จัดการเรื่องไวยากรณ์ไม่ว่าจะทางไหนก็ตาม

สรุปและฝึกฝน

สิ่งที่ควรจำ

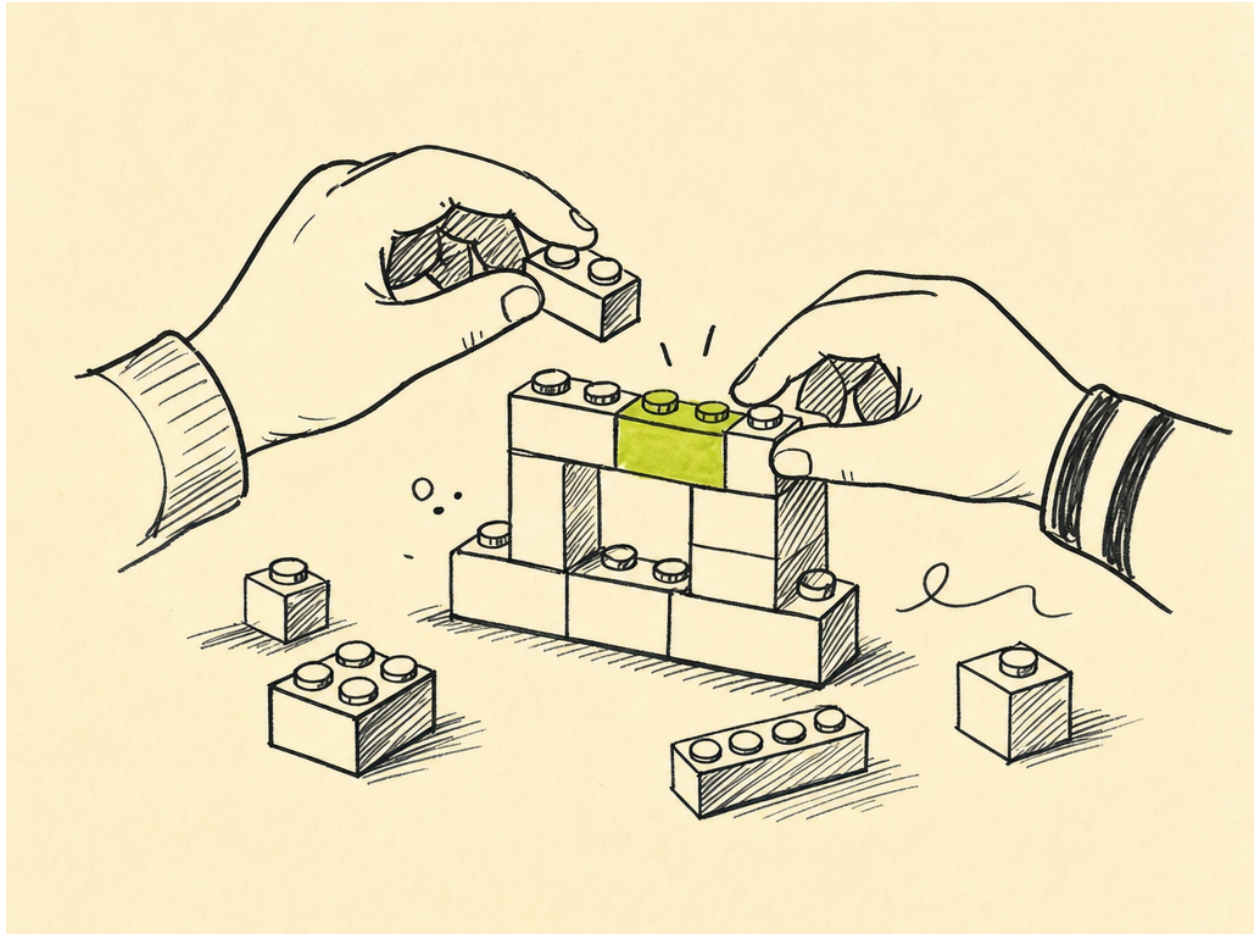
- AI คล่องแคล่วในภาษายอดนิยมมากกว่าภาษาเฉพาะกลุ่มมาก — ความนิยมคือฟีเจอรอย่างหนึ่ง มักเป็นปัจจัยเดียวที่ใหญ่ที่สุดว่าโปรเจกต์จะราบรื่นหรือไม่
- TypeScript และ Python คือตัวเลือกปลอดภัยที่เป็นดีฟอลต์: TypeScript สำหรับเว็บ/full-stack, Python สำหรับ AI/ข้อมูล/ระบบอัตโนมัติ และทั้งคู่มี AI รองรับได้อย่างยอดเยี่ยม
- Go (backend ใบนารีเดียว), Rust (ความเร็ว/ความปลอดภัย แต่จริงแสบมากขึ้น) และภาษาอื่น ๆ ต่างมีจุดที่เหมาะสมแคบ ๆ ของตัวเอง — เอื้อไปหามันเมื่อมีเหตุผลที่เป็นรูปธรรมเท่านั้น
- SQL ไม่ใช่ตัวเลือกแบบเลือกอย่างใดอย่างหนึ่ง มันอยู่เคียงข้างสิ่งที่คุณเลือกทันทีที่แอปของคุณเก็บข้อมูล
- จับคู่ภาษาให้ตรงกับเป้าหมายของคุณ ไม่ใช่กับสิ่งที่ฟังดูน่าประทับใจ — โหมเมนต์สำคัญกว่าการปรับให้เหมาะสมที่สุดในโปรเจกต์แรก

ลองทำดู

หยิบไอเดียโปรเจกต์ที่คุณอยากสร้างจริง ๆ แล้วเขียนหนึ่งประโยคอธิบายเป้าหมายของมัน จับคู่กับแถวหนึ่งใน “เลือกตามเป้าหมาย” จากนั้นขอให้ AI ผู้ช่วยของคุณยืนยันหรือท้าทายการเลือกนั้น ให้มันวางสแต็กดีฟอลต์ (ภาษา, เฟรมเวิร์ก, ที่ที่คุณจะดีฟลอย) เป็นสามข้อ แล้วขอให้มันบอกหนึ่งอย่างที่จะทำให้ภาษาอื่น เหมาะสมกว่า คุณจะจบด้วยการตัดสินใจเลือกสแต็กที่มีเหตุผลรองรับก่อนจะเขียนโค้ดสักบรรทัด

I want to build the following project: [describe it in one or two sentences]. Recommend the best language and framework for vibe coding it, optimizing for how fluently an AI assistant can write and fix the code, not for raw performance. Give me: (1) the language and main framework, (2) where I'd deploy it, (3) one realistic gotcha to watch for, and (4) one scenario in which a different language would actually be the better choice. Keep it short.

สร้างพีเจอร์ ทีละ Vibe



ตอนนี้คุณคุยกับ AI แล้วให้มันเขียนโค้ดได้แล้ว แต่พีเจอร์จริงนั้นใหญ่กว่าพรมมปีเดียว ทักษะที่แยกคนที่ส่งงานได้จริงออกจากคนที่ติดหล่มอยู่ตรงที่ คือการรู้วิธีตัดพีเจอร์ออกเป็นชิ้นเล็ก ๆ แล้วขับเคลื่อน AI ผ่านแต่ละชั้นทีละชั้น พร้อมกับรักษาให้แอปทำงานได้อยู่เสมอในทุกขั้นตอน

บทนี้คือวงจรการสร้าง (build loop) หลัก เรียนรู้มันครั้งเดียวแล้วคุณจะใช้ซ้ำกับทุกพีเจอร์ที่คุณสร้างต่อจากนี้ มันคือความต่างระหว่างโปรเจกต์ที่เติบโตอย่างมั่นคง กับโปรเจกต์ที่พังครืนลงมาด้วยน้ำหนักของตัวเองทันทีที่มันเริ่มซับซ้อนขึ้น

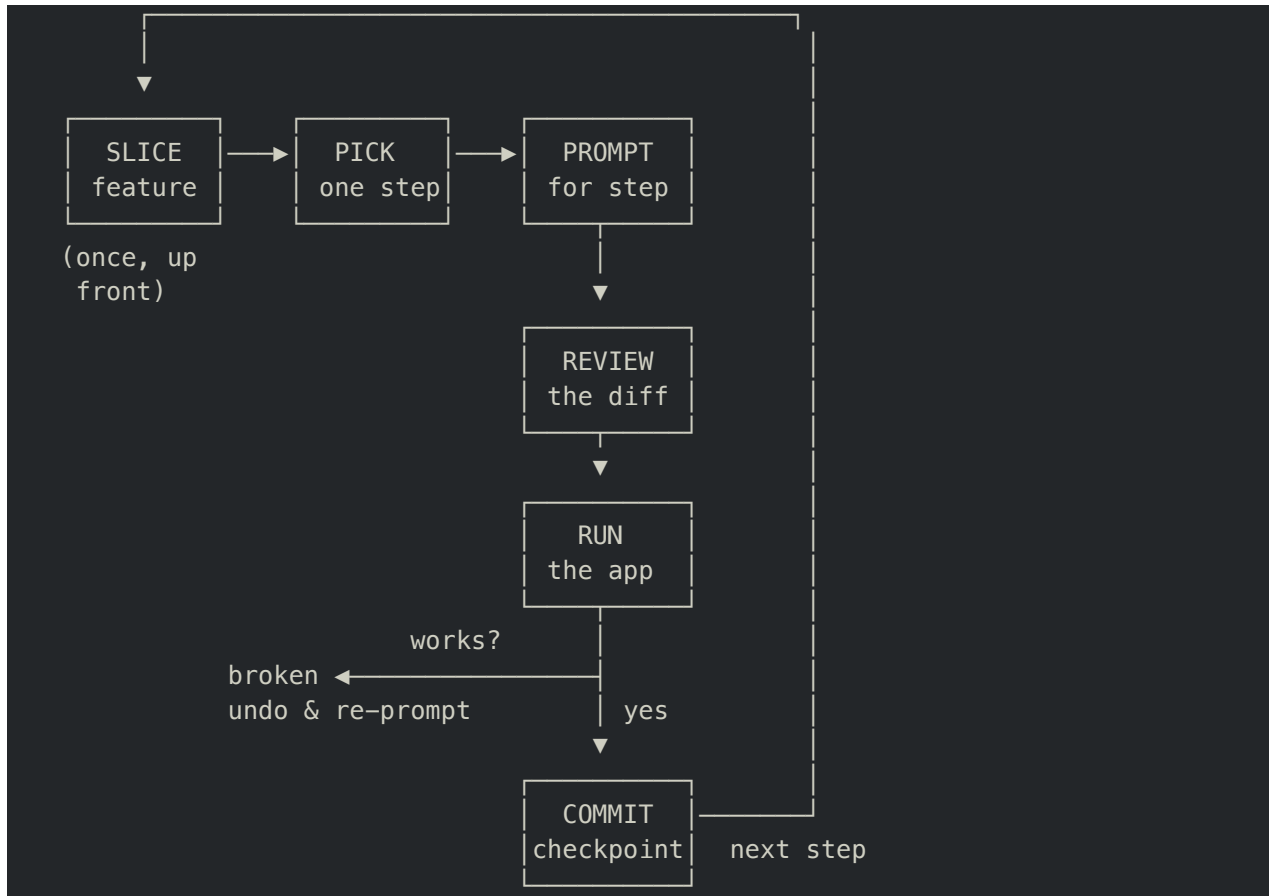
วงจรการสร้าง (The Build Loop)

นี่คือวงจรที่คุณจะวนทำซ้ำแล้วซ้ำอีก มันถูกออกแบบมาให้เล็กโดยตั้งใจ

1. ตัดพีเจอร์ออกเป็นขั้นตอนแนวตั้งเล็ก ๆ (vertical steps) แต่ละขั้นตอนควรเป็นสิ่งที่คุณสร้าง รัน และ เห็นมัน

- ทำงานจริง ได้ภายในไม่กี่นาที
2. เลือกมาหนึ่งขั้นตอน แคหนึ่งเดียว ละเลยอย่างอื่นไปก่อน
3. สังเคราะห์สำหรับขั้นตอนนั้นขั้นตอนเดียว ระบุให้ชัดว่าอะไรเปลี่ยน อะไรต้องคงเดิม
4. ตรวจสอบการเปลี่ยนแปลง อ่านสิ่งที่ AI เขียนออกมา คุณไม่จำเป็นต้องเข้าใจทุกบรรทัด แต่ให้กวาดตาดูว่ามีอะไรที่ผิดเห็นได้ชัดหรือไม่
5. รันแอป คลิกปุ่มจริง ๆ โหลดหน้าเพจจริง ๆ ถึง endpoint จริง ๆ ยืนยันว่าขั้นตอนนี้ใช้งานได้
6. คอมมิต บันทึกจุดตรวจ (checkpoint) ไว้ในระบบควบคุมเวอร์ชันพร้อมข้อความสั้น ๆ
7. ทำซ้ำ จากขั้นตอนที่ 2 ด้วยขั้นตอนถัดไป — จนกว่าพีเจอรันจะเสร็จสมบูรณ์

เมื่อวาดเป็นวงจร มันจะออกมาแบบนี้ — คุณวนไปเรื่อย ๆ รอบละหนึ่งขั้นตอนเล็ก ๆ จนกว่าพีเจอรันจะเสร็จ:



ประเด็นทั้งหมดคือคุณจะไม่วันห่างไกลจากแอปที่ใช้งานได้ ถ้าขั้นตอนที่ 4 ทำให้เกิดอะไรพัง คุณจะมีการเปลี่ยนแปลงเล็ก ๆ แคจุดเดียวที่ต้องย้อนกลับ ไม่ใช่ความยุ่งเหยิงที่พันกันไปหมด

คนที่ข้ามขั้นตอน 4 ถึง 6 จะรู้สึกว่าตัวเองเร็วขึ้นอยู่รายยี่สิบนาที จากนั้นพวกเขาจะเจอกับที่หาต้นตอไม่เจอ เพราะเวอร์ชัน ที่รู้แน่ชัดว่าใช้งานได้ ล่าสุดของแอปย้อนไปไกลนับร้อยการเปลี่ยนแปลง วงจรนี้รู้สึกซ้ำก็เพราะมันไม่เคยปล่อยให้คนสะสมหนี้แบบนั้นได้เลย

“Vertical Slice” คืออะไร

Vertical slice (ชิ้นส่วนแนวตั้ง) คือชิ้นบาง ๆ ของพีเจอรันที่ทะลุผ่านแอปของคุณตลอดทั้งสาย — มี UI นิดหน่อย มีตรรกะ นิดหน่อย มีข้อมูลนิดหน่อย — แทนที่จะสร้างทีละชั้นทั้งชั้น

แนวทางที่น่าดึงดูดแต่ผิด (แนวนอน — horizontal):

- สร้างสคีมารฐานข้อมูลทั้งหมดก่อน

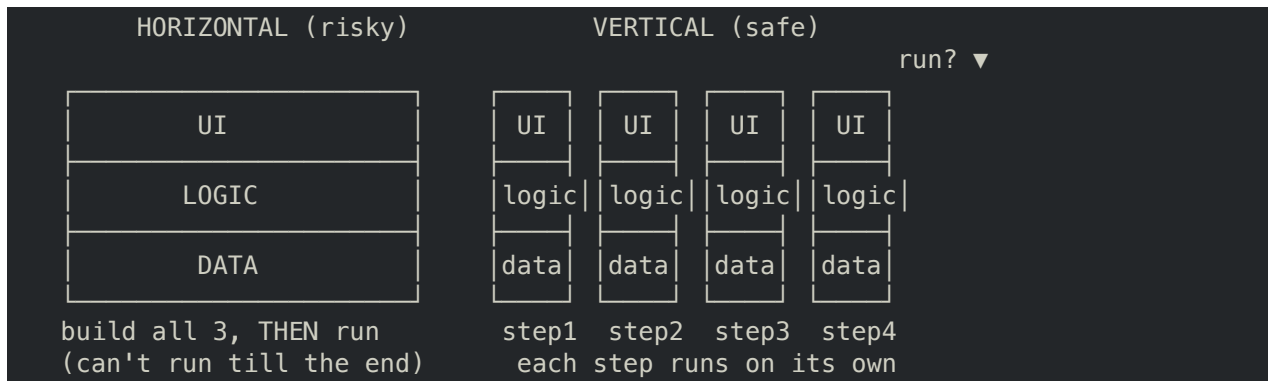
- แล้วสร้างตรรกะฝั่งเซิร์ฟเวอร์ (backend) ทั้งหมด
- แล้วสร้าง UI ทั้งหมด

คุณจ้ะรัน อะไรไม่ได้เลย จนกว่าทั้งสามอย่างจะเสร็จ ดังนั้นคุณจึงจับปัญหาแต่เนิ่น ๆ ไม่ได้

แนวทางที่ดีกว่า (แนวตั้ง — vertical):

- ขั้นตอนที่ 1: ปุ่มที่ทำเวอร์ชันง่ายที่สุดของสิ่งนั้น โดยใช้ข้อมูลปลอม
- ขั้นตอนที่ 2: ให้มันใช้ข้อมูลจริง
- ขั้นตอนที่ 3: จัดการกรณีขอบ (edge case)

ความต่างนี้เห็นได้ชัดที่สุดเมื่อวางเทียบกัน แนวอนสร้างทีละชั้นเต็มขั้นและไม่รันอะไรเลยจนกว่าจะจบ ส่วนแนวตั้งสร้างแถบบาง ๆ ที่รันได้ทะลุทุกชั้นในแต่ละขั้นตอน:



แต่ละ slice รันได้ นั่นแหละคือสิ่งที่ทำให้แอปทำงานได้อยู่เสมอ

ทดสอบง่าย ๆ ว่า slice ดีหรือไม่: คุณเดโมมันได้ไหม? ถ้าคุณเข้าไปที่หน้าจอแล้วพูดว่า “ดูสิ” ได้ นั่นคือ vertical slice ตัวจริง แต่ถ้าเดโมที่ซื่อสัตย์ที่สุดที่ทำได้คือ “เออ สคิมาถูกวางไว้แล้วนะ” แสดงว่าคุณสร้างชิ้นแนวอนขึ้นมา และคุณจะไม่รู้ว่ามันถูกต้องหรือไม่จนกว่าจะถึงทีหลังมาก — มักจะเป็นช่วงเวลาที่ย่ำแย่ที่สุด

ตัวอย่างที่ลงมือทำจริง: ฟีเจอร์ “รายการโปรด”

มาสร้างฟีเจอร์เล็ก ๆ กันตั้งแต่ต้นจนจบ: ผู้ใช้สามารถทำเครื่องหมายรายการในลิสต์ว่าเป็นรายการโปรด และเห็นจำนวนของรายการที่พวกเขาถูกใจแล้ว

การตัดเป็นชิ้น (Slicing)

นี่คือวิธีที่ผมจะตัดมันออกเป็นขั้นตอนแนวตั้ง:

1. เพิ่มปุ่มดาวข้างรายการแต่ละอัน (ยังไม่ทำอะไร แคปรากฏขึ้นมา)
2. คลิกดาวเพื่อสลับรูปลักษณ์เปิด/ปิด (แค่ state เฉย ๆ ยังไม่บันทึก)
3. แสดงตัวนับ “★ 3 favorites” ที่สะท้อนการสลับเหล่านั้น
4. บันทึกรายการโปรดให้คงอยู่แม้โหลดหน้าใหม่

สังเกตว่าแต่ละขั้นตอนรันได้และมองเห็นได้อย่างอิสระ คุณสามารถหยุดหลังจากขั้นตอนใดขั้นตอนหนึ่งก็ได้ แล้วยังมีแอปที่ใช้งานได้อยู่

ขั้นตอนที่ 1: ปุ่ม

พร้อมปีทีจะระบุชื่อไฟล์ การเปลี่ยนแปลง และสิ่งที่ ห้ามแตะ:

In the ItemCard component (src/components/ItemCard.jsx), add a star button in the top-right corner of each card. It should render an

outline star icon. Don't wire up any click behavior yet – just make it appear. Keep the rest of the card layout unchanged.

รันแอป เห็นดาวไหม? ดี คอมมิตเลย:

```
git add -A && git commit -m "Add star button to item cards"
```

ขั้นตอนที่ 2: สลับ state

Make the star button toggle between filled and outline when clicked. Track this with local component state – don't save it anywhere yet. A filled star means favorited.

ลองคิดว่า state คือความจำระยะสั้นของแอป — สิ่งที่เป็นจริง ณ ตอนนี้ เช่น ดาวดวงหนึ่งกำลังถูกเติมสีอยู่หรือไม่ ตอนนี้คุณกำลังจัดการ state และ UI ไปพร้อมกัน ซึ่งเป็นเรื่องปกติ (UI หรือ user interface คือแค่ส่วนของแอปที่คนมองเห็นและคลิกได้) AI น่าจะเพิ่มโค้ดประมาณนี้:

```
const [isFavorite, setIsFavorite] = useState(false);

<button onClick={() => setIsFavorite(!isFavorite)}>
  {isFavorite ? <StarFilled /> : <StarOutline />}
</button>
```

คลิกดาวสักสองสามดวง มันสลับไหม? คอมมิต

ขั้นตอนที่ 3: ตัวนับ

Add a "★ N favorites" counter at the top of the item list. It should count how many items currently have their star filled. Lift the favorite state up to the list component if you need to so the counter can see it.

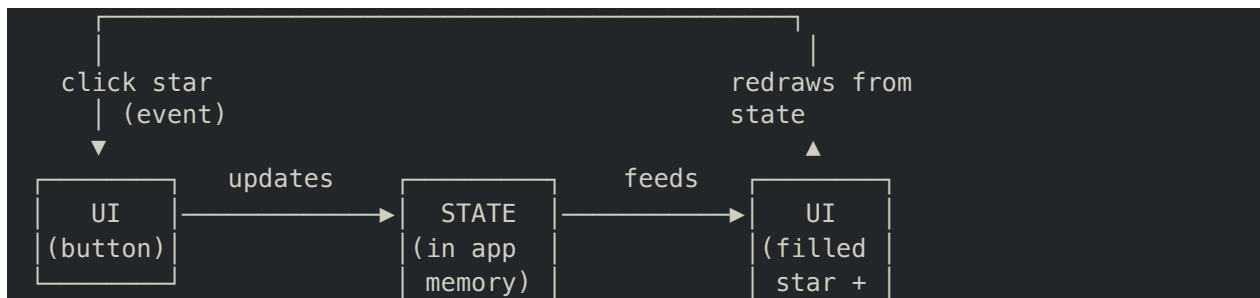
ขั้นตอนนี้อาจต้องให้ AI จัดโครงสร้างใหม่ว่า state อยู่ตรงไหน — นั่นเป็นเรื่องที่คาดไว้แล้ว และนี่แหละคือเหตุผลที่คุณทำมันเป็นขั้นตอนเล็ก ๆ แยกต่างหาก รันมัน สลับดาวสักสองสามดวง ดูตัวเลขเปลี่ยน คอมมิต

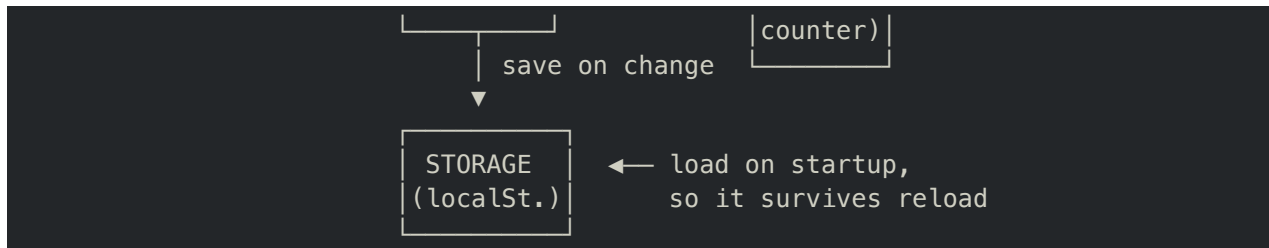
ขั้นตอนที่ 4: ทำให้คงอยู่ (persist)

Persist favorites so they survive a page reload. Use localStorage for now – save the list of favorited item IDs whenever it changes, and load it on startup.

(localStorage เป็นแค่สมุดบันทึกเล็ก ๆ ที่เบราว์เซอร์เก็บไว้บนคอมพิวเตอร์ของผู้เยี่ยมชมเอง เพื่อให้แอปจดข้อมูลไว้แล้วหามันเจออีกครั้งหลังโหลดหน้าใหม่) โหลดหน้าใหม่ ยังเป็นรายการโปรดอยู่ไหม? คุณเพิ่งส่งฟีดแบ็กออกไปแล้วที่ละ slice ที่รันได้จริง

นี่คือวิธีที่ชิ้นส่วนต่าง ๆ ของฟีดแบ็กที่เสร็จสมบูรณ์แล้วสื่อสารกัน การคลิกไหลลงไปสู่ state หน้าจาวาดใหม่จาก state และ state ถูกสะท้อนไปยัง storage เพื่อให้มันอยู่รอดหลังโหลดหน้าใหม่:





การอ่าน Diff โดยไม่ต้องเข้าใจทุกบรรทัด

ขั้นตอนที่ 4 ของวงจรคือ “ตรวจสอบการเปลี่ยนแปลง” และมันคือขั้นตอนที่คนมักจะทำแบบขอไปทีมากที่สุด คุณไม่จำเป็นต้องเขียนโค้ดเองได้เพื่อตรวจสอบมันอย่างมีประสิทธิภาพ คุณแค่ต้องกวาดตาดหาสัญญาณเตือนกลุ่มเล็ก ๆ

เมื่อ AI ส่งการเปลี่ยนแปลงมาให้คุณ วิธีที่สะอาดที่สุดในการดูมันคือ diff — บรรทัดที่ถูกเพิ่มและถูกลบออกจริง ๆ:

`git diff`

บรรทัดที่ขึ้นต้นด้วย + คือสิ่งที่ถูกเพิ่ม บรรทัดที่มี - คือสิ่งที่ถูกลบ อ่านมันแบบผู้ตรวจสอบ ไม่ใช่แบบคอมพิวเตอร์ คุณกำลังมองหาสิ่งเหล่านี้:

- มันแตะไฟล์ที่ไม่มีเหตุผลอะไรจะไปแตะหรือเปล่า? คุณขอให้ทำตัวนับ ทำไม `auth.js` ถึงอยู่ใน diff ด้วย?
- มันลบอะไรที่สำคัญหรือไม่? บรรทัดที่ถูกลบ (-) ที่คุณจำไม่ได้ว่าคืออะไร ควรค่าแก่การถามคำถาม
- มันเปลี่ยนพฤติกรรมเจียบ ๆ โดยที่คุณไม่ได้ขอหรือไม่? ค่าเริ่มต้นใหม่ เงื่อนไขที่เปลี่ยนไป ของ “แถม” ที่ “ช่วยเหลือ”
- รูปร่างของมันสอดคล้องกับสิ่งที่ขอไหม? ค่าขอประโยคเดียวที่ผลิตโค้ดออกมาสองร้อยบรรทัด สมควรได้รับการมองซ้ำอีกครั้ง

ถ้ามีอะไรดูแปลก ๆ คุณไม่จำเป็นต้องแก้ไขมันเอง แค่ถามว่า: “ทำไมคุณถึงเปลี่ยน `auth.js`? ฉันขอแค่ตัวนับเท่านั้น” ครั้งหนึ่งของเวลา AI จะตอบว่า “จับได้ดี” แล้วย้อนกลับให้ นิสัยเดียวนี้จับบักได้มากกว่าการจ้องมองโค้ดที่คุณไม่เข้าใจทั้งหมดเสียอีก

เมื่อไหร่ควรปล่อยให้ AI รันเอง กับเมื่อไหร่ควรคุมพวงมาลัยเอง

ไม่ใช่ทุกขั้นตอนที่ต้องการการควบคุมระดับเดียวกัน

ปล่อยให้ AI รันได้อย่างอิสระเมื่อ:

- ขั้นตอนนั้นเล็กและนิยามไว้ชัดเจน (“เพิ่มปุ่ม”)
- มันเป็นแพทเทิร์นทั่วไปที่ AI เห็นมาแล้วนับพันครั้งอย่างชัดเจน
- ความผิดพลาดมีต้นทุนต่ำและเห็นได้ชัด (คุณจะได้เห็นมันทันทีที่รันแอป)

คุมพวงมาลัยเอง — พรอมปีเล็กลง ตรวจสอบละเอียดขึ้น — เมื่อ:

- การเปลี่ยนแปลงจะต้องข้อมูลที่กู้คืนได้ยาก (ลบเรคคอร์ด, ทำ migration)
- AI ยังคง “แก้” สิ่งเดิมวนไปวนมา
- พฤติกรรมนั้นละเอียดอ่อนและจะไม่ปรากฏให้เห็นแค่จากการมองเห็นหน้าจอ (การคำนวณเงิน สิทธิการเข้าถึง การยืนยันตัวตน)
- คุณลองมาแล้วสองครั้งและมันแย่ง ไม่ดีขึ้น

กฎที่เชื่อถือได้: ยิ่งมีความเสี่ยงกว้างเท่าไร ขั้นตอนก็ควรเล็กลงเท่านั้น

รักษาให้แอปทำงานได้อยู่เสมอ

นี่คือวินัยที่ทำให้ทุกอย่างอื่นเป็นไปได้:

- รันหลังทุกขั้นตอน ไม่ใช่ทุกสามขั้นตอน ทุกขั้นตอนเดียว
- คอมมิตเฉพาะสถานะที่ใช้งานได้นั้น คอมมิตคือจุดบันทึก (save point) ที่คุณกลับไปได้ ถ้าขั้นตอนหนึ่งทำให้บางอย่างพังจนคุณแก้ไขไม่ได้ภายในหนึ่งหรือสองนาที ให้ทิ้งมันไป (git restore . หรือย้อนกลับ) แล้วสั่งพรอมป์ใหม่ให้ชัดเจนขึ้น — อย่าเอาการแก้ไขไปกองทับบนฐานที่พังอยู่แล้ว
- หนึ่งเรื่องต่อหนึ่งขั้นตอน ถ้าคุณอยากจะ “รีบแก์เรื่องอื่นด้วยเลย” อดใจไว้ก่อน ทำ slice ปัจจุบันให้เสร็จ คอมมิตแล้วค่อยเริ่มขั้นตอนใหม่
- อธิบายบ๊ัก อย่าวินิจฉัยมันเอง เมื่อมีอะไรพัง บอก AI ตรง ๆ ว่าคุณเห็นอะไร: “หลังจากคลิกดาวดวงที่สอง ตัวนับแสดงเป็น 1 ไม่ใช่ 2” ให้มันหาสาเหตุเอง

คอมมิตเป็นขั้นเล็ก ๆ

คอมมิตเล็ก ๆ คือดาข่ายนิรภัยของคุณ แต่ละอันควรเป็นขั้นตอนที่มีความสอดคล้องกันเพียงหนึ่งเดียว:

```
git commit -m "Add star button to item cards"
git commit -m "Toggle favorite state on star click"
git commit -m "Add favorites counter to list header"
git commit -m "Persist favorites in localStorage"
```

ถ้าขั้นตอนที่ 4 พังไป สามขั้นตอนแรกของคุณยังปลอดภัยอยู่ และคุณแทบไม่เสียอะไรเลย คอมมิตที่ใหญ่และนาน ๆ ครั้งคือสาเหตุที่ทำให้คนกลัวที่จะแตะโค้ดของตัวเอง

เมื่อ Slice ใหญ่เกินไป

บางครั้งคุณเลือก “หนึ่งขั้นตอน” แล้วมันก็ยังผิดพลาดอยู่ดี — AI เขียนแอปครึ่งหนึ่งใหม่ทั้งหมด diff อ่านไม่รู้เรื่อง และการรันมันเผยบั๊กใหม่สามตัวพร้อมกัน นั่นแทบจะไม่ใช่โชคร้ายเลย มันเป็นสัญญาณว่า slice นั้นใหญ่เกินไป

วิธีแก้คือตัดให้บางลง ถ้า “เพิ่มรายการโปรดพร้อมการคงอยู่” ระเบิด นั่นไม่ใช่ slice เดียว มันคือฟีเจอร์ทั้งหมดที่ปloom ตัวเป็นพรอมป์เดียว ย้อนกลับไปทีคอมมิตดีล่าสุดของคุณแล้วแบ่งมันใหม่:

```
# Don't ask for this in one shot:
"Add a favorites feature with a counter and localStorage persistence."

# Ask for this instead, one commit each:
"Add a star button to each item card. No behavior yet."
"Make the star toggle filled/outline with local state."
"Add a counter that reflects the filled stars."
"Persist the favorited IDs to localStorage and load them on startup."
```

กฎง่าย ๆ ที่ใช้ได้ดี: ถ้าคุณอธิบายไม่ได้ว่าขั้นตอนหนึ่งทำอะไรด้วยประโยคสั้น ๆ เดียว แสดงว่ามันคือสองขั้นตอน และถ้า diff ของขั้นตอนหนึ่งยาวเกินกว่าจะกวาดดูได้ในหนึ่งนาที ขั้นตอนถัดไปควรเล็กลง การตัด slice เป็นทักษะที่คุณจะเก่งขึ้นเรื่อย ๆ จากการรู้สึกได้ว่า slice ก่อนหน้าหนาเกินไปตรงไหน

สรุปประเด็นสำคัญ

ฟีเจอร์หนึ่งไม่ใช่พรอมป์ยักยัดเดียว — มันคือกองของ slice เล็ก ๆ ที่รันได้ ตัดมันเป็นชิ้น พรอมป์ทีละขั้นตอน ตรวจสอบรัน คอมมิต ทำซ้ำ ปล่อยให้ AI รันเองในเรื่องง่าย ๆ และคุมพวงมาลัยแน่น ๆ ใกล้เคียงอะไรก็ตามที่มีความเสี่ยง รักษาให้แอปทำงานได้อยู่เสมอในทุกจุดตรวจ แล้วคุณก็จะห่างจากพื้นที่มั่นคงแค่การกด undo เล็ก ๆ เดียวเสมอ

ในบทถัดไป เราจะดูว่าต้องทำอะไรเมื่อขั้นตอนหนึ่งไม่เป็นไปตามแผน — การดีบั๊กด้วย vibe

สรุปและฝึกปฏิบัติ

ประเด็นสำคัญ

- ฟีเจอร์คือกองของ vertical slice เล็ก ๆ ไม่ใช่พรมป์ยักษ์เดียว — สร้าง รัน และ เห็น แต่ละ slice ทำงานก่อนจะไปต่อ
- รันวงจรการสร้างทุกครั้ง: ตัดเป็นชั้น → เลือกหนึ่งชั้นตอน → พรมป์ → ตรวจสอบ diff → รันแอป → คอมมิต → ทำซ้ำ
- ตรวจสอบ diff แบบผู้ตรวจสอบ ไม่ใช่แบบคอมไพเลอร์ — จับตาไฟล์ที่ไม่ควรถูกแตะ การเปลี่ยนพฤติกรรมแบบเจียบ ๆ และ diff ที่ใหญ่กว่าค่าขอมาก
- ปลอ่ยให้ AI รันเองในชั้นตอนที่เล็ก ต้นทุนต่ำ และพบเจอบ่อย คมพวงมาลัยเอง (พรมป์เล็กลง ตรวจสอบละเอียดขึ้น) ในจุดที่รัศมีความเสียหายกว้าง
- คอมมิตเฉพาะสถานะที่ใช้งานได้เป็นชั้นเล็ก ๆ — ถ้า slice หนึ่งพังไป ให้ทิ้งมันแล้วส่งพรมป์ใหม่ แทนที่จะเอาการแก้ไขไปกองทับบนฐานที่พังอยู่แล้ว

ลองทำดู

เลือกฟีเจอร์เล็ก ๆ หนึ่งอันที่คุณต้องการในโปรเจกต์ แล้วตัดมันออกเป็นขั้นตอนแนวตั้ง 3-5 ชั้น เขียนแต่ละชั้นเป็นประโยคสั้น ๆ เดียว สร้าง เฉพาะขั้นตอนแรกเท่านั้น: พรมป์สำหรับมัน รัน git diff และกวดดูการเปลี่ยนแปลง รันแอปเพื่อยืนยันว่ามันทำงานได้ แล้วคอมมิตด้วยข้อความบรรทัดเดียว หยุดตรงนั้นไปก่อน เป้าหมายคือให้คุณรู้สึกถึงการวนวงจรที่สะอาดหนึ่งรอบ — หนึ่ง slice ที่รันได้ หนึ่ง diff ที่คุณอ่านจริง ๆ หนึ่งคอมมิตที่คุณกลับไปได้

```
I want to add this feature: [describe the feature in one or two sentences].  
Don't write any code yet. First, slice it into 3 to 5 vertical steps, where  
each step is something I can build, run, and SEE working on its own. Order  
them so the app keeps working after every step. For each step give me a  
one-sentence description and name the first file you'd change. Then wait — I'll  
ask you to build them one at a time.
```


การดีบั๊กด้วย AI



โปรแกรมทุกตัวที่คุณสร้างจะพังในบางจุด นั่นไม่ใช่สัญญาณว่าคุณทำผิด — มันคือสภาวะปกติของซอฟต์แวร์ บั๊ก (bug) ไม่ใช่ความล้มเหลวทางทักษะ แต่เป็นสิ่งที่เกิดขึ้นเมื่อโค้ดจริงเจอกับข้อมูลจริง ผู้ใช้จริง และกรณีขอบ (edge case) ที่ไม่มีใครคาดคิด ความแตกต่างระหว่างปลายวันหนึ่งที่น่าหงุดหงิดกับการแก้ปัญหาใน 5 นาทีคือ วิธีการ เมื่อคุณดีบั๊กแบบ vibe coding คุณไม่ใช่คนที่ต้องจ้องมอง stack trace (รายงานข้อผิดพลาดที่โปรแกรมพิมพ์ออกมาเมื่อมันล่ม ซึ่งเราจะอธิบายรายละเอียดทีหลัง) จนกว่ามันจะเข้าใจได้ คุณคือผู้กำกับ: คุณทำให้ปัญหาเกิดซ้ำ รวบรวมหลักฐานที่ถูกต้อง แล้วส่งมอบเคสที่กระชับและมีกรอบชัดเจนให้ AI เพื่อให้มันรับภาระงานหนักไป บทนี้จะให้กระบวนการที่ทำซ้ำได้สำหรับสิ่งนั้นโดยเฉพาะ — กระบวนการที่ใช้ได้ไม่ว่าบั๊กจะเป็นแค่การพิมพ์ผิดบรรทัดเดียว หรือเป็น race condition ที่พันกันยุ่งเหยิงอยู่ลึกสามชั้น (race condition คือบั๊กเรื่องจังหวะเวลา: สองสิ่งเกิดขึ้นในลำดับที่คุณไม่คาดคิด เหมือนสองคนแย่งคว่าที่นั่งสุดท้ายพร้อมกัน)

แนวคิดหลัก

ความผิดพลาดที่ใหญ่ที่สุดที่มีใหม่มักทำคือการวางข้อความ “มันใช้ไม่ได้ ช่วยแก้ที” ลงในแชทแล้วหวังผล AI มองไม่เห็นหน้าจอ ข้อมูลของคุณ หรือสิ่งที่คุณคาดหวังว่าจะเกิดขึ้น มันรู้แค่สิ่งที่คุณบอกมันเท่านั้น เมื่อคุณไม่ให้ข้อมูลอะไรเลย มันก็ทำได้แค่สิ่งเดียว: มันเดา — และการเดาที่มันใจแต่ผิด อันตรายยิ่งกว่าการไม่มีคำตอบเลย เพราะคุณจะเชื่อและลงมือทำตามมัน การดีบั๊กด้วย AI ส่วนใหญ่จึงเป็นเรื่องของการป้อนบริบทที่ถูกต้อง ให้มัน เพื่อให้มันสามารถให้เหตุผลได้แทนที่จะเดา

รายงานบั๊กที่ดี — ไม่ว่าจะให้มนุษย์อ่านหรือ AI อ่าน — จะตอบคำถามสามข้อนี้:

- คุณทำอะไรไป? (ขั้นตอนที่ทำให้เกิดปัญหา)

- คุณคาดหวังว่าจะเกิดอะไรขึ้น?
- จริง ๆ แล้วเกิดอะไรขึ้น? (ข้อผิดพลาด ผลลัพธ์ที่ผิด หรือการล่ม)

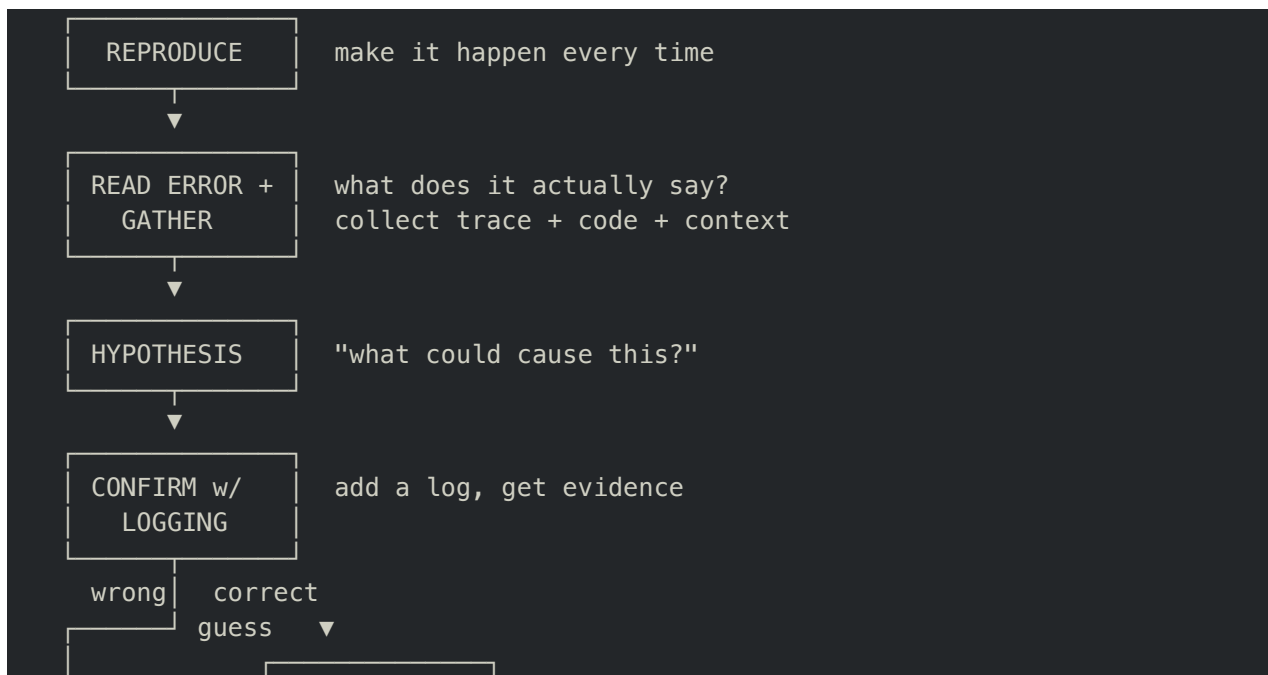
ถ้าคุณตอบสามข้อนี้ได้ คุณก็น่าจะคนส่วนใหญ่ไปแล้ว เหตุผลที่สามข้อนี้สำคัญก็เพราะบั๊ก โดยนิยามแล้ว คือ ช่องว่างระหว่างสิ่งที่คุณคาดหวังกับสิ่งที่เกิดขึ้นจริง AI ไม่สามารถหาช่องว่างที่มันมองไม่เห็นทั้งสองด้านได้ “มันพัง” อธิบายแค่ด้านเดียว ส่วน “ฉันคาดหวังหน้าจอตอนรับ แต่ได้หน้าจอว่างเปล่าพร้อมข้อผิดพลาดนี้” อธิบายทั้งสองด้าน — และช่องว่างก็แทบจะบอกตัวมันเองอยู่แล้ว

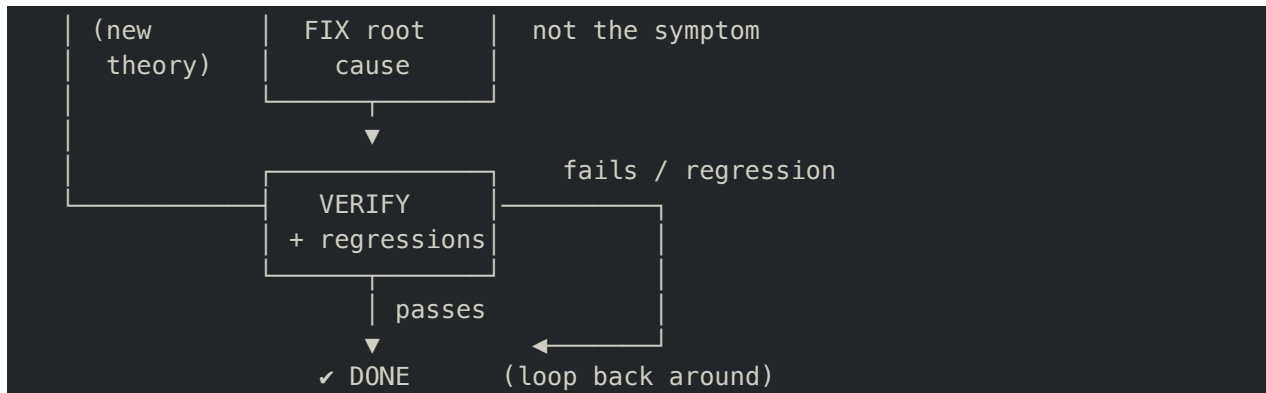
เช็กลิสต์การดีบั๊ก

ทำตามลำดับนี้ทีละข้อ การกระโดดตรงไป “แก้เลย” คือสิ่งที่ทำให้ AI วนเวียนไม่จบ

1. ทำให้ปัญหาเกิดซ้ำได้อย่างแน่นอน หาขั้นตอนที่แน่ชัดซึ่งทำให้บั๊กเกิดขึ้นทุกครั้ง บั๊กที่คุณทำให้เกิดซ้ำไม่ได้ก็คือบั๊กที่คุณยืนยันไม่ได้ว่าแก้แล้วจริงหรือไม่ — คุณจะได้แต่เดาว่ามันหายไปหรือเปล่า ถ้ามันเกิดขึ้นเป็นครั้งคราวให้ตามหาตัวแปรที่เป็นตัวสวิตช์: อินพุตเฉพาะเจาะจง สถานะล็อกอินหรือไม่ล็อกอิน หรือเครือข่ายที่ช้า
2. อ่านข้อผิดพลาดจริง ๆ อย่าอ่านผ่าน ๆ ข้อความแสดงข้อผิดพลาดและส่วน บนสุด ของ stack trace มักจะระบุไฟล์และบรรทัดที่เกิดปัญหา มือใหม่มักเลื่อนผ่านประโยคที่มีประโยชน์ที่สุดบนหน้าจอไปเพราะมันดูน่ากลัว
3. รวบรวมบริบท เก็บข้อความข้อผิดพลาด stack trace โค้ดที่เกี่ยวข้อง และคำอธิบายสั้น ๆ ว่าคุณคาดหวังอะไร
4. ส่งภาพรวมทั้งหมดให้ AI วางทั้งหมดข้างต้นในข้อความเดียว ให้มันตั้งสมมติฐาน
5. ตั้งสมมติฐาน ไม่ใช่ทางแก้ ถามว่า “อะไรที่อาจทำให้เกิดสิ่งนี้ได้?” ก่อนจะถามว่า “แก้ตรงนี้ที” ทางแก้ที่ผิดจากทฤษฎีที่เสียเวลาเปล่า — และบางครั้งยังเพิ่มบั๊ก ตัวที่สอง ทับซ้อนกับตัวแรกอีกด้วย
6. เพิ่ม logging เพื่อทดสอบสมมติฐาน ยืนยันว่าโค้ดกำลังทำอะไรอยู่จริง ๆ ก่อนที่จะเปลี่ยนแปลงมัน
7. ค้นหาแบบทวิภาค (binary search) หาจุดปัญหา จำกัดขอบเขตว่า ตรงไหน ที่พัง โดยตัดพื้นที่ต้องสงสัยลงครึ่งหนึ่งซ้ำ ๆ
8. แก้ที่สาเหตุที่แท้จริง ไม่ใช่แค่อาการ ให้แน่ใจว่าทางแก้จัดการกับ สาเหตุ ที่มันพัง ไม่ใช่แค่ปัญหาที่มองเห็น
9. ตรวจสอบทางแก้และเช็คว่ามีผลกระทบย้อนกลับหรือไม่ รันขั้นตอนที่ทำให้เกิดปัญหาซ้ำอีกครั้ง แล้วตรวจสอบว่าพีเจอร์ใกล้เคียงยังทำงานได้ — regression คือเมื่อการแก้ปัญหาแอบทำให้สิ่งที่เคยใช้ได้พังไปเจียบ ๆ เหมือนการซ่อมรอยรั่วจุดหนึ่งแล้วทำให้ท่อข้าง ๆ แตก

เช็กลิสต์นั้นแท้จริงแล้วคือลูปเดียวที่คุณวนซ้ำจนกว่าบั๊กจะหายไป แต่ละขั้นตอนส่งหลักฐานต่อไปให้ขั้นถัดไป และการตรวจสอบที่ล้มเหลวก็แค่ส่งคุณกลับไปวนอีกครั้งพร้อมกับสิ่งที่คุณเรียนรู้มา:





ลำดับนี้ไม่ได้สุ่มมาเฉย ๆ แต่ละขั้นตอนสร้างหลักฐานที่ขั้นตอนถัดไปต้องการ การทำให้เกิดซ้ำให้สิ่งที่คุณอ่านได้ การอ่านให้บริบทที่ต้องรวบรวม บริบทให้สิ่งที่ AI ใช้ตั้งสมมติฐานได้ ข้ามขั้นตอนใดไปก็เท่ากับป้อนช่องโหว่ให้ขั้นตอนถัดไป

การป้อนบริบทที่ถูกต้องให้ AI

ตรงนี้เป็นจุดที่คุณมีอำนาจต่อรองมากที่สุด เมื่อรายงานบั๊ก ให้ใส่:

- ข้อความข้อผิดพลาด แบบคำต่อคำ คัดลอกมาให้เป๊ะ — อย่าถอดความเอง การถอดความข้อผิดพลาดก็เหมือนกับ การบรรยายภาพถ่ายผ่านโทรศัพท์: คุณจะพลาดรายละเอียดสำคัญที่สุดไปโดยไม่รู้ตัว
- Stack trace โดยเฉพาะไม่ก็บรรทัดบนสุด
- โค้ดที่เกี่ยวข้อง — ฟังก์ชันที่ทำให้เกิดข้อผิดพลาด บวกกับทุกอย่างที่มันเรียกใช้
- สิ่งที่คุณคาดหวัง เทียบกับสิ่งที่เกิดขึ้นจริง

นี่คือตัวอย่างของพรมต์การดีบั๊กที่ดี:

I'm getting an error when I submit the signup form. Here's the full output:

```

TypeError: Cannot read properties of undefined (reading 'email')
    at validateUser (src/auth/validate.js:14:23)
    at handleSignup (src/routes/signup.js:31:10)
    at processTicksAndRejections (node:internal/process/task_queues:95:5)
  
```

Steps to reproduce: fill in the form, click "Sign up".

Expected: a "Welcome" message.

Actual: the page goes blank and the error above appears in the console.

Here is validate.js (lines around 14):

[paste the function]

And here is the handleSignup function that calls it:

[paste the function]

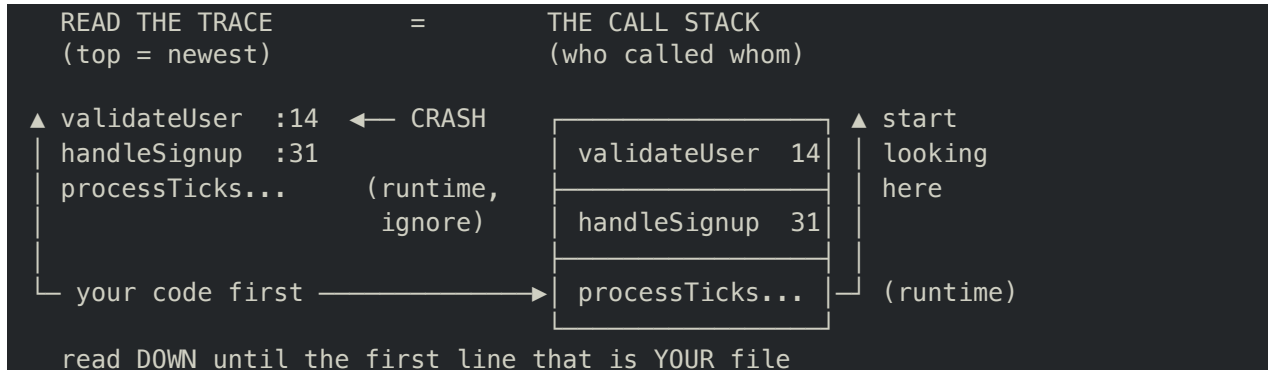
What could be causing this, and how do I confirm it before changing anything?

สังเกตบรรทัดสุดท้าย คุณกำลังขอ สมมติฐานและขั้นตอนยืนยัน ไม่ใช่การแก้ไขทันที นั่นทำให้ AI ยังคงยึดติดกับหลักฐาน

การรู้วิธี อ่าน trace ที่คุณกำลังวางลงไปนั้นช่วยได้มาก stack trace คือประวัติการเรียกฟังก์ชัน โดยเรียงจากล่าสุดไปเก่าสุด: บรรทัดบนสุดคือจุดที่โปรแกรมระเบิดจริง ๆ และแต่ละบรรทัดด้านล่างคือฟังก์ชันที่เรียกบรรทัดด้านบนมันขึ้นมา ดังนั้นในตัวอย่างนี้ validateUser คือจุดที่ email ถูกอ่านจากสิ่งที่เป็น undefined และ handleSignup คือผู้

ที่เรียก validateUser ส่วนเฟรมด้านล่างโค้ดของคุณเอง — อย่าง processTicksAndRejections — เป็นของรันไทม์ของภาษา คุณมักจะข้ามมันไปได้เลย อ่านจากบนลงล่างจนกว่าจะเจอบรรทัดแรกที่เข้าไปที่ ไฟล์ของคุณเอง และตรงนั้นมักจะเป็นจุดที่ควรเริ่มมองหา

trace เดียวกัน เมื่อมองเป็นกองของการเรียกฟังก์ชัน จะทำให้ลำดับชัดเจนขึ้น การล่มอยู่บนสุด และแต่ละเฟรมด้านล่างคือผู้เรียกของเฟรมด้านบน:



การอ่านข้อผิดพลาดที่คุณไม่เข้าใจ

ไม่ช้าก็เร็วคุณจะเจอข้อผิดพลาดที่ดูเหมือนตัวอักษรปนกันมั่ว ๆ — ชื่อ type ยาว ๆ, stack ที่เต็มไปด้วยสิ่งภายในของเฟรมเวิร์ก, ข้อความที่เป็นศัพท์เฉพาะที่คุณไม่เคยเห็นมาก่อน อย่าให้มันทำให้คุณติดขัด ข้อผิดพลาดคือข้อมูล ไม่ใช่คำตัดสินความสามารถของคุณ เมื่อข้อความนั้นเข้าใจยาก ให้วางมันลงไปแล้วขอให้ AI แปลก่อนที่จะแก้:

```
I don't understand this error. Before suggesting any fix, explain in plain language: what is it actually saying, which line is it pointing at, and what kinds of things typically cause this exact error?
```

```
[paste the full error and stack trace]
```

การขอความหมายในภาษาที่เข้าใจง่ายก่อนนั้นทำให้เกิดผลสองอย่าง มันสอนให้คุณรู้จักรูปแบบ ดังนั้นครั้งต่อไปที่คุณเจอข้อผิดพลาดแบบนี้ คุณจะจำมันได้เอง — และในอีกไม่กี่เดือนนี่คือวิธีที่คุณจะเรียนรู้การอ่าน stack ของตัวเองได้จริง ๆ และมันบังคับให้ AI วางฐานของทางกลับสิ่งที่ข้อผิดพลาด บอก จริง ๆ แทนที่จะกระโดดไปหาการแก้แบบทั่วไป ระวังกับดักข้อผิดพลาดที่พบบ่อยที่สุด: ข้อผิดพลาดที่เข้าไปที่บรรทัด 14 มักจะ เกิดจาก ที่อื่นทั้งหมด บรรทัดใน trace คือจุดที่โปรแกรมสังเกตเห็นปัญหา ไม่ใช่จุดที่ปัญหาเริ่มเสมอไป

สมมติฐาน การทำ logging และการค้นหาแบบทวิภาค

เมื่อ AI เสนอสาเหตุมาแล้ว อย่าเพิ่งยอมรับทันที ทดสอบมันแบบประหยัด วิธีทดสอบที่ประหยัดที่สุดมักจะเป็นบรรทัด log ขอให้ AI เพิ่ม logging ชั่วคราว:

```
“ก่อนที่จะเราจะเปลี่ยนอะไร ให้เพิ่ม console.log (หรือ print) ที่แสดงค่าของ user ก่อนบรรทัด 14 เพื่อดูว่ามันเป็น undefined จริงหรือไม่”
```

รันมัน แล้ววางผลลัพธ์กลับมา ตอนนี้คุณมี หลักฐาน ไม่ใช่ทฤษฎี บางทีอาจเป็นว่า user เป็น undefined จริง — ยืนยันสมมติฐาน หรือบางทีมันอาจปกติดี และตัวการที่แท้จริงอยู่ในระดับที่สูงกว่าขึ้นไป ไม่ว่าจะทางไหนคุณก็ก้าวหน้าขึ้น เพิ่ม log อย่างเต็มที่ระหว่างที่คุณสืบสวน: ค่าของตัวแปร และ ป้ายกำกับที่บอกว่ามันคือตัวไหน และ log อยู่ตรงไหน console.log(user) เปล่า ๆ ที่พิมพ์ undefined ออกมามีประโยชน์น้อยกว่า console.log("user at validate.js:14 =", user) มาก และอย่าลืมถึง log ชั่วคราวออกเมื่อมันถูกแก้แล้ว — log ดีักที่หลงเหลืออยู่ก็เป็นความยุ่งเหยิงเล็ก ๆ ของมันเอง

เมื่อบั๊กอยู่ที่ไหนสักแห่งในกระบวนการที่ยาวและคุณบอกไม่ได้ว่าอยู่ตรงไหน ให้ใช้ การค้นหาแบบทวิภาค (binary search) ตัดพื้นที่ที่ต้องสงสัยลงครึ่งหนึ่ง: เพิ่ม log ตรงจุดกึ่งกลาง

แนวคิดคือ: การตรวจสอบแต่ละครั้งทั้งครึ่งหนึ่งของสิ่งที่เหลืออยู่ไป ดังนั้นพื้นที่ค้นหาขนาดใหญ่จะยุบตัวลงอย่างรวดเร็ว ในที่นี้บั๊กซ่อนอยู่ที่ไหนสักแห่งระหว่างขั้นตอนที่ 1 ถึง 8 — สามครั้งของการตรวจสอบก็จับมันได้:

```
step:  1  2  3  4  5  6  7  8      a bug is somewhere in here
      [XXXXXXXXXXXXXXXXXXXXX]

check 1 — log at 4 — OK →  bug is AFTER 4, drop 1-4
                        [XXXXX] (5 6 7 8 left)

check 2 — log at 6 — BAD →  bug is AT/BEFORE 6, drop 7-8
                        [XXXX]  (5 6 left)

check 3 — log at 5 — OK →  bug is at step 6    ✓ found

8 steps → 3 checks.  1000 steps → about 10 checks.
```

ค่านั้นถูกต้องตรงจุดนั้นหรือไม่? ถ้าใช่ บั๊กก็อยู่ หลัง จุดนั้น — ค้นหาในครึ่งนั้น ถ้าผิดไปแล้ว บั๊กก็อยู่ ก่อนหน้า — ค้นหาในครึ่งนั้นแทน แต่ละขั้นตอนตัดพื้นที่ค้นหาลงครึ่งหนึ่ง ดังนั้นแม้แต่เส้นทางที่ยาวเป็นพันบรรทัดก็จะแคบลงเหลือแค่ประมาณสิบครั้งของการตรวจสอบ บอก AI อย่างชัดเจนว่า “ช่วยฉัน bisect ปัญหานี้หน่อย — จุดกึ่งกลางที่ควร log อยู่ตรงไหน?” เทคนิคเดียวกันนี้ใช้ได้กับประวัติด้วย ไม่ใช่แค่กับโค้ด: ถ้าพีเจอร์หนึ่งเคยทำงานได้เมื่อสัปดาห์ที่แล้วแต่ทั้งวันนี้ ให้ bisect คอมมิตของคุณเพื่อหาการเปลี่ยนแปลงที่แนชต์ที่ทำให้เกิดบั๊ก

สาเหตุที่แท้จริง เทียบกับ อาการ

กับดักที่ล่อใจที่สุดในการดีบั๊กคือแพตช์ที่ทำให้ข้อผิดพลาด หายไป โดยไม่ได้แก้ที่สาเหตุ ถ้า user เป็น undefined และคุณ “แก้” ด้วย if (user) { ... } ข้อผิดพลาดก็จะหยุด — แต่ตอนนี้การสมัครสมาชิกจะไม่ทำอะไรเลยแบบเงียบ ๆ ซึ่งแย่กว่าเดิม: การล่มดัง ๆ อย่างน้อยก็บอกคุณว่ามีอะไรผิดปกติ แต่ตอนนี้ไม่มีอะไรบอกอีกแล้ว คำถามที่แท้จริงคือทำไม user ถึงเป็น undefined ตรงนี้? บางทีการค้นหาในฐานะข้อมูลอาจล้นเหลือ บางทีชื่อฟิลด์อาจถูกเปลี่ยน บางทีอาจขาด await ไปตรงต้นทาง

เมื่อ AI เสนอทางแก้ ให้ถามมันว่า:

- “นี่เป็นการแก้ที่สาเหตุ หรือแค่ซ่อนอาการ?”
- “เหตุผล ดั้งเดิม ที่ค่านี้ผิดคืออะไร?”
- “สาเหตุเดียวกันนี้อาจทำให้อะไรอื่นพังด้วยหรือเปล่า?”

ทางแก้ที่ตรงสาเหตุมักจะทำให้โค้ด ง่ายขึ้น หรือ ชัดเจนขึ้น ส่วนแพตช์ที่แค่แก้อาการมักจะเพิ่มการ์ด (guard), try/catch ที่กลืนข้อผิดพลาดไป หรือค่า default ที่กลบปัญหาที่แท้จริงไว้ ให้ระวังประเภทหลังนี้ไว้เสมอ มีข้อสังเกตที่ควรเรียนรู้ไว้: ถ้าทางแก้เพิ่มโค้ดที่มีหน้าที่เดียวคือ ป้องกัน การล่ม แทนที่จะ ทำสิ่งที่ถูกต้อง คุณก็อาจจะกำลังแพตช์อาการอยู่ ทางแก้ที่แท้จริงมักจะกำจัดสถานะที่เป็นไปไม่ได้ออกไปทั้งหมด ไม่ใช่แค่รอดพ้นจากมัน

เมื่อ AI วนเวียนไม่จบ

บางครั้ง AI เสนอทางแก้ แล้วมันใช้ไม่ได้ มันเสนออีกอัน แล้วก็ใช้ไม่ได้อีก และคุณก็จมอยู่ในลูป “ลองอันนี้แทน” มาสามรอบแล้ว นี่คือสัญญาณ ไม่ใช่ทางตัน ให้ทำแบบนี้:

- หยุดและรีเซ็ตบริบท เริ่มข้อความใหม่ที่สรุปทุกสิ่งที่คุณ รู้แน่ชัด ในตอนนี้ (ที่ยืนยันด้วย log แล้ว) ไม่ใช่ทุกสิ่งที่คุณเคยลอง แสทหาย ๆ ที่เต็มไปด้วยความพยายามที่ล้นเหลือเป็นพิษต่อคำตอบถัดไปของ AI จริง ๆ — มันจะยังคงจับคู่รูปแบบกับทางตันในประวัติของตัวเอง
- ระบุสิ่งที่คุณได้ตัดออกไปแล้ว “เรายืนยันแล้วว่า user เป็น undefined และการ query ฐานข้อมูลคืนค่า null สตริง query ถูกต้อง ดังนั้นปัญหาน่าจะอยู่ก่อนหน้า query” การทำแบบนี้หยุด AI จากการเสนอทางตันซ้ำ

- ขอให้มันอธิบาย ไม่ใช่แค่ “อธิบายที่ละขั้นตอนหน่อยว่า user ได้คำมาจาก request จนถึงบรรทัด 14 ได้อย่างไร” การบังคับให้อธิบายมักจะเผยให้เห็นช่องว่าง
- เพิ่ม logging ไม่ใช่การเดามากขึ้น เมื่อทฤษฎีหมดลง หลักฐานคือทางไปข้างหน้า
- ย่อขนาดปัญหาลง ขอให้ AI ช่วยคุณสร้างโค้ดตัวอย่างที่เล็กที่สุดเท่าที่จะเป็นไปได้ซึ่งยังคงทำให้บั๊กเกิดซ้ำได้บ่อยครั้งมักจะเห็นได้ชัดเจนเมื่อสัญญาณรบกวนหายไป — และถ้ามันไม่เกิดซ้ำในเวอร์ชันเล็กนั้น คุณก็ได้เรียนรู้แล้วว่าสาเหตุอยู่ในส่วนที่คุณตัดออกไป

ทำความเข้าใจทางแก้ก่อนที่จะเก็บมันไว้

ทางแก้ที่ใช้งานได้แต่คุณไม่เข้าใจนั้นเป็นเหมือนเงินกู้ ไม่ใช่ทางแก้ปัญหา — และดอกเบี้ยจะมาถึงกำหนดในครั้งต่อไปที่มีอะไรใกล้เคียงพังลง เพราะ AI เป็นคนพิมพ์ มันง่ายมากที่จะรับเครื่องหมายถูกสีเขียวแล้วเดินหน้าต่อไปโดยไม่เคยรู้เลยว่า ทำไม โค้ดถึงเป็นสีเขียวแล้ว จงถantuanสิ่งนั้น ก่อนที่คุณจะเก็บทางแก้ไว้ ให้ AI อธิบายมันให้ได้:

- “อธิบายว่าทำไมสิ่งนี้ถึงแก้ปัญหาได้ ในหนึ่งหรือสองประโยค” ถ้าคำอธิบายคลุมเครือ — เช่น “สิ่งนี้ทำให้มันทนทานขึ้น” “สิ่งนี้จัดการกรณีขอบได้” — ให้ผลักดันกลับไป คำอธิบายที่แท้จริงจะระบุสาเหตุที่ชัดเจนและวิธีที่การเปลี่ยนแปลงกำจัดมันออกไป
- “บั๊กคืออะไร ในแง่ของเหตุและผล?” คุณควรจะสามารถพูดคำตอบซ้ำด้วยคำพูดของคุณเองได้ ถ้าทำไม่ได้ แสดงว่าคุณยังไม่เข้าใจโปรแกรมของตัวเอง
- “มีส่วนอื่นใดที่ต้องต้องเส้นทางโค้ดนี้ที่ฉันควรทดสอบซ้ำ?” สิ่งนี้เปลี่ยนทางแก้ให้กลายเป็นการตรวจสอบ regression

นี่คือปรัชญา “เข้าใจสิ่งที่คุณส่งมอบ” ที่ถูกนำมาใช้กับกรณีที่ยากที่สุด คุณไม่จำเป็นต้อง เขียน ทางแก้จากศูนย์ได้ — นั่นคืองานของ AI แต่คุณจำเป็นต้องเก็บเรื่องราวของเหตุและผลไว้ในหัว เพราะคุณคือคนที่จะอยู่ตรงนั้นตอนที่สองเมื่อมันพังอีกครั้ง และสิ่งเดียวที่จะช่วยได้ในตอนนั้นคือการรู้ว่าสิ่งนั้นทำงานอย่างไรจริง ๆ

บทสรุป

การติดกับด้วย AI ไม่ใช่เรื่องของ การรู้คำตอบ แต่เป็นเรื่องของการดำเนินการกระบวนการที่มีวินัย: ทำให้เกิดซ้ำ รวบรวมบริบท ตั้งสมมติฐาน ยืนยันด้วย logging ค้นหาแบบทวิภาค และแก้ที่สาเหตุที่แท้จริง — แล้วจึงเข้าใจทางแก้ที่มันให้ดีพอที่จะเก็บไว้ได้อย่างชัดเจน หน้าที่ของคุณในฐานะผู้กำกับคือทำให้การสืบสวนยึดโยงอยู่กับหลักฐาน และรู้จักสังเกตเมื่อลูบเริ่มเน่าเสีย ทำแบบนั้น และบั๊กส่วนใหญ่จะเลิกเป็นปริศนาและกลายเป็นเพียงอีกขั้นตอนหนึ่งของการสร้างซอฟต์แวร์จริง

สรุปและฝึกฝน

ประเด็นสำคัญ

- บั๊กคือช่องว่างระหว่างสิ่งที่คุณคาดหวังกับสิ่งที่เกิดขึ้น — ให้ AI เห็นทั้งสองด้านพร้อมขั้นตอนที่ชัดเจน อย่าพูดแค่ “มันพัง”
- วางข้อความข้อผิดพลาดแบบคำต่อคำและส่วนบนสุดของ stack trace บรรทัดแรกที่ชี้ไปที่ ไฟล์ของคุณเอง มักจะเป็นจุดที่ควรเริ่มมองหาเสมอ
- ยืนยันก่อนที่จะเปลี่ยนแปลง: เพิ่มบรรทัด log ที่มีป้ายกำกับเพื่อทดสอบสมมติฐาน และใช้การ bisect (ในโค้ดหรือประวัติคอมมิต) เพื่อลดพื้นที่ค้นหาครึ่งหนึ่งในแต่ละขั้นตอน
- แก้ที่สาเหตุที่แท้จริง ไม่ใช่แค่อาการ — การหัดหรือ try/catch ที่แค่หยุดการล่มมักจะกลบปัญหาที่แท้จริงไว้
- เมื่อ AI วนเวียน ให้รีเซตบริบทกลับไปยังสิ่งที่คุณ รู้แน่ชัด ขอให้มันอธิบายแทนที่จะแก้ และเพิ่มหลักฐานแทนการเดา

ลองทำดู

ครั้งต่อไปที่มีอะไรพัง ให้เขียนรายงานบั๊ก ก่อน ที่จะขอทางแก้ ในแชทใหม่ ให้ข้อมูลสามอย่างแก่ AI: ขั้นตอนที่ทำให้เกิดซ้ำ สิ่งที่คุณคาดหวัง และสิ่งที่เกิดขึ้นจริง — พร้อมข้อผิดพลาดที่คัดลอกมาแบบคำต่อคำและเฟรมบนสุดของ stack ที่แนบมาด้วย จากนั้น แทนที่จะถามว่า “แก้เลย” ให้ขอสมมติฐานและวิธียืนยันแบบประหยัด (มักจะเป็นบรรทัด log) รัน

การตรวจสอบนั้น วางผลลัพธ์กลับมา แล้วจึงค่อยให้ AI เสนอทางแก้ สังเกตดูว่าเส้นทางตรงกว่า “มันใช้ไม่ได้ ช่วยด้วย” มากแค่ไหน

```
I have a bug. Before suggesting any fix, give me a hypothesis and a cheap way to confirm it (such as a log line) – don't edit code yet.
```

```
Steps to reproduce: [what you did]
```

```
Expected: [what should have happened]
```

```
Actual: [what happened]
```

```
Full error and stack trace:
```

```
[paste verbatim]
```

```
Relevant code:
```

```
[paste the function that failed, plus anything it calls]
```

```
What could be causing this, and what should I log to confirm it before we change anything?
```

โครงสร้างพื้นฐานเซิร์ฟเวอร์



คุณ vibe-code บางอย่างที่ทำงานได้บนแล็ปท็อปของคุณแล้ว ตอนนี้คุณอยากให้คนอื่นใช้งานมันบ้าง ชองว่างนั้น — ระหว่าง “รันไทม์เครื่องของฉัน” กับ “รันไทม์อินเทอร์เน็ต” — คือสิ่งที่โครงสร้างพื้นฐาน (infrastructure) พุดถึง ชาวดี คือ คุณไม่จำเป็นต้องเป็นสาย DevOps มาทั้งชีวิตเพื่อข้ามช่องว่างนี้ คุณแค่ต้องมีแบบจำลองความคิดที่ใช้งานได้ และ

ความยินดีที่จะปล่อยให้ AI จัดการค่าคอนฟิกร์จกัแทนคุณ

บทนี้จะไขความสลับซับซ้อนของโลกการโฮสต์ (hosting) ให้คุณ พร้อมอธิบายข้อแลกเปลี่ยนต่าง ๆ ด้วยภาษาง่าย ๆ และมอบเส้นทางเริ่มต้นที่พาผู้สร้างคนเดียวจากต้นแบบไปสู่การใช้งานจริง โดยไม่จมอยู่กับตัวเลือกมากมาย

ห้าที่อยู่ที่คุณสามารถอาศัยอยู่ได้

ผลิตภัณฑ์โฮสต์แทบทุกตัวบนโลกนี้ล้วนเป็นรูปแบบหนึ่งของห้าหมวดหมู่นี้ เรียนรู้ห้าหมวดหมู่แล้วหน้าการตลาดของบริการต่าง ๆ จะไม่สับสนอีกต่อไป

- การโฮสต์แบบสถิต (Static hosting) — ไฟล์ธรรมดา (HTML (HyperText Markup Language โครงสร้างของหน้าเว็บ), CSS (Cascading Style Sheets การจัดสไตล์), JavaScript, รูปภาพ) ที่ถูกส่งตรงไปยังเบราว์เซอร์ ไม่มีเซิร์ฟเวอร์ไหน “รัน” โค้ดของคุณ มันแค่ส่งไฟล์ให้ คิดซะว่าเป็นโพลเดอร์บนอินเทอร์เน็ต เพราะไม่มีอะไรจะฟังและไม่มีอะไรต้องอัปเดตแพตช์ นี่คือนิสัยที่เชื่อถือได้มากที่สุดที่คุณจะเอาขึ้นออนไลน์ได้ ตัวอย่าง: GitHub Pages, Cloudflare Pages, Netlify, Vercel (โฮสต์สถิต)
- ฟังก์ชันขอบเครือข่าย (Edge functions) — ชิ้นส่วนโค้ดเล็ก ๆ ที่รันบนเซิร์ฟเวอร์ที่อยู่ใกล้ผู้ใช้ของคุณทางกายภาพ กระจายอยู่ทั่วโลก มันเริ่มทำงานได้เกือบจะทันที และเหมาะกับตรรกะเบา ๆ เช่น การตรวจสอบสิทธิ์ การรีไดเรกต์ การเรียก API (Application Programming Interface — ประตุนี้นี่โปรแกรมหนึ่งใช้คุยกับอีกโปรแกรม) เล็ก ๆ ข้อเสียคือมันรันในสภาพแวดล้อมที่ถูกตัดตอน ทำให้ไม่ใช่ทุกไลบรารีจะใช้งานได้ทันที ตัวอย่าง: Cloudflare Workers, Vercel Edge Functions, Deno Deploy
- ฟังก์ชันเซิร์ฟเวอร์เลส (Serverless functions) — โค้ดที่รันตามความต้องการและหายไปเมื่อไม่มีใครใช้ คุณจ่ายเงินตามจำนวนคำขอ ไม่ใช่ตามชั่วโมง คุณเขียนฟังก์ชันขึ้นมา ส่วนแพลตฟอร์มจัดการเรื่องเครื่องจักรให้ ใช้งานได้ดีจนกว่าคำขอเดียวจะต้องรอนานหลายนาที — แพลตฟอร์มส่วนใหญ่จะตัดฟังก์ชันทิ้งหลังจากหมดเวลา (มักจะ 10–60 วินาที) ตัวอย่าง: AWS Lambda, Google Cloud Functions, Vercel Functions
- คอนเทนเนอร์ (Containers) — แอปของคุณพร้อมทั้งสภาพแวดล้อมทั้งหมด (เวอร์ชันภาษาที่ต้อง ไลบรารีเครื่องมือระบบ) ที่ถูกบรรจุลงในกล่องที่พกพาได้เรียกว่าคอนเทนเนอร์ แพลตฟอร์มจะทำให้กล่องนั้นทำงานต่อเนื่อง ควบคุมได้มากขึ้น แลกกับการตั้งค่าที่มากขึ้นเล็กน้อย ไฟล์เดียวที่กำหนดกล่องนี้คือ Dockerfile และ AI เขียนไฟล์แบบนี้ได้ดีมาก ตัวอย่าง: Fly.io, Railway, Render, Google Cloud Run
- เครื่องเสมือน (Virtual machines หรือ VMs) — คอมพิวเตอร์ทั้งเครื่องที่คุณเช่าในศูนย์ข้อมูล คุณติดตั้งและจัดการทุกอย่างเอง — ระบบปฏิบัติการ อัปเดตความปลอดภัย เว็บเซิร์ฟเวอร์ ทุกอย่าง ควบคุมได้สูงสุด แต่รับผิดชอบสูงสุดเช่นกัน ตัวอย่าง: AWS EC2, DigitalOcean Droplets, Hetzner

เมื่อวางเรียงกัน ห้าหมวดหมู่จะแลก สิ่งที่ต้องจัดการน้อยลง กับ การควบคุมที่มากขึ้น เมื่อไล่ลงไปเรื่อย ๆ :

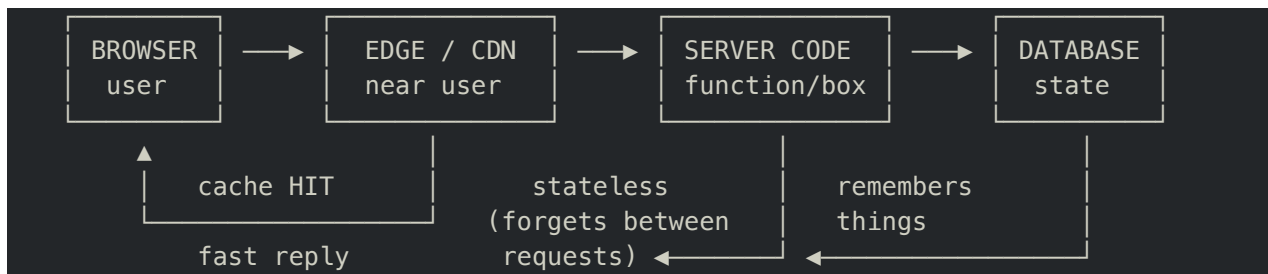


หลักการคร่าว ๆ เรื่องต้นทุน: ยิ่งแพลตฟอร์มจัดการให้คุณมากเท่าไร มันก็ยิ่งถูกเมื่อไม่มีใครใช้แอปของคุณ และมันยิ่งช่วยแบ่งความกังวลมากขึ้นเมื่อทุกคนกำลังใช้งานพร้อมกัน

ศัพท์เทคนิคสองสามคำ ถอดรหัสให้แล้ว

- Cold start (การเริ่มเครื่องเย็น) — เมื่อโค้ดไม่ได้รันมาสักพัก แพลตฟอร์มต้องปลุกมันขึ้นมา ค่าขอแรกนั้นจะช้ากว่า (เศษเสี้ยววินาทีไปจนถึงสองสามวินาที) ฟังก์ชันขอบเครือข่ายแทบไม่มีปัญหานี้ ฟังก์ชันเซิร์ฟเวอร์เลสบางครั้งก็มี ส่วนคอนเทนเนอร์และ VM ที่รันตลอดเวลาไม่มีปัญหานี้เลย สำหรับแอปส่วนใหญ่สิ่งนี้มองไม่เห็นเลย แต่สำหรับขั้นตอนการชำระเงินที่ต้องการความเร็ว มันอาจสำคัญ
- การขยายขนาด (Scaling) — การรองรับผู้ใช้จำนวนมากขึ้นโดยอัตโนมัติ (automatic) หมายถึงแพลตฟอร์มเพิ่มกำลังให้คุณเอง แบบมือ (manual) หมายถึงคุณต้องกดปุ่มเอง หรือไม่มันก็จะพังเมื่อโหลดสูง “Scales to zero” (ขยายลงถึงศูนย์) เป็นวลีที่เกี่ยวข้องที่คุณจะเห็นบ่อย: เมื่อไม่มีใครใช้แอปของคุณ มันจะไม่มีค่าใช้จ่ายเลย
- สถานะ (State) — ข้อมูลที่ต้องคงอยู่ (ฐานข้อมูล ไฟล์ที่อัปโหลด เซสชันของผู้ใช้) เลเยอร์แบบสถิตและเซิร์ฟเวอร์เลสมักจะเป็น “stateless” — คือลืมทุกอย่างระหว่างคำขอ — ดังนั้นสถานะจึงอยู่ในฐานข้อมูลหรือบริการจัดเก็บข้อมูลแยกต่างหาก นี่คือเหตุผลที่แอปจริงเกือบทุกตัวคือ “การโฮสต์ของคุณบวกกับฐานข้อมูลที่ไหนสักแห่ง” ไม่ใช่กล่องวิเศษกล่องเดียว
- ภูมิภาค (Region) — ส่วนไหนของโลกที่โค้ดและข้อมูลของคุณตั้งอยู่ทางกายภาพ มันส่งผลต่อความเร็ว (ใกล้กว่า = เร็วกว่า) และบางครั้งก็ส่งผลกระทบต่อกฎหมาย (ข้อมูลบางอย่างต้องอยู่ในประเทศใดประเทศหนึ่ง) สำหรับโปรเจกต์แรก เลือกภูมิภาคที่ใกล้ผู้ใช้ของคุณที่สุดแล้วก็เดินหน้าต่อ

นี่คือเส้นทางเดินทางทั้งหมดของคำขอ และแต่ละส่วนอาศัยอยู่ที่ไหน เบราวเซอร์คุยกับขอบเครือข่ายที่อยู่ใกล้ ซึ่งจะส่งไฟล์ที่แคชไว้หรือปลุกโค้ดของคุณขึ้นมา ซึ่งจะไปติดต่อฐานข้อมูลสำหรับสิ่งที่มันต้องจำ:



ตารางข้อแลกเปลี่ยน

ประเภทการโฮสต์	ต้นทุน (ว่าง → คึกคัก)	การขยายขนาด	Cold starts	ความซับซ้อน	เหมาะกับ
แบบสถิต	ฟรี → ถูกมาก	อัตโนมัติ ง่ายตาย	ไม่มี	ต่ำที่สุด	เว็บไซต์การตลาด เอกสารบล็อก ฟรอนต์เอนด์
ฟังก์ชันขอบเครือข่าย	ระดับฟรี → ถูก	อัตโนมัติ ทั่วโลก	แทบไม่มี	ต่ำ	การยืนยันตัวตน รีไดเรกต์ API เบา ๆ การปรับแต่งส่วนบุคคล
เซิร์ฟเวอร์เลส	\$0 เมื่อว่าง → ขยายตามการใช้งาน	อัตโนมัติ	บางครั้งสังเกตได้	ต่ำ-ปานกลาง	API, webhook, ทราฟฟิกที่ไม่แน่นอน/เป็นช่วง ๆ
คอนเทนเนอร์	จ่ายขณะที่รัน	อัตโนมัติแบบมีเงื่อนไข ปรับได้	ไม่มี (ถ้าเปิดค้างไว้)	ปานกลาง	แบ็กเอนด์เต็มรูปแบบ งานที่รันต่อเนื่องยาวนาน รันไทม์แบบกำหนดเอง

ประเภทการโฮสต์	ต้นทุน (ว่าง → คึกคัก)	การขยายขนาด	Cold starts	ความซับซ้อน	เหมาะกับ
VM	จ่าย 24/7	แบบมือ	ไม่มี	สูงที่สุด	ฮาร์ดแวร์พิเศษ ซอฟต์แวร์รุ่นเก่า การควบคุม ทั้งหมด

รูปแบบคือ: ยิงเลื่อนลงในตารางมากเท่าไร คุณก็ยิ่งแลก ความเรียบง่ายและต้นทุนตอนว่างที่ต่ำ กับ การควบคุมและความยืดหยุ่น ผู้ vibe coder ส่วนใหญ่ควรพยายามอยู่ใกล้ ๆ ด้านบนของตาราง

จะเลือกอันไหนเมื่อไหร่

- แค่วีเบโค้ดหรือฟรอนต์เอนด์เฉย ๆ? โฮสต์แบบสถิต จบ มันฟรี เร็ว และแทบจะพังไม่ได้เลย
- ต้องการตรรกะแบ็กเอนด์เล็ก ๆ น้อย ๆ — ฟังก์ชันติดต่อ, ฤกษ์ API ที่ต้องเก็บเป็นความลับ, การตรวจสอบสิทธิ์? ฟังก์ชันขอบเครือข่ายหรือเซิร์ฟเวอร์เลส คุณยังคงอยู่บนระดับฟรีที่ใช้ได้มาก และขยายลงถึงศูนย์เมื่อไม่มีใครใช้
- แอปจริงที่มีฐานข้อมูล งานเบื้องหลัง หรือเฟรมเวิร์กที่คาดหวังเซิร์ฟเวอร์ที่รันต่อเนื่องยาวนาน? แพลตฟอร์มคอนเทนเนอร์อย่าง Railway, Render หรือ Fly.io นี่คือจุดที่เหมาะสมที่สุดสำหรับโปรเจกต์ “ผลิตภัณฑ์จริง” ส่วนใหญ่
- ต้องการ GPU เฉพาะ ซอฟต์แวร์ระบบพิเศษ หรือควบคุมทั้งหมด? VM ไปที่นั่นเมื่อสิ่งที่กล่าวมาข้างต้นทำงานนี้จริง ๆ ไม่ได้เท่านั้น เพราะตอนนี้ คุณ คือผู้ดูแลระบบแล้ว

กฎที่มีประโยชน์: เลือกตัวเลือกที่ง่ายที่สุดที่ทำงานได้ และเลื่อนระดับก็ต่อเมื่อคุณเจอกำแพงจริง ๆ — ไม่ใช่กำแพงในจินตนาการ

“กำแพงจริง” หน้าตาเป็นยังไงกันแน่? ตัวอย่างจริงสองสามอย่าง: การ build ของคุณสร้างเซิร์ฟเวอร์ที่ต้องคงรันต่อเนื่อง (แบ็กเอนด์แชทที่ถือการเชื่อมต่อสด ๆ ไว้) ดังนั้นแบบสถิตใช้ไม่ได้ หรือฟังก์ชันหนึ่งชนกำแพงเวลาหมดอายุครั้งแล้วครั้งเล่าเพราะค่าขอเดี่ยวยังต้องทำงานหนักจริง ๆ (ปรับขนาดวิดีโอ สร้าง PDF) จึงต้องเลื่อนขึ้นไปเป็นคอนเทนเนอร์ หรือซิดจำกัดค่าขอของระดับฟรีของคุณถูกใช้จนหมดจริง ๆ จากทราฟฟิกจริง สิ่งที่เกิดขึ้นที่ตัวอย่างเหล่านี้มีร่วมกัน: แต่ละอย่างคือการวัดผล ไม่ใช่ความรู้สึก “มันอาจจะโตขึ้นในอนาคต” ไม่ใช่กำแพง

AI ต่อสายการ deploy และการตั้งค่าอย่างไร

นี่คือจุดที่ vibe coding ส่องแสง ค่าคอนฟิกการ deploy เป็นงานประเภทที่แม่นยำ มีเอกสารกำกับชัดเจน เต็มไปด้วยงานซ้ำ ๆ ตามแบบแผน ที่ AI เก่งมาก คุณเป็นผู้กำกับทิศทาง มันเป็นผู้เขียน

พธอมป์ที่ได้ผลจะมีลักษณะแบบนี้:

- “เพิ่ม Dockerfile ให้แอป Node นี้ และค่าคอนฟิก Railway เพื่อ deploy มัน”
- “เขียน GitHub Actions workflow ที่ deploy โพลเดอร์ dist/ ไปยัง Cloudflare Pages ทุกครั้งที่ push ไปยัง main”
- “ฟังก์ชันเซิร์ฟเวอร์เลสของฉันเจอ cold-start timeout — น่าจะเกิดจากอะไร และฉันจะแก้ค่าคอนฟิกยังไง?”

นี่คือตัวอย่างไฟล์ที่ AI จะสร้างให้คุณ — workflow การ deploy ที่ส่งเว็บไซต์แบบสถิตออกไปโดยอัตโนมัติทุกครั้งที่คุณ push โค้ด:

```
# .github/workflows/deploy.yml
name: Deploy
on:
  push:
    branches: [main]
jobs:
```

```

build-and-deploy:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: 20
    - run: npm ci && npm run build
    - name: Publish to Cloudflare Pages
      uses: cloudflare/wrangler-action@v3
      with:
        apiToken: ${ secrets.CF_API_TOKEN }
        command: pages deploy dist --project-name=my-app

```

คุณไม่จำเป็นต้องจำไวยากรณ์นี้ให้ขึ้นใจ คุณแค่ต้องเข้าใจว่ามันทำอะไร: ทุกครั้งที่ push ไปยัง main มันจะ checkout โค้ดของคุณ build แล้วเผยแพร่ผลลัพธ์ เมื่อมีอะไรพัง คุณก็แค่ดลอกข้อความ error กลับไปให้ AI แล้วขอให้มันแก้ค่าคอนฟิก

ผู้สร้างหลายคนข้ามไฟล์ GitHub Actions ไปเลยในตอนแรก และแค่ deploy จากเทอร์มินัลของตัวเองด้วยคำสั่งเดียวนั้นมักเป็นวิธีที่เร็วที่สุดที่จะเห็นอะไรบางอย่างขึ้นออนไลน์ในวันแรก การ deploy Cloudflare Pages แบบเดียวกันหน้าตาแบบนี้จากบรรทัดคำสั่ง:

```

# install the CLI once
npm install -g wrangler

# build your site, then push the output folder live
npm run build
wrangler pages deploy dist --project-name=my-app

```

การรันครั้งแรกจะขอให้คุณล็อกอินผ่านเบราว์เซอร์ หลังจากนั้นก็เหลือแค่คำสั่งเดียว ข้อแลกเปลี่ยนนั้นง่ายมาก: คำสั่งเทอร์มินัลเหมาะกับ “deploy เดี๋ยวนี้เลย” และไฟล์ GitHub Actions เหมาะกับ “deploy อัตโนมัติตลอดไป” โปรเจกต์ส่วนใหญ่เริ่มจากแบบแรก แล้วค่อยเพิ่มแบบหลังเมื่อการ deploy กลายเป็นเรื่องปกติ ถาม AI ว่าแบบไหนเหมาะกับจุดที่คุณอยู่

เคล็ดลับการกำกับทิศทางสองสามข้อที่ช่วยประหยัดความเจ็บปวดจริง ๆ:

- ให้ AI อธิบายก่อนลงมือ “ก่อนเขียนค่าคอนฟิก บอกฉันหน่อยว่าประเภทการโฮสต์แบบไหนเหมาะกับโปรเจกต์นี้และทำไม” วิธีนี้จับการออกแบบที่ซับซ้อนเกินจำเป็นได้ตั้งแต่เนิ่น ๆ
- เก็บความลับให้อยู่นอกโค้ด คุญแจ API และรหัสผ่านต้องอยู่ในตัวแปรสภาพแวดล้อม / ที่เก็บความลับของแพลตฟอร์ม ไม่ใช่ในไฟล์ที่คุณ commit ขอให้ AI ใช้การอ้างอิงความลับ เหมือนกับ CF_API_TOKEN ด้านบน
- Deploy แต่เนิ่น ๆ deploy บ่อย ๆ ให้ “hello world” ขึ้นออนไลน์ในวันแรก ไปป์ไลน์ที่ทำงานมาตั้งแต่ต้นดีก็ง่ายกว่ามากเมื่อเทียบกับที่ประกอบขึ้นตอนเปิดตัว
- ขอเรื่องราวการย้อนกลับ (rollback) ล่วงหน้า “ถ้า deploy นี้พัง ฉันจะกลับไปหาเวอร์ชันที่ใช้งานได้ยังไง?” แพลตฟอร์มดี ๆ จะเก็บ deploy ก่อนหน้าของคุณไว้ให้กดกลับได้ในคลิกเดียว การรู้ว่าปุ่มย้อนกลับมีอยู่ก่อนที่คุณจะต้องไข่ม้น เปลี่ยนความตื่นตระหนกตอนดีสองให้กลายเป็นแค่การยกไหล่

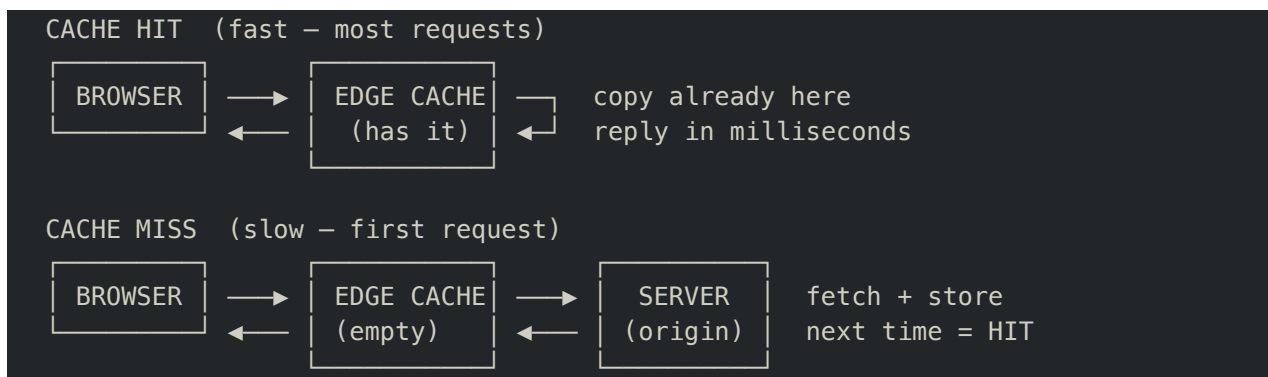
เส้นทางเริ่มต้นที่แนะนำสำหรับผู้สร้างคนเดียว

นี่คือเส้นทางที่ใช้ได้ผลกับโปรเจกต์คนเดียวฉบับไม่ถาวร แต่ละขั้นตอนจะทำก็ต่อเมื่อคุณต้องการมันจริง ๆ เท่านั้น

1. ต้นแบบ: แบบสกิด + บริการแบ็กเอนด์ที่มีคนจัดการให้ วางฟรอนต์เอนด์ของคุณไว้บนโฮสต์แบบสกิด (Cloudflare Pages, Vercel หรือ Netlify) สำหรับข้อมูลและการยืนยันตัวตน ฟังพาแบ็กเอนด์ที่มีคนจัดการให้อย่าง Supabase หรือ Firebase เพื่อให้คุณไม่ต้องรันฐานข้อมูลเอง ต้นทุน: แทบจะเป็นศูนย์

2. ช่วงการใช้งานจริงระยะแรก: เพิ่มฟังก์ชันขอบเครือข่ายหรือเซิร์ฟเวอร์เลส เมื่อคุณต้องการตรรกะฝั่งเซิร์ฟเวอร์ — การเรียก API ลับ ๆ, webhook, การยืนยันตัวตนแบบกำหนดเอง — เพิ่มฟังก์ชันบนแพลตฟอร์มเดียวกัน ยังคงขยายลงถึงศูนย์ ยังคงถูก ยังคง deploy ครั้งเดียวจบ
3. ผลลัพธ์ที่จริง: ย้ายแบ็กเอนด์ไปยังแพลตฟอร์มคอนเทนเนอร์ เมื่อแอปเติบโตเกินฟังก์ชัน (กระบวนการที่รันต่อเนื่องยาวนาน เฟรมเวิร์กที่ต้องการเซิร์ฟเวอร์แบบถาวร งานเบื้องหลัง) ให้บรรจุมันเป็นคอนเทนเนอร์แล้ว deploy ไปยัง Railway, Render หรือ Fly.io ให้ AI เขียน Dockerfile และค่าคอนฟิกของแพลตฟอร์ม
4. ขยายขนาดหรือเฉพาะทาง: พิจารณา VM หรือผู้ให้บริการคลาวด์อย่าง AWS เมื่อถึงตอนนั้นเท่านั้น นี่คือการจุดที่ความซับซ้อน (และค่าใช้จ่ายของคุณ) พุ่งขึ้น ผู้สร้างคนเดียวส่วนใหญ่ไม่เคยต้องการขั้นนี้เลย และนั่นคือสัญญาณว่ากำลังทำสิ่งถูกต้อง ไม่ใช่ข้อจำกัด

CDN (Content Delivery Network — เครือข่ายกระจายเนื้อหา) อยู่หน้าเส้นทางนี้ และเปลี่ยนความเร็วอย่างมากขึ้นอยู่กับว่ามันมีสำเนาอยู่แล้วหรือไม่ cache HIT ตอบกลับทันทีจากขอบเครือข่าย ส่วน cache MISS ต้องเดินทางกลับไปยังเซิร์ฟเวอร์ของคุณก่อน:



ข้อผิดพลาดที่ควรหลีกเลี่ยงคือการเริ่มต้นที่ขั้นที่ 4 เพราะคิดว่ามันคือสิ่งที่ “วิศวกรตัวจริง” ใช้กัน ความจริงตรงกันข้าม: การเริ่มต้นแบบง่าย ๆ คือทำทางของมือโปร คุณสามารถเลื่อนขั้นไปสู่การควบคุมที่มากขึ้นได้เสมอ แต่คุณไม่สามารถเรียกเดือนเวลาที่เสียไปกับการต่อสู้กับโครงสร้างพื้นฐานที่คุณไม่ได้ต้องการกลับคืนมาได้ง่าย ๆ

“มีคนจัดการให้” หมายถึงอะไรกันแน่ (และคุณยังต้องรับผิดชอบอะไรอยู่)

บริการที่มีคนจัดการให้ (managed service) คือบริการที่แพลตฟอร์มจัดการส่วนที่น่าเบื่อ และอันตรายให้คุณ — เซิร์ฟเวอร์ แพตช์ความปลอดภัย การขยายขนาด การสำรองข้อมูล “เซิร์ฟเวอร์เลส” “ฐานข้อมูลที่มีคนจัดการให้” และ “platform as a service” ส่วนเป็นรูปแบบต่าง ๆ ของข้อตกลงเดียวกัน: คุณส่งมอบงานปฏิบัติการและมุ่งสร้างต่อ สำหรับผู้สร้างคนเดียว นี่ไม่ใช่ความขี้เกียจ มันคือเลเวอเรจ ชั่วโมงที่คุณไม่ต้องใช้ไปกับการอัปเดตแพตช์ความปลอดภัยของ Linux คือชั่วโมงที่คุณใช้ไปกับผลลัพธ์จริง ๆ

แต่ “มีคนจัดการให้” ไม่ได้แปลว่า “วิเศษ” มีบางเรื่องที่ยังคงเป็นความรับผิดชอบของคุณ ไม่ว่าแพลตฟอร์มจะจัดการมากแค่ไหนก็ตาม:

- ข้อมูลของคุณ แพลตฟอร์มทำให้ฐานข้อมูลรันต่อไป แต่การมั่นใจว่าคุณมีข้อมูลสำรองที่คุณเคยทดสอบกู้คืนจริง ๆ แล้วนั้นเป็นหน้าที่ของคุณ กดปุ่ม “export” สักครั้งแล้วยืนยันว่าไฟล์ไม่ว่างเปล่า
- ความลับของคุณ กุญแจ API อยู่ในที่เก็บความลับของแพลตฟอร์ม แต่การหมุนเวียน (rotate) กุญแจที่รั่วไหล และการไม่วางกุญแจลงในรีโพสิทอรีสาธารณะหรือภาพหน้าจอ เป็นหน้าที่ของคุณ
- บิลของคุณ การขยายขนาดอัตโนมัติมีวิเศษมาก จนกว่ามันจะขยายขนาดไบต์แฉ่งนี้ของคุณโดยอัตโนมัติเช่นกัน ตั้งการแจ้งเตือนและเพดานการใช้จ่าย — หัวข้อถัดไปพูดถึงเรื่องนี้ทั้งหมด เพราะมันคือความล้มเหลวที่เจ็บปวดที่สุด
- ความเข้าใจของคุณ เมื่อมีอะไรพังตอนเที่ยงคืน “แพลตฟอร์มจัดการให้” ช่วยได้แค่ระดับหนึ่งเท่านั้น คุณไม่จำเป็นต้องเป็นผู้เชี่ยวชาญ แต่คุณควรอ่าน log ได้และอธิบายความล้มเหลวให้ AI ฟังได้อย่างชัดเจน นั่นคือปรัชญา “เข้าใจสิ่งที่คุณส่งมอบ” ทั้งหมดในประโยคเดียว

ทัศนคติที่ดีต่อสุขภาพ: ส่งมอบ งานปฏิบัติการ ไม่ใช่ ความเข้าใจ ให้แพลตฟอร์มรันเครื่องจักรไป ส่วนคุณยังคงเป็นคน ที่รู้ว่าเครื่องจักรเหล่านั้นกำลังทำอะไรอยู่

รายการตรวจสอบก่อนเปิดตัวแบบรวดเร็ว

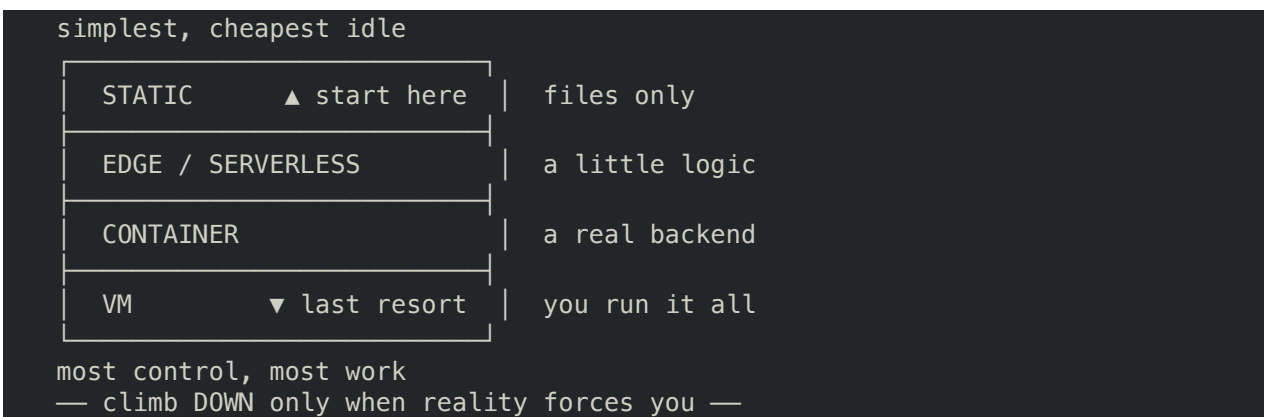
ก่อนบอกให้คนไปเยี่ยมชม URL ที่ใช้งานจริงของคุณ เดินตามรายการนี้ก่อน ไม่มีข้อไหนต้องการความเชี่ยวชาญด้าน DevOps เลย — ส่วนใหญ่คือแค่หนึ่งพรมปีให้ AI หรือหนึ่งคลิกในแดชบอร์ด

- มันโหลดได้จริง เปิด URL จริง (Uniform Resource Locator — ที่อยู่เว็บ) ในเบราว์เซอร์ที่เพิ่งเปิดใหม่ (หรือมือ ถือของคุณ) ไม่ใช่แค่ตัวอย่างในเครื่องของคุณเอง จำนวนการเปิดตัวที่ “มันทำงานได้บนเครื่องของฉัน” แล้วกลับ ขึ้น 404 ต่อสาธารณะนั้นมียู่มากมายในตำนาน
- ความลับไม่ได้อยู่ในโค้ด ค้นหาโทเคนของคุณเพื่อหากุญแจของคุณเองก่อน push ถาม AI ว่า: “สแกนโปรเจกต์นี้หา กุญแจ API รหัสผ่าน หรือ token ที่ฝังไว้ในโค้ด”
- เพดานการใช้จ่ายถูกตั้งไว้แล้ว เพดานตายตัวสำหรับทุก API ที่มีค่าใช้จ่าย และการแจ้งเตือนการเรียกเก็บเงินบน บัญชีโฮสต์ของคุณ ตั้งเพดานก่อน แล้วค่อยฉลอง
- เปิดใช้งาน HTTPS แล้ว HTTPS (HyperText Transfer Protocol Secure) คือเวอร์ชันที่เข้ารหัสของระบบ ส่งข้อมูลพื้นฐานของเว็บ — มันคือสิ่งที่ทำให้เกิดไอคอนแม่กุญแจในแถบที่อยู่ แพลตฟอร์มสมัยใหม่ทำสิ่งนี้ให้อัตโนมัติ แต่ยืนยันว่าไอคอนแม่กุญแจเล็ก ๆ นั้นมีอยู่จริง เบราว์เซอร์ลงโทษเว็บไซต์ที่ไม่ได้เข้ารหัส
- คุณรู้วิธีย้อนกลับ (roll back) หาปุ่ม deploy ก่อนหน้าตอนนี้เลย ในขณะที่ยังไม่มียะไรไฟไหม้
- มีช่องทางให้ติดต่อคุณได้ เครื่องมือบันทึก log หรือรายงานข้อผิดพลาด แม้จะเป็นแบบพื้นฐานก็ตาม เพื่อให้ สัญญาณแรกของปัญหาไม่ใช่ผู้ใช้ที่โกรธเกรี้ยว ขอให้ AI ต่อสายระบบบันทึก error ให้กับสแต็กของคุณ

นี่ไม่ใช่การตรวจสอบการใช้งานจริงแบบครอบคลุมทั้งหมด — มันคือรายการสั้น ๆ ที่จับข้อผิดพลาดที่น่าอายและมีค่าใช้จ่าย และหลีกเลี่ยงได้ง่าย ๆ ทำมันทุกครั้งที่เปิดตัวจนกว่ามันจะกลายเป็นปฏิกิริยาอัตโนมัติ

สรุปส่งท้าย

โครงสร้างพื้นฐานเป็นบันได ไม่ใช่การกระโดดข้าม แบบสถิตยูนิต VM อยู่ล่างสุด และงานส่วนใหญ่ของคุณควรอยู่ อย่างสุขสบายบนชั้นบน ๆ



รู้จักห้าหมวดหมู่ เลือกอันที่ง่ายที่สุดที่ทำงานได้ ให้ AI เขียนค่าคอนฟิก และส่งบางอย่างขึ้นออนไลน์ให้เร็วที่สุดเท่าที่ทำได้ ที่เหลือก็แค่การเลื่อนลงหนึ่งขั้นเมื่อ — และเมื่อเท่านั้น — ความเป็นจริงบังคับให้คุณต้องทำ

สรุปและฝึกฝน

ประเด็นสำคัญ

- โครงสร้างพื้นฐานเป็นบันไดห้าชั้น — โฮสต์แบบสถิต, เซิร์ฟเวอร์เลส, PaaS (Platform as a Service — โฮสต์ที่ รันแอปของคุณเพื่อให้คุณไม่ต้องจัดการเซิร์ฟเวอร์เอง), คอนเทนเนอร์ที่มีคนจัดการให้ และ VM ดิบ ๆ — และคุณ

ควรเลือกชั้นที่ง่ายที่สุดที่ทำงานได้

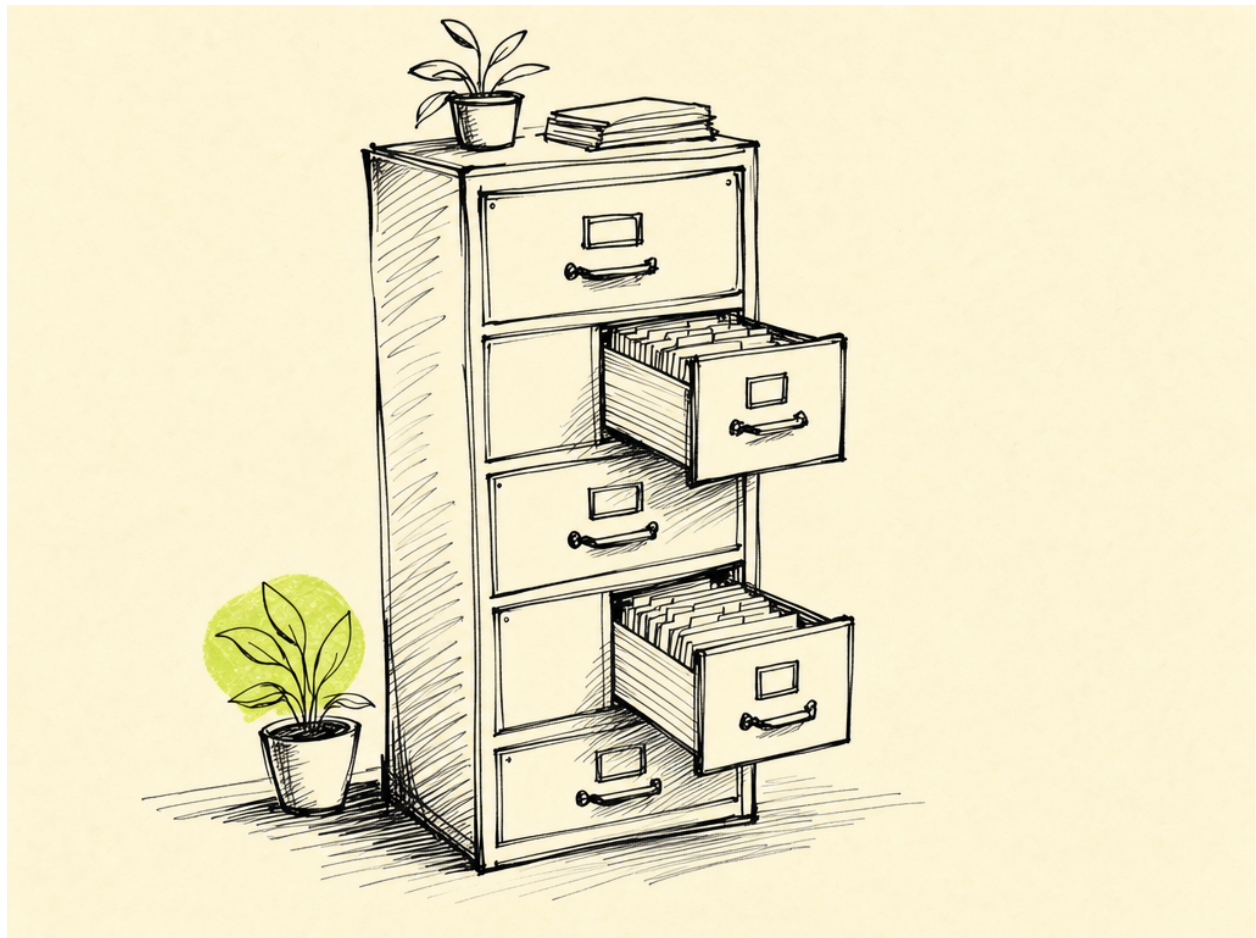
- เลือกบริการ ที่มีคนจัดการให้ (managed) เป็นหลัก: ส่งมอบงานปฏิบัติการ (เซิร์ฟเวอร์ แพตช์ การขยายขนาด การสำรองข้อมูล) ในขณะที่ยังคงความเข้าใจไว้
- ให้ AI เขียนคำคอนฟิกการ deploy และ workflow ของ CI (Continuous Integration — หุ่นยนต์อัตโนมัติที่รันการตรวจสอบทุกครั้งที่คุณ push โค้ด) ให้คุณ แต่อ่านมันก่อนที่จะเชื่อใจมัน
- “มีคนจัดการให้” ไม่ใช่ “วิเศษ” — ข้อมูลของคุณ ความลับของคุณ บิลของคุณ และความเข้าใจของคุณ ยังคงเป็นความรับผิดชอบของคุณ
- กำหนดเพดานการใช้จ่าย ยืนยัน HTTPS และรู้วิธีย้อนกลับ ก่อน ที่คุณจะทำเปิดตัว

ลองทำดู

หยิบโปรเจกต์เล็ก ๆ ที่คุณมีอยู่แล้ว (แม้แต่หน้า HTML เดียว) แล้ว deploy มันไปยังโฮสต์แบบสแตติก จากนั้นขอให้ AI สร้าง GitHub Actions workflow ที่ deploy ใหม่ทุกครั้งที่คุณ push อ่าน YAML ที่สร้างขึ้นทีละบรรทัด และยืนยันว่าคุณสามารถหาปุ่ม “ย้อนกลับไปยัง deploy ก่อนหน้า” ในแดชบอร์ดของโฮสต์ของคุณได้

```
I have a static site in this repo. Walk me through deploying it to a free static host, then generate a GitHub Actions workflow that rebuilds and redeploys on every push to main. Explain each step of the YAML, and tell me where in the dashboard I roll back a bad deploy.
```

โครงสร้างพื้นฐานฐานข้อมูล



แอปจริงทุกตัวสุดท้ายแล้วต้องมี “ความจำ” — ผู้ใช้ คำสั่งซื้อ ข้อความ การตั้งค่า ความจำเหล่านั้นอยู่ในฐานข้อมูลสำหรับมือใหม่ คำว่า “ฐานข้อมูล” อาจรู้สึกเหมือนกล่องดำที่มีแต่ผู้เชี่ยวชาญเท่านั้นที่เข้าใจ แต่ความจริงไม่ได้เป็นแบบนั้น บทนี้จะไขความลึกกลับของชั้นข้อมูล (data layer) เพื่อให้คุณตัดสินใจได้ดี และสั่งให้ AI ทำงานหนักได้อย่างปลอดภัย

สิ่งที่ควรจดจำไว้ตั้งแต่ต้น คือ โค้ดสามารถเขียนใหม่ได้ แต่ข้อมูลส่วนใหญ่เขียนใหม่ไม่ได้ คุณสามารถเขียนฟังก์ชันที่มีปีกใหม่ได้ภายในปายเดียว แต่คอลัมน์ที่คุณทำเสียหายไปเมื่อวันอังคารที่แล้ว — หรือแบ็กอัพที่คุณไม่เคยทำไว้ — อาจหายไปตลอดกาล ความไม่สมมาตรนี้เองที่ทำให้ชั้นข้อมูลคุ่มค่ากับการทำความเข้าใจล่วงหน้าเล็กน้อย มากกว่าส่วนอื่นเกือบทุกส่วนของสแต็ก (stack) ของคุณ

รูปแบบทั้งสามของการจัดเก็บข้อมูล

ฐานข้อมูลส่วนใหญ่แบ่งออกเป็นสามตระกูล คุณไม่จำเป็นต้องจำชื่อผลิตภัณฑ์ทั้งหมด — คุณแค่ต้องรู้ว่า รูปแบบ ไหนเหมาะกับปัญหาของคุณ

ประเภท	ตัวอย่าง	เหมาะกับ	ข้อแลกเปลี่ยน
เชิงสัมพันธ์ (SQL)	PostgreSQL, MySQL, SQLite	ข้อมูลที่มีโครงสร้างและความสัมพันธ์กัน; อะไรก็ตามที่เกี่ยวกับเงินหรือต้องแม่นยำ	ต้องออกแบบสคีมา (schema) ไว้ล่วงหน้า; การสเกลการเขียนข้อมูลจำนวนมากทำได้ยากกว่า
เอกสาร / NoSQL	MongoDB, Firestore, DynamoDB	ข้อมูลที่ยืดหยุ่นหรือซ้อนกัน, การพัฒนาแบบเร็ว, รูปแบบเรคคอร์ดที่หลากหลาย	สร้างข้อมูลที่ไม่สอดคล้องกันได้ง่าย; ความสัมพันธ์และการทำรายงานจะยุ่งยาก
คีย์-แวลู	Redis, Memcached, Cloudflare KV	การแคช, เซสชัน, ตัวนับ, การค้นหาที่รวดเร็วด้วยคีย์ที่รู้อยู่แล้ว	ค้นหาตามเนื้อหาไม่ได้; มักไม่ใช่แหล่งข้อมูลหลักของคุณ

ทั้งสามรูปแบบเก็บข้อมูลด้วยวิธีที่แตกต่างกันจริง ๆ ลองนึกภาพเทียบกันดู

RELATIONAL (SQL) rows + columns	DOCUMENT (NoSQL) self-contained blobs	KEY-VALUE one key → one value															
<table><tr><th>id</th><th>email</th><th>name</th></tr><tr><td>1</td><td>a@...</td><td>Ana</td></tr><tr><td>2</td><td>b@...</td><td>Bob</td></tr></table>	id	email	name	1	a@...	Ana	2	b@...	Bob	<pre>{ id, email, name, posts: [...] }</pre>	<table><tr><th>key</th><th>value</th></tr><tr><td>session</td><td>"x9f2"</td></tr><tr><td>count</td><td>"42"</td></tr></table>	key	value	session	"x9f2"	count	"42"
id	email	name															
1	a@...	Ana															
2	b@...	Bob															
key	value																
session	"x9f2"																
count	"42"																
tables reference each other	everything nested	no querying inside – look up by key only															

ลองนึกภาพฐานข้อมูลเชิงสัมพันธ์เป็นชุดสเปรดชีตที่รู้วิธีอ้างอิงถึงกันเอง ภาษาที่คุณใช้คุยกับมันคือ SQL (Structured Query Language — ภาษาสำหรับสอบถามโครงสร้าง) คำเริ่มต้นที่ดีสำหรับโปรเจกต์ 90% คือ เริ่มด้วยฐานข้อมูลเชิงสัมพันธ์ (Postgres) มันเป็นเทคโนโลยีที่เติบโตเต็มที่ คาดเดาได้ บังคับให้มีโครงสร้าง และรองรับความสัมพันธ์ที่แอปจริงยอมเติบโตขึ้นมาในที่สุด หันไปใช้ document store เมื่อข้อมูลของคุณไม่มีรูปแบบตายตัวจริง ๆ และใช้ key-value store เป็นชั้น แคช (การแคช) ไม่ใช่ที่เก็บข้อมูลหลักของคุณ

กับดักที่ควรหลีกเลี่ยงคือการเลือกตามความรู้สึก ประโยคที่ว่า “NoSQL (ฐานข้อมูลที่ไม่ใช่ตาราง SQL — เก็บเรคคอร์ดแบบยืดหยุ่น ไม่มีรูปแบบตายตัว) สเกลได้ดีกว่า” เป็นจริงเฉพาะกับปัญหากลุ่มแคบ ๆ ที่แอปส่วนใหญ่ไม่เคยเจอ และแม้จะเจอ ราคาที่ต้องจ่ายคือการยอมสละการรับประกันความสอดคล้อง (consistency) ที่ทำให้ข้อมูลน่าเชื่อถือ เลือกรูปแบบที่ตรงกับ ข้อมูล ของคุณ ไม่ใช่รูปแบบที่ฟังดูเท่ที่สุดในวิดีโอสอนออกแบบระบบ เมื่อไม่แน่ใจ ให้เลือกฐานข้อมูลเชิง

สัมพันธ์ไว้ก่อน เพราะปลอดภัยกว่า — คุณสามารถเพิ่มแคชหรือ document store ที่หลังได้เสมอ แต่การคลี่คลายโมเดลข้อมูล NoSQL ที่พันกันยุ่งเหยิงนั้นเจ็บปวดกว่ามาก

ดูแลเองหรือใช้บริการที่มีคนดูแลให้

คุณสามารถรันฐานข้อมูลเองบนเซิร์ฟเวอร์ หรือเช่าฐานข้อมูลแบบ มีคนดูแลให้ (managed) (Supabase, Neon, PlanetScale, RDS, Cloudflare D1) ที่ผู้ให้บริการจัดการเรื่องแบ็กอัพ การอัปเดตแพตช์ และความพร้อมใช้งานให้

สำหรับ vibe coding ให้เริ่มจากบริการที่มีคนดูแลให้เสมอ การดูแลฐานข้อมูลเองหมายความว่าตอนนี้คุณต้องรับผิดชอบเรื่องแบ็กอัพ การอัปเดตความปลอดภัย พื้นที่ดิสก์ และการกู้ระบบตอนตีสามด้วยตัวเอง นั่นคืองานปฏิบัติการจริง ๆ ที่จะต้องความสนใจของคุณออกจากการสร้างสิ่งที่คุณต้องการสร้าง บริการที่มีคนดูแลให้มีความง่ายเล็กน้อย แต่ประหยัดเวลาและความเสี่ยงมหาศาล ให้ขยับไปดูแลฐานข้อมูลเองก็ต่อเมื่อคุณมีเหตุผลที่ชัดเจนและมีทักษะรองรับแล้วเท่านั้น

เมื่อคุณเลือกผู้ให้บริการแบบมีคนดูแลให้ ควรรู้ไว้ว่าจริง ๆ แล้วคุณกำลังจ่ายเงินเพื่ออะไร เพราะหน้าโฆษณาการตลาดของแต่ละเจ้าดูคล้ายกันไปหมด มีบางสิ่งที่แตกต่างกันจริง ๆ ดังนี้

- รูปแบบการเชื่อมต่อ (connection model) แพลตฟอร์มแบบ serverless (Cloudflare Workers, Vercel functions) เปิดและปิดการเชื่อมต่อฐานข้อมูลตลอดเวลา ซึ่งอาจทำให้ connection pool แบบดั้งเดิมของ Postgres หดลง ผู้ให้บริการอย่าง Neon และ Supabase มี connection pooler (PgBouncer หรือเทียบเท่า) มาให้เพื่อแก้ปัญหานี้โดยเฉพาะ — ใช้ connection string แบบ pooled ในโค้ด serverless ของคุณ
- การแตกกิ่ง (branching) และแบ็กอัพ Neon และ PlanetScale ให้คุณ “แตกกิ่ง” ฐานข้อมูลได้เหมือน Git ทำให้คุณทดสอบการทำ migration บนสำเนาได้ ตรวจสอบให้แน่ใจว่าเปิดใช้งานการกู้คืนแบบ point-in-time และรู้ว่ามันย้อนกลับได้ไกลแค่ไหน
- การผูกติดกับแพลตฟอร์ม (lock-in) D1 และ DynamoDB นั้นยอดเยี่ยม แต่ผูกคุณไว้กับคลาวด์เจ้าเดียว ส่วน Postgres ธรรมดา (Supabase, Neon, RDS) เคลื่อนย้ายได้ — สคีมามีและคิวรีของคุณย้ายไปโฮสต์ Postgres ที่ไหนก็ได้ สำหรับมือใหม่ส่วนใหญ่ ตัวเลือกที่ย้ายได้จะปลอดภัยกว่าในระยะยาว

พื้นฐานการออกแบบสคีมามี

สคีมามี (schema) คือพิมพ์เขียวของข้อมูลของคุณ ว่ามีตารางอะไรบ้าง แต่ละตารางมีคอลัมน์อะไร และตารางเหล่านั้นสัมพันธ์กันอย่างไร การออกแบบสคีมามีที่ดีส่วนใหญ่เป็นเรื่องของสามัญสำนึกบวกกับนิสัยไม่กี่อย่าง

- หนึ่งตารางต่อ “สิ่งของ” หนึ่งชนิด ผู้ใช้ โพสต์ คำสั่งซื้อ — แต่ละอย่างมีตารางของตัวเอง
- ทุกแถวต้องมี ID ที่ไม่ซ้ำกัน (primary key)
- เชื่อมตารางด้วย foreign key ไม่ใช่ด้วยการคัดลอกข้อมูลไปมา
- เลือกชนิดข้อมูลที่ตรงกับความเป็นจริง เงินคือ decimal ไม่ใช่ float วันที่คือ timestamp ไม่ใช่ข้อความ
- อย่าเก็บข้อมูลซ้ำซ้อน ถ้าอีเมลของผู้ใช้อยู่ในห้าตาราง สุดท้ายคุณจะมีอีเมลห้าอันที่ไม่ตรงกัน

นี่คือสคีมามีง่าย ๆ สำหรับบล็อก แบบที่ AI จะสร้างให้คุณได้

```
CREATE TABLE users (  
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),  
  email       TEXT NOT NULL UNIQUE,  
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
CREATE TABLE posts (  
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),  
  author_id   UUID NOT NULL REFERENCES users(id),  
  title       TEXT NOT NULL,  
  body        TEXT NOT NULL,  
  published   BOOLEAN NOT NULL DEFAULT false,  
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
```

```
);
```

```
CREATE INDEX idx_posts_author ON posts(author_id);
```

บรรทัด `author_id ... REFERENCES users(id)` นั่นคือ foreign key มันเชื่อมโยงแต่ละโพสต์กลับไปยังผู้ใช้จริงเพียงคนเดียว ดังนั้นฐานข้อมูลเองจะปฏิเสธการเก็บโพสต์ที่ไม่มีเจ้าของที่ถูกต้อง



สังเกต `REFERENCES` (foreign key) ที่ผูกโพสต์เข้ากับผู้ใช้จริง) และเงื่อนไข `NOT NULL` กับ `UNIQUE` สิ่งเหล่านี้คือรั้วกันที่ฐานข้อมูลบังคับใช้ให้คุณ เพื่อไม่ให้ข้อมูลเสียแอบเล็ดลอดเข้ามาได้ แม้ว่าโค้ดของคุณจะมีบั๊กก็ตาม

เงื่อนไขที่ควรเพิ่มอย่างตั้งใจคือสิ่งที่เกิดขึ้นเมื่อฝั่ง “พ่อแม่” หายไป โดยค่าเริ่มต้น การลบผู้ใช้ที่ยังมีโพสต์อ้างอิงถึงอยู่จะล้มเหลว — ซึ่งมันจะเป็นสิ่งที่คุณต้องการอยู่แล้ว แต่คุณสามารถกำหนดให้ชัดเจนได้ `REFERENCES users(id) ON DELETE CASCADE` จะลบโพสต์ไปด้วย ในขณะที่ `ON DELETE RESTRICT` จะบล็อกการลบไว้ การเลือกด้วยความตั้งใจดีกว่าการมาค้นพบค่าเริ่มต้นแบบเจ็บ ๆ ในระบบจริง สัญชาตญาณแบบ “ให้ฐานข้อมูลปกป้องคุณ” แบบเดียวกันนี้ใช้ได้กับเงื่อนไข `CHECK` (`CHECK (price >= 0)`) — มันปฏิเสธข้อมูลที่ไม่สมเหตุผลตั้งแต่ต้นทาง แทนที่จะไว้ใจให้ทุกเส้นทางของโค้ดตรวจสอบเอง

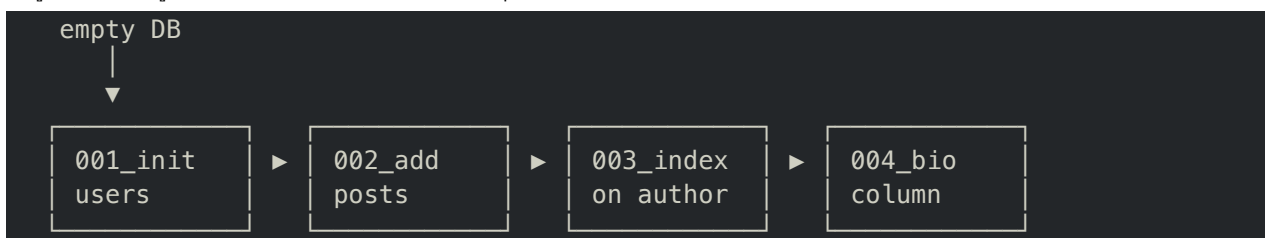
Migration: การเปลี่ยนสคีมอย่างปลอดภัย

สคีมของคุณจะเปลี่ยนแปลงไปเรื่อย ๆ — คุณจะเพิ่มคอลัมน์ เปลี่ยนชื่อตาราง เพิ่ม index ลองนึกภาพว่ามันเหมือนการติดตามการแก้ไขในเอกสารที่ใช้ร่วมกัน ทุกการเปลี่ยนแปลงถูกบันทึกไว้ตามลำดับ เพื่อให้ทุกคนอัปเดตสำเนาของตัวเองให้ตรงกันได้ด้วยวิธีเดียวกัน migration คือสคริปต์ที่มีเวอร์ชันและทำซ้ำได้ ซึ่งอธิบายการเปลี่ยนแปลงหนึ่งอย่างนั้น migration สำคัญเพราะสคีมในเครื่องของคุณ เครื่องของเพื่อนร่วมทีม และระบบจริง ต้องเหมือนกันทุกอย่าง

กฎที่ทำให้ migration ปลอดภัย

- อย่าแก้ไขฐานข้อมูลด้วยมือในระบบจริงเด็ดขาด ให้เขียน migration แทน
- Migration เดิมหน้าเสมอ แต่ละอันเป็นไฟล์ใหม่ คุณไม่เขียนไฟล์เก่าที่รันไปแล้วซ้ำ
- ทำการเปลี่ยนแปลงที่ทำลายข้อมูลด้วยความระมัดระวัง การลบคอลัมน์หรือตารางจะลบข้อมูลอย่างถาวร ต้องมีแบ็กอัพไว้ก่อน
- การเปลี่ยนแปลงแบบเพิ่มเติมปลอดภัยที่สุด การเพิ่มคอลัมน์ที่ยอมให้เป็น null ไม่ค่อยทำให้อะไรพัง แต่การเปลี่ยนชื่อหรือลบคอลัมน์อาจทำให้โค้ดที่กำลังรันอยู่พังได้

Migration ก่อตัวเป็นห่วงโซ่ตามลำดับ แต่ละอันคือขั้นตอนที่มีหมายเลขกำกับ ซึ่งพาสำเนาฐานข้อมูลทุกชุดจากสถานะที่รู้จักหนึ่งไปสู่สถานะถัดไป เพื่อให้เครื่องของคุณ เครื่องของเพื่อนร่วมทีม และระบบจริง ลงเอยเหมือนกันหมด



apply in order —————> current
never edit a step that already ran – add a NEW one

เครื่องมืออย่าง Prisma, Drizzle หรือ Rails migrations สร้างและติดตามสิ่งเหล่านี้ให้คุณโดยอัตโนมัติ ปลอ่ยให้มันทำหน้าที่ไป

migration หนึ่งอันประกอบด้วยสองส่วน คือการเปลี่ยนแปลงที่จะใช้ (up) และวิธียกเลิกมัน (down) เครื่องมือที่เชื่อถือได้จะสร้างทั้งสองส่วนให้ เพื่อให้การ deploy ที่ผิดพลาดสามารถย้อนกลับได้อย่างเรียบร้อย

```
-- up: add an optional bio, safe to apply live
ALTER TABLE users ADD COLUMN bio TEXT;

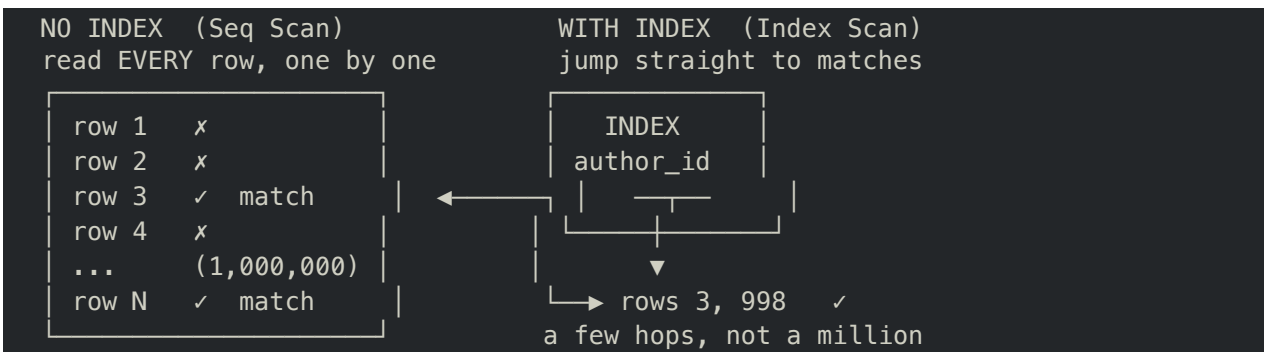
-- down: reverse it
ALTER TABLE users DROP COLUMN bio;
```

migration ที่อันตรายจริง ๆ คือการ เปลี่ยนชื่อ (rename) และ เปลี่ยนชนิดข้อมูล เพราะโค้ดเก่ากับสคิมาใหม่จะขัดแย้งกันชั่วคราว วิธีที่มีโอกาสพังใช้คือแบ่งมันออกเป็นขั้นตอนแบบเพิ่มเติม เพิ่มคอลัมน์ใหม่ เติมข้อมูลย้อนหลัง (backfill) สลับให้โค้ดอ่านคอลัมน์ใหม่ แล้วค่อยลบคอลัมน์เก่าใน deploy ครั้งถัดไป เมื่อไม่มีอะไรอ้างอิงถึงมันแล้วเท่านั้น มันดูเหมือนจะง่าย แต่มันคือความแตกต่างระหว่างการปล่อยเวอร์ชันที่ราบรื่นกับการเกิดข้อผิดพลาดหน้า 500 ถล่มหลายกับผู้ใช้ทุกคนกลางคืน deploy

การทำ Index

Index เปรียบเหมือนดัชนีท้ายเล่มหนังสือ มันทำให้ฐานข้อมูลกระโดดตรงไปยังแถวที่ตรงกันได้ทันที แทนที่จะต้องสแกนทุกแถว หากไม่มี index คิวรี่จะช้าลงเรื่อย ๆ เมื่อข้อมูลเพิ่มขึ้น — ที่ 100 แถวไม่มีปัญหาเลย แต่ที่ 1,000,000 แถวจะเจ็บปวดมาก

หากไม่มี index ฐานข้อมูลจะตรวจสอบทุกแถวตามลำดับ แต่ถ้ามี index มันจะกระโดดตรงไปยังจุดที่ตรงกันเลย เหมือนกับดัชนีในหนังสือที่พาคุณไปยังหน้าที่ถูกต้องแทนที่จะต้องอ่านตั้งแต่ต้นจนจบ



คำแนะนำเชิงปฏิบัติ

- ทำ index ให้กับคอลัมน์ที่คุณ กรอง (filter) หรือ join บ่อย ๆ (เช่น author_id, email)
- Primary key ถูกทำ index ให้อัตโนมัติอยู่แล้ว
- อย่า index ทุกอย่าง — แต่ละ index ช่วยให้การอ่านเร็วขึ้น แต่ทำให้การเขียนช้าลงและใช้พื้นที่เพิ่ม
- เพิ่ม index เมื่อคุณเห็นคิวรี่ที่ทำงานช้า ไม่ใช่เพิ่มล่วงหน้าให้ทุกคอลัมน์

วิธีที่เชื่อถือได้ในการหาว่า index ช่วยได้จริงหรือไม่ คือการถามฐานข้อมูล ไม่ใช่การเดาเอาเอง ทุกเอนจิน SQL มีคำสั่ง EXPLAIN (ใน Postgres คือ EXPLAIN ANALYZE) ซึ่งแสดงแผนการทำงานที่มันจะใช้

```
EXPLAIN ANALYZE
SELECT * FROM posts WHERE author_id = '...';
```

ถ้าผลลัพธ์แสดง Seq Scan บนตารางขนาดใหญ่ แปลว่าฐานข้อมูลกำลังอ่านทุกแถว ซึ่งเป็นสัญญาณว่า index จะช่วย

ได้ หลังจากเพิ่ม index แล้ว มันควรเปลี่ยนไปเป็น Index Scan นี่เป็นงานที่เหมาะสมมากที่จะมอบให้ AI ทำ วางคิวที่ช้าและผลลัพธ์ของ EXPLAIN ลงไป แล้วถามว่าควรเพิ่ม index ตัวไหนและเพราะอะไร คุณยังคงควบคุมสถานการณ์ได้ด้วยการเข้าใจคำตอบ ไม่ใช่ด้วยการท่องจำทฤษฎี index

การแคช

แคช (cache) เก็บคำตอบของคำถามที่มีต้นทุนสูง เพื่อที่คุณจะได้ไม่ต้องถามซ้ำอีก ถ้าหน้าแรกของคุณรันคิวรีหนักตัวเต็มซ้ำสำหรับผู้เข้าชมทุกคน ให้แคชผลลัพธ์ไว้ใน Redis เป็นเวลา 60 วินาที แล้วให้บริการทันที

การแคชเป็นการปรับแต่งประสิทธิภาพ ไม่ใช่แหล่งข้อมูลหลัก กับดักคลาสสิกของมือใหม่คือการลืม การล้างค่าแคช (cache invalidation) — สำเนาที่แคชไว้จะเก่าลง และผู้ใช้จะเห็นข้อมูลเก่า กล้วยๆ คือ แคชเฉพาะเมื่อคุณวัดแล้วว่ามีคอบวดยิ่ง และต้องมีกฎที่ชัดเจนเสมอว่าค่าที่แคชไว้จะหมดอายุหรือถูกรีเฟรชเมื่อไหร่

มีกลยุทธ์ที่สะอาดอยู่สองแบบ และการปนกันระหว่างสองแบบนี้คือสาเหตุของบั๊กแคชส่วนใหญ่ การหมดอายุตามเวลา (TTL — Time To Live トラバースวันหมดอายุบนค่าที่แคชไว้) บอกว่า “ค่านี้ใช้ได้ 60 วินาที แล้วทิ้งไป” — ง่าย และเหมาะเมื่อข้อมูลที่เก่าเล็กน้อยไม่เป็นอันตราย (จำนวนการเข้าชม พีคหน้าแรก) การล้างค่าตามเหตุการณ์ (event-based invalidation) บอกว่า “ลบค่าที่แคชไว้ทันทีที่ข้อมูลต้นทางเปลี่ยน” — ถูกต้อง แต่คุณต้องจำให้ได้ว่าต้องทำแบบนี้ใน ทุกเส้นทางที่มีการเขียนข้อมูล ซึ่งเป็นจุดที่คนมักพลาด เมื่อไม่แน่ใจ ให้เลือก TTL สั้น ๆ ไว้ก่อน เพราะค่าที่เก่าไม่เกินหนึ่งนาที่เป็นบั๊กที่เล็กกว่าค่าที่เก่าตลอดไปเพราะคุณลืมล้างค่าแคชจุดหนึ่ง

คิว (Queue)

คิว (queue) ให้คุณพูดว่า “ทำสิ่งนี้ทีหลัง” แทนที่จะพูดว่า “ทำสิ่งนี้เดี๋ยวนี้” เมื่อผู้ใช้สมัครสมาชิก คุณไม่อยากให้เขาต้องรอในขณะที่คุณส่งอีเมลต้อนรับและปรับขนาดรูปโปรไฟล์ของเขา ให้ส่งงานเหล่านั้นเข้าคิว (RabbitMQ, BullMQ, Cloudflare Queues, SQS) แล้วให้ worker เบื้องหลังคอยหยิบไปทำ ผู้ใช้จะได้รับการตอบสนองทันที

ใช้คิวสำหรับงานใด ๆ ที่ทำงานช้า สามารถล้มเหลวและลองใหม่ได้ หรือไม่จำเป็นต้องเสร็จก่อนที่คุณจะตอบสนองผู้ใช้

กับดักที่มือใหม่ส่วนใหญ่เจอคืองานที่เข้าคิวอาจถูกรันได้ มากกว่าหนึ่งครั้ง — เครือข่ายสะดุดทำให้ worker ลองใหม่ และตอนนี้อีเมลต้อนรับก็ถูกส่งไปสองครั้ง วิธีแก้คือทำให้งาน idempotent (ไม่ว่าจะรันกี่ครั้งก็ให้ผลลัพธ์เดียวกัน) ออกแบบให้การรันงานเดิมซ้ำสองครั้งมีผลเหมือนกับรันแค่ครั้งเดียว (ตรวจสอบว่า “ฉันส่งสิ่งนี้ไปแล้วหรือยัง” ก่อนที่จะส่ง) นอกจากนี้ ควรตัดสินใจล่วงหน้าด้วยว่าจะเกิดอะไรขึ้นกับงานที่ล้มเหลวซ้ำ ๆ ไม่หยุด คิวที่มีความเป็นผู้นำจะส่งงานนั้นไปยัง dead-letter queue เพื่อไม่ให้ข้อความที่มีปัญหาหนึ่งอันไปปิดกั้นงานอื่น ๆ ที่ต่อคิวอยู่ข้างหลัง

ข้อผิดพลาดทั่วไปของมือใหม่

- หันไปใช้ NoSQL เพราะฟังดูทันสมัย แอปส่วนใหญ่มีความสัมพันธ์ระหว่างข้อมูล SQL จัดการเรื่องนี้ได้ดีกว่า
- เก็บเงินเป็นเลขทศนิยมแบบ floating-point 0.1 + 0.2 ไม่เท่ากับ 0.3 อย่างมีชื่อเสียง ให้ใช้ชนิดข้อมูล decimal แทน
- ไม่มีแบ็กอัพ บริการที่มีคนดูแลให้มักแบ็กอัพให้อัตโนมัติ — ตรวจสอบให้แน่ใจ อย่าสันนิษฐานเอาเอง
- เก็บข้อมูลลับหรือไฟล์ขนาดใหญ่ไว้ในฐานข้อมูล ไฟล์ควรอยู่ใน object storage (S3, R2) ฐานข้อมูลเก็บแค่ ลิงก์
- ปลอมให้ AI เขียน คิวรี ดิบ ที่เอาข้อมูลจากผู้ใช้ มาต่อรวมกันตรง ๆ สิ่งนี้ทำให้เกิด SQL injection ให้ใช้ parameterized queries แทน
- คิวรีแบบ N+1 ลูป (loop) ที่รันคิวรีหนึ่งครั้งต่อรายการหนึ่งชิ้น (ดึงโพสต์ 100 อัน แล้วค้นหาผู้เขียนแยกอีก 100 ครั้ง) ถล่มฐานข้อมูลอย่างหนัก ให้ดึงข้อมูลที่เกี่ยวข้องด้วยคิวรีเดียวโดยใช้ join หรือรวมการค้นหาเป็นกลุ่ม
- ไม่เคยทดสอบภายใต้โหลด คิวรีที่รันเร็วทันทีบน 50 แถวในเครื่องคุณ อาจคลานช้าบนระบบจริงที่มี 5 ล้านแถว ให้เตรียมข้อมูลปลอมจำนวนสมจริงก่อนที่คุณจะเชื่อผลของประสิทธิภาพ

สั่งให้ AI สร้างชั้นข้อมูลของคุณ

AI เก่งมากในการออกแบบสคีมาและเขียนคิวรี — ถ้า คุณให้ข้อจำกัดที่มันควรออกแบบตาม อย่าพูดแค่ “สร้างฐานข้อมูลให้หน่อย” ให้อธิบายโดเมนของปัญหา ความสัมพันธ์ และสิ่งที่ต้องรองรับไม่ได้ของคุณ

พรมแดนที่ดีมีลักษณะแบบนี้

Design a PostgreSQL schema for a task-management app.

Entities and relationships:

- A user can own many projects.
- A project has many tasks.
- A task belongs to one project, has a status (todo / doing / done), and an optional due date.

Requirements:

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL, plus a forward migration file.
- Explain any trade-offs you made.

จากนั้น ตรวจสอบผลลัพธ์ ด้วยเช็คลิสต์จากบทนี้ ชนิดข้อมูลตรงกับความเป็นจริง มี foreign key และเงื่อนไขครบถ้วน มี index ที่สมเหตุสมผล ไม่มีข้อมูลซ้ำซ้อน เมื่อคุณขอคิวรี ให้ระบุอย่างชัดเจนว่าต้องเป็น parameterized queries เพื่อไม่ให้ข้อมูลจากผู้ใช้ถูกต่อรวมเข้ากับ SQL โดยตรง และสำหรับ migration ใดก็ตามที่ลบหรือเปลี่ยนชื่อบางอย่าง ให้ขอให้ AI ชี้ให้เห็นชัดเจน เพื่อให้คุณแบ็กอัปก่อนได้

ฐานข้อมูลเป็นส่วนของแอปที่ความผิดพลาดย้อนกลับได้ยากที่สุด — ข้อมูลเมื่อเสียหายหรือหายไปแล้ว มักไม่ได้กลับคืนมาเสมอไป นี่คือเหตุผลว่าทำไมพื้นฐานเหล่านี้จึงคุ้มค่าที่จะรู้ ไม่ใช่เพื่อให้คุณเขียนโค้ดทุกบรรทัดด้วยตัวเอง แต่เพื่อให้คุณสั่ง AI ได้อย่างมั่นใจ และจับความเคลื่อนไหวที่อันตรายได้ก่อนที่มันจะถูกปล่อยออกไป

สรุปและแบบฝึกหัด

ประเด็นสำคัญที่ควรจำ

- แอปส่วนใหญ่ต้องการเพียงหนึ่งในสามรูปแบบการจัดเก็บข้อมูล — เชิงสัมพันธ์ เอกสาร หรือคีย์-ค่า — และเชิงสัมพันธ์คือค่าเริ่มต้นที่ปลอดภัย
- เริ่มด้วยฐานข้อมูลที่มีคนดูแลให้ คุณยังคงเป็นเจ้าของข้อมูลและตัดสินใจเรื่องสคีมา ส่วนแพลตฟอร์มดูแลเรื่องเซิร์ฟเวอร์ให้
- การออกแบบสคีมาที่ดีหมายถึงชนิดข้อมูลที่ตรงกับความเป็นจริง มี foreign key และเงื่อนไขครบถ้วน มี index ที่สมเหตุสมผล และไม่มีข้อมูลซ้ำซ้อน
- ปฏิบัติกับทุก migration ว่าเป็นประตูดังเดียว แบ็กอัปก่อนทำสิ่งใดที่ลบหรือเปลี่ยนชื่อ และตรวจสอบ migration ที่ AI เขียนอย่างละเอียด
- เรียกร้องให้ใช้ parameterized queries เสมอ เพื่อไม่ให้ข้อมูลจากผู้ใช้ถูกต่อรวมเข้ากับ SQL โดยตรง

ลองทำดู

เลือกไอเดียแอปเล็ก ๆ สักตัว แล้วเขียนเอนทิตี (entities) และความสัมพันธ์ของมันเป็นภาษารวมตาก่อน จากนั้นส่งคำอธิบายนั้นให้ AI แล้วขอสคีมาพร้อมไฟล์ migration แบบเดินทาง — และตรวจสอบผลลัพธ์เทียบกับเช็คลิสต์ของบทนี้ ก่อนที่จะรันอะไรทั้งสิ้น

Design a relational schema for a personal bookmarks app.

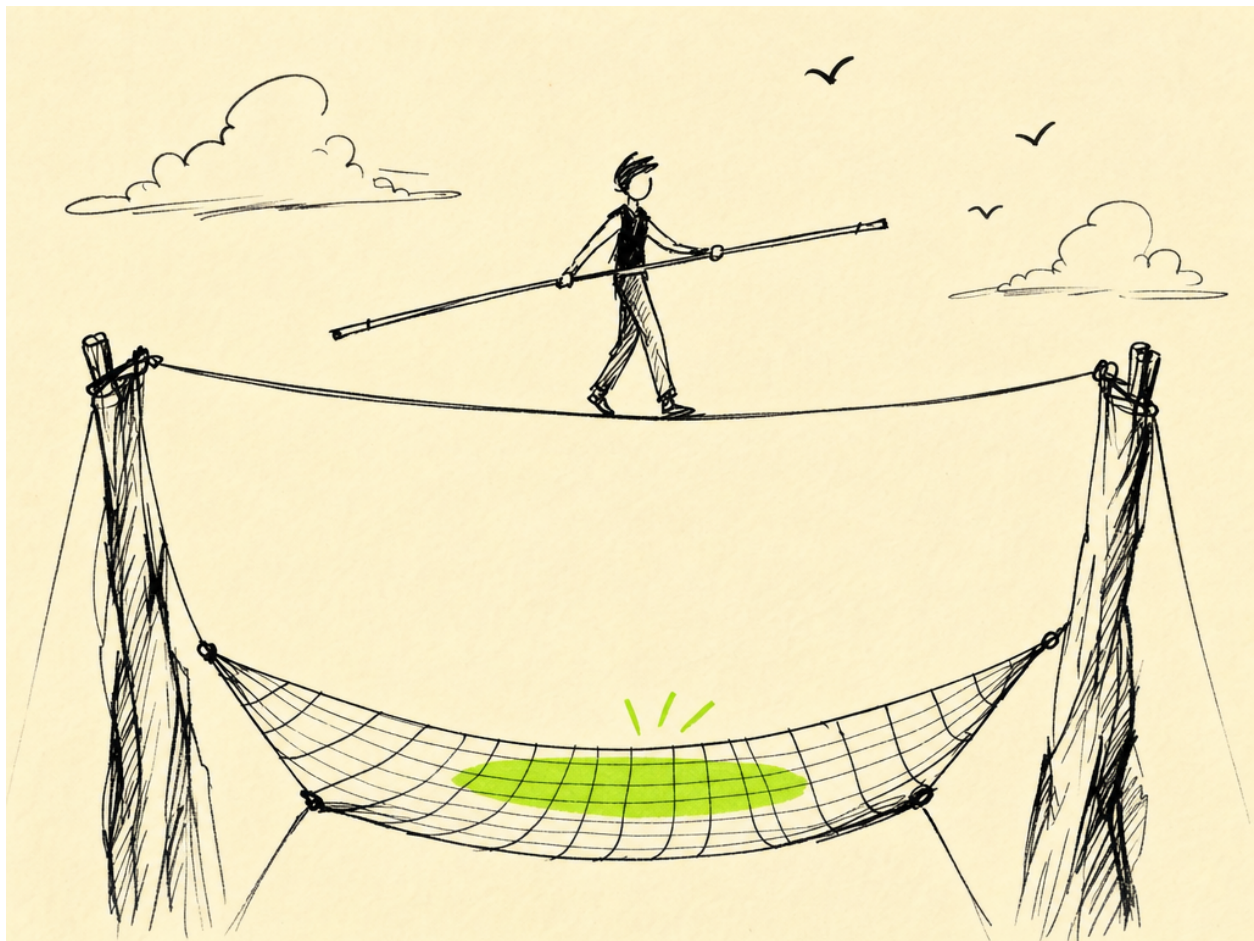
Entities and relationships:

- A user has many bookmarks.
- A bookmark belongs to one user, has a URL, a title, and many tags.
- A tag can apply to many bookmarks.

Requirements:

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL plus a forward migration file.
- Flag any migration step that drops or renames so I can back up first.

รู้กันความปลอดภัย การทดสอบ และการรีวิว



Vibe coding ทำให้คุณทำงานได้เร็ว อันตรายก็คือคุณอาจเร็วไปในทิศทางที่ผิดด้วย — และเพราะ AI ฟังดูมั่นใจเสมอ คุณอาจไม่ทันสังเกตจนกว่าบางอย่างจะพังต่อหน้าผู้ใช้จริง รู้กันความปลอดภัย (guardrails) คือสิ่งที่ทำให้คุณรักษาความเร็วไว้ได้โดยไม่ต้องแลกกับความพึง

บทนี้ไม่ได้ต้องการเปลี่ยนโปรเจกต์ของคุณให้กลายเป็นระบบราชการที่อึดอัด แต่เป็นเรื่องของการตรวจสอบอัตโนมัติ ราคาถูกไม่กี่ยาง ที่คอยจับความผิดพลาดของ AI (และของคุณเอง) ก่อนที่มันจะไปถึงระบบจริง ตั้งค่าไว้ครั้งเดียว แล้วมันจะทำงานอยู่เบื้องหลังตลอดไป ปรึษาทั้งหมดของบทนี้สรุปได้ในบรรทัดเดียวว่า อย่าส่งมอบสิ่งที่คุณไม่เข้าใจ รู้กันความปลอดภัยคือสิ่งที่ทำให้ “การเข้าใจมัน” มีต้นทุนต่ำพอที่คุณจะทำมันจริง ๆ ทุกครั้ง

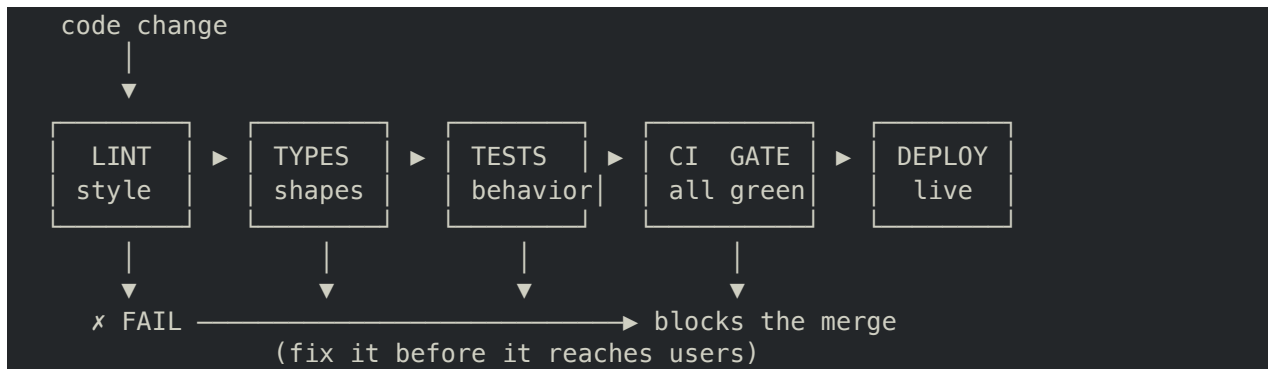
ทำไมคุณถึงต้องมีร้วกันความปลอดภัยเมื่อสั่งงาน AI

AI ไม่รู้ว่าแอปของคุณ ควรจะ ทำงานอย่างไร มันเดารูปแบบจากสิ่งที่ดูสมเหตุสมผล นั้นหมายความว่ามันพร้อมจะ:

- ลบเช็คที่มันไม่เข้าใจทิ้งไป “เพื่อแก้ error”
- รีแฟกเตอร์สามไฟล์ ทั้งที่คุณขอให้แก้แค่ไฟล์เดียว
- แอบสร้างบั๊กที่เนียนจนเส้นทางการใช้งานปกติ (happy path) ไม่เผยให้เห็น
- บอกคุณอย่างมั่นใจว่าเสร็จแล้ว ทั้งที่ไม่เคยรันโค้ดจริง

ร้วกันความปลอดภัยเปลี่ยน “เชื่อผมสิ” ให้กลายเป็น “พิสูจน์มาสิ” เป้าหมายคือระบบที่ ยาก ที่จะทำอะไรพังโดยไม่ตั้งใจ เพื่อให้คุณยังพูดได้ว่า “ลองเลย” โดยไม่ต้องกลั่นหายใจรอ

ลองนึกภาพร้วกันความปลอดภัยเป็นเหมือนท่อลำเลียงที่ทุกการเปลี่ยนแปลงต้องผ่านก่อนจะไปถึงผู้ใช้จริง แต่ละด่านราคาถูกและทำงานอัตโนมัติ แค่ด่านเดียวก็หยุดการเปลี่ยนแปลงที่แย่ได้แล้ว:

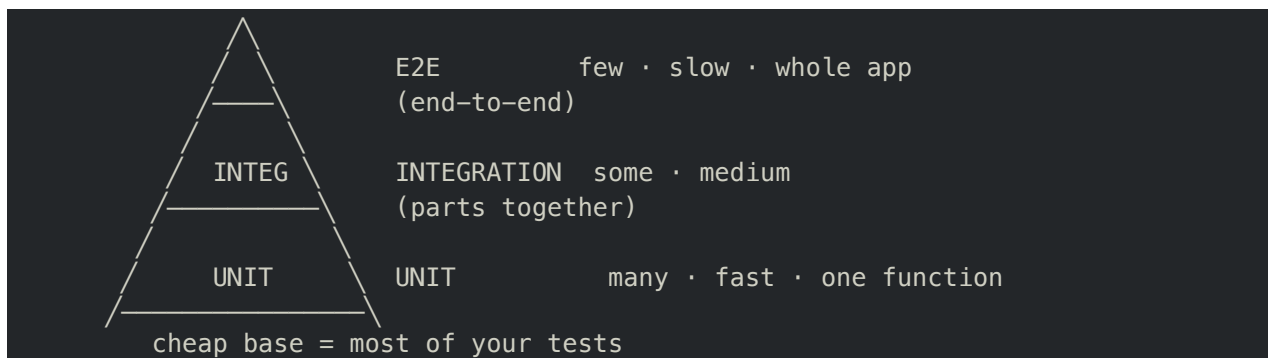


มีเหตุผลที่ลึกกว่านั้นว่าทำไมเรื่องนี้ถึงสำคัญกับ AI มากกว่ากับเพื่อนร่วมทีมที่เป็นมนุษย์ มนุษย์ที่ลบการตรวจสอบความถูกต้อง (validation check) ที่มักจะรู้ตัวว่ากำลังเสี่ยงอะไรอยู่ และรู้สึกถึงน้ำหนักของมัน ส่วน AI ไม่รู้สึกอะไรเลย มันจะลบการ์ด เขียนไฟล์ใหม่ และรายงานว่าสำเร็จด้วยน้ำเสียงร่าเริงเหมือนเดิม ไม่ว่าการเปลี่ยนแปลงนั้นจะยอดเยี่ยมหรือหายนะ คุณอ่านภาษากายของมันไม่ได้ เพราะมันไม่มีภาษากาย ร้วกันความปลอดภัย คือ ภาษากายของมัน — เป็นสัญญาณที่ซื้อสัตย์เพียงอย่างเดียวที่คุณจะได้รู้ว่ามันแท้จริงแล้วดีหรือไม่

การทดสอบ: ตาข่ายนิรภัยของคุณ

การทดสอบ (test) คือโค้ดที่ตรวจสอบว่าโค้ดอื่นทำงานตามที่คุณคาดหวังหรือไม่ เมื่อเขียนไว้แล้ว มันจะรันภายในไม่กี่วินาที และบอกคุณทันทีถ้ามีอะไรตกอวย (regress)

ชุดทดสอบที่ดีจะมีลักษณะเหมือนพีระมิด: ฐานเป็นการทดสอบระดับหน่วย (unit test) เล็ก ๆ เร็ว ๆ จำนวนมาก ตรงกลางเป็นการทดสอบเชิงบูรณาการ (integration test) จำนวนน้อยลง และบนยอดเป็นการทดสอบแบบครบวงจร (end-to-end test) ที่ช้าและมีจำนวนน้อยที่สุด ความปลอดภัยส่วนใหญ่ของคุณมาจากฐานที่ราคาถูก:



คุณไม่จำเป็นต้องมี coverage 100% เริ่มจากการทดสอบสิ่งที่จะสร้างความเสียหายมากที่สุดถ้ามันพัง: การล็อกอิน

การชำระเงิน ข้อมูลที่สร้างขึ้นใหม่ไม่ได้ และตรรกะธุรกิจหลัก AI เก่งมากในการเขียนเทสต์ — บางทีเก่งกว่าการเขียนฟิเจอร์ด้วยซ้ำ เพราะพฤติกรรมที่คาดหวังนั้นชัดเจนเป็นรูปธรรม

นี่คือพรมบิตัวอย่างที่ใช้ได้ผลดี:

Write tests for the `calculateDiscount` function in `src/pricing.js`.
Cover: a normal case, the zero/empty case, the maximum-discount boundary, and one invalid input. Use the existing test setup in this repo (check how other `*.test.js` files are structured).
Run the tests and show me they pass before you finish.

บรรทัดสุดท้ายนั้นสำคัญ ให้ AI รัน เทสต์เสมอ ไม่ใช่แค่เขียนมันขึ้นมา เทสต์ที่ไม่เคยถูกรันคือการเดา และก็คุ้มค่าที่จะระบุให้ชัดเจนว่ากรณีไหนที่ต้องครอบคลุม เพราะถ้าปล่อยให้ AI ตัดสินใจเอง มันมักจะเขียนสามแบบของเส้นทางการใช้งานปกติแล้วเรียกว่าครอบคลุมแล้ว กรณีที่จับบักได้จริงคือกรณีน่าเบื่อ ๆ: อินพุตว่างเปล่า ค่าขอบเขต สิ่งที่ต้องโยนข้อผิดพลาด (throw)

ตัวอย่างง่าย ๆ ของสิ่งที่มันควรผลิตออกมา:

```
import { calculateDiscount } from "../pricing.js";
import { describe, it, expect } from "vitest";

describe("calculateDiscount", () => {
  it("applies a normal percentage discount", () => {
    expect(calculateDiscount(100, 0.2)).toBe(80);
  });

  it("returns the full price when discount is zero", () => {
    expect(calculateDiscount(100, 0)).toBe(100);
  });

  it("never discounts below zero", () => {
    expect(calculateDiscount(100, 1.5)).toBe(0);
  });

  it("rejects a negative price", () => {
    expect(() => calculateDiscount(-10, 0.2)).toThrow();
  });
});
```

เมื่อคุณเปลี่ยนฟิเจอร์ในภายหลัง ให้รันเทสต์ก่อน ถ้ามันขึ้นสีแดง แปลว่าคุณเจอจุดพังก่อนที่ผู้ใช้ของคุณจะเจอ เมื่อ AI แก้มัก ให้ขอให้มันเพิ่มเทสต์ที่จะล้มเหลวกับพฤติกรรมเดิม — เพื่อให้บักนั้นไม่สามารถแอบกลับมาได้อีก นิสัยนี้มีชื่อเรียกว่า regression test (เทสต์กันบักย้อนกลับ) บักทุกตัวที่คุณเจอคือบทเรียนที่โค้ดเบสได้จ่ายเงินซื้อมาแล้ว การมี regression test คือวิธีที่คุณเก็บรักษาบทเรียนนั้นไว้

ข้อควรระวังหนึ่งข้อ: เทสต์เองก็ผิดได้เหมือนกัน ถ้า AI เขียนเทสต์ที่ยืนยันว่าพฤติกรรม ที่มีบัก นั้นถูกต้อง มันจะผ่านตลอดไปในขณะที่แอปของคุณยังพังอยู่ ดังนั้นให้อ่านเงื่อนไขการยืนยัน (assertion) ไม่ใช่แค่เครื่องหมายถูกสีเขียว เทสต์ที่ผ่านหมายความว่า “โค้ดทำในสิ่งที่เทสต์บอก” — คุณยังต้องยืนยันว่าเทสต์นั้นบอกในสิ่งที่ถูกต้อง

การตรวจสอบชนิดข้อมูลและ linting: จับความผิดพลาดได้ฟรี

มีเครื่องมือสองตัวที่จับความผิดพลาดทั้งหมดมาได้โดยแทบไม่ต้องออกแรงเลย:

- การตรวจสอบชนิดข้อมูล (type checking) (TypeScript, mypy ฯลฯ) จับกรณี “คุณส่งสตริงเข้าไปในที่ที่ควรเป็นตัวเลข” และ “ค่านี้อาจเป็น undefined ได้” AI ชอบลิมกรณีขอบเขต (edge case) ที่ชนิดข้อมูลทำให้เป็นไปไม่ได้อยู่แล้ว

- Linting (ESLint, Ruff ฯลฯ) จับตัวแปรที่ไม่ได้ใช้ โค้ดที่ไม่ไปถึง (unreachable code) และความเบี่ยงเบนของสไตล์ มันช่วยไม่ให้โค้ดเบสที่ AI แตะต้องค่อย ๆ กลายเป็นโຈจ

ตั้งค่าเครื่องมือเหล่านี้ตั้งแต่เนิ่น ๆ และบอก AI ให้ปฏิบัติตาม: “รันตัวตรวจสอบชนิดข้อมูลและ linter แล้วแก้ทุกอย่างที่มันรายงาน ก่อนที่จะบอกว่าเสร็จ” ให้ถือว่าการตรวจสอบชนิดข้อมูลที่สะอาดเป็นส่วนหนึ่งของนิยามคำว่า “เสร็จแล้ว”

แต่ให้ระวังทางลัดที่เกียจ เมื่อตัวตรวจสอบชนิดข้อมูลบ่น วิธีแก้ที่ถูกต้องจริง ๆ คือจัดการกับกรณีที่มีน้จ — ค่า undefined ชนิดข้อมูลที่ผิด ส่วนวิธีแก้ แบบเร็ว คือปิดปากคำบ่นด้วย `// @ts-ignore`, `any`, หรือคอมเมนต์ `# type: ignore` และ AI มักจะหยิบวิธีนี้มาใช้บ่อยมาก เพราะมันทำให้ข้อความสีแดงหายไป มันยังทำให้การตรวจสอบนั้นไร้ประโยชน์อย่างตรงจุดที่มันควรจะช่วยพอดิ ถ้าคุณเห็นคอมเมนต์ปิดการเตือนโผล่ขึ้นมาใน diff ให้ถือว่ามันเป็นคำถาม ไม่ใช่คำตอบ: “ทำไมถึงต้องมีสิ่งนี้ ให้แก้ที่ชนิดข้อมูลจริง ๆ แทนที่จะเมินเฉยมัน”

จำกัดขอบเขตการเปลี่ยนแปลงของ AI

สาเหตุใหญ่ที่สุดของความเสียหายที่เกิดจาก AI คือการปล่อยให้มันเปลี่ยนแปลงมากเกินไปในคราวเดียว diff ยาว 400 บรรทัดครอบคลุมเก้าไฟล์นั้นแทบเป็นไปไม่ได้ที่จะรีวิว และเป็นที่ที่ซ่อนบั๊กได้ง่ายมาก

จำกัดการเปลี่ยนแปลงให้เล็กและมีขอบเขตชัดเจน:

- ขอให้ทำ การเปลี่ยนแปลงเชิงตรรกะเดียวในแต่ละครั้ง ไม่ใช่ “ในเมื่ออยู่ตรงนั้นแล้ว ช่วย... ด้วย”
- บอกให้ชัดเจน: “แต่แค่ `src/auth.js` เท่านั้น อย่าไปรีแฟกเตอร์อย่างอื่น”
- คอมมิตสถานะที่ใช้งานได้บ่อย ๆ เพื่อให้คุณมีจุดที่สะอาดสำหรับย้อนกลับเสมอ
- เมื่อมันเสนอการเขียนใหม่ครั้งใหญ่ ให้ขอให้มันแบ่งงานออกเป็นขั้นตอนที่คุณรีวิวได้ที่ละขั้น

ขอบเขตที่เล็กทำให้ขั้นตอนถัดไป — การอ่าน diff — เป็นไปได้จริง มีผลพวงที่ทบตันกันอยู่ตรงนี้: คอมมิตเล็ก ๆ ก็คือ รัศมีความเสียหายที่เล็กตามไปด้วย ถ้าการเปลี่ยนแปลงที่มีขอบเขตแคบกลายเป็นผิด คุณแค่ revert หนึ่งคอมมิตแล้วเสียเวลาสิบนาที แต่ถ้าการเปลี่ยนแปลงที่กระจายไปทั่วผิดพลาด คุณจะต้องเสียเวลาหนึ่งชั่วโมงคลี่คลายว่าส่วนไหนทำให้พัง หรือไม่ก็ต้องทิ้งงานทั้งปายไป การควบคุมขอบเขตเป็นประกันภัยราคาถูกที่คุณซื้อไว้ก่อนที่คุณจะรู้ว่าต้องใช้มัน

อ่าน diff อย่างมีวิจารณ์ญาณ

diff คือรายการที่บอกอย่างชัดเจนว่าอะไรเปลี่ยนไป: บรรทัดสีแดงคือถูกลบ บรรทัดสีเขียวคือถูกเพิ่ม การอ่าน diff คือนิสัยที่ให้ผลตอบแทนสูงที่สุดใน vibe coding เพราะมันคือจุดที่คุณจับได้ว่า AI กำลังทำสิ่งที่คุณไม่ได้ขอ

คุณไม่จำเป็นต้องเข้าใจทุกบรรทัด แต่ให้สแกนหาสัญญาณเตือนเหล่านี้:

- ไฟล์ที่คุณไม่คาดคิดว่าจะถูกเปลี่ยน ทำไมไฟล์ config ถึงถูกแตะต้อง?
- โค้ดที่ถูกลบ มันลบการตรวจสอบความถูกต้อง ตัวจัดการข้อผิดพลาด หรือเทสต์ “เพื่อให้มันผ่าน” หรือเปล่า?
- ความลับหรือค่าที่ฝังตายตัว คีย์ API (Application Programming Interface — อินเทอร์เฟซโปรแกรมประยุกต์) รหัสผ่าน หรือ URL (Uniform Resource Locator — ที่อยู่เว็บ) ที่ไม่ควรอยู่ในโค้ด
- การตรวจสอบที่ถูกปิดใช้งาน เทสต์ที่ถูกข้าม การ์ดที่ถูกคอมเมนต์ทิ้ง `// @ts-ignore` หรือสิทธิ์ที่ถูกผ่อนปรน
- ขอบเขตที่บานปลาย การเปลี่ยนแปลงที่ห่างไกลจากสิ่งที่คุณขอ

ถ้ามีอะไรดูแปลก ๆ ให้ถามว่า: “อธิบายว่าทำไมถึงเปลี่ยน X — ฉันไม่ได้ขอแบบนั้น” ให้ AI อธิบายเหตุผลของการลปเป็นพิเศษ “มันทำให้เกิด error” คือเหตุผลที่ควรเข้าใจตัว error นั้น ไม่ใช่เหตุผลที่จะลบการตรวจสอบทิ้ง

วิธีปฏิบัติที่ช่วยสร้างนิสัยนี้ได้จริง: อ่าน diff ก่อน ที่คุณจะยอมรับการเปลี่ยนแปลง ไม่ใช่หลังจากนั้น เมื่อโค้ดถูก merge ไปแล้วและแอปดูเหมือนจะทำงานได้ คุณจะไม่มีวันย้อนกลับไปอ่านมันอีก — เพราะไม่มีแรงเสียดทานใด ๆ บังคับให้คุณทำ ช่วงเวลาของการรีวิวต้องเป็นช่วงเวลาเดียวกับการตัดสินใจ เครื่องมือช่วยได้ตรงนี้: `git diff --stat` ให้สรุปหนึ่งบรรทัดต่อไฟล์ ทำให้ไฟล์ที่ไม่คาดคิดโดดเด่นออกมาทันที และการรีวิวบนหน้า pull request จะแสดงการเปลี่ยนแปลงทั้งหมดในมุมมองเดียวที่เลื่อนดูได้ แทนที่จะกระจัดกระจายอยู่ในเอดิเตอร์ของคุณ

ด้านรีวิว: เช็คลิสต์ก่อนการ merge

ด้านรีวิว (review gate) คือรายการสั้น ๆ ของสิ่งที่ต้องเป็นจริงก่อนที่การเปลี่ยนแปลงจะถูกนำไปใช้จริง ให้คุณทำเอง หรือให้ AI ยืนยันแต่ละข้อ:

- ☐ ฉันอ่าน diff แล้วและเข้าใจว่าอะไรเปลี่ยนไปและทำไม
- ☐ การเปลี่ยนแปลงมีขอบเขตตรงกับสิ่งที่ฉันขอจริง ๆ
- ☐ เทสต์ผ่านทั้งหมด (และพฤติกรรมใหม่มีเทสต์รองรับ)
- ☐ การตรวจสอบชนิดข้อมูลและ linter สะอาด
- ☐ ไม่มีความลับ log สำหรับดีบัก หรือโค้ดที่ถูกคอมเมนต์ทิ้งค้างอยู่
- ☐ ไม่มีการตรวจสอบความปลอดภัยใดถูกลบออกไปอย่างเจียบ ๆ
- ☐ ฉันได้ลองใช้งานการเปลี่ยนแปลงนี้เองแล้ว อย่างน้อยก็เส้นทางการใช้งานปกติ

ถ้าคุณตึกช่องไหนไม่ได้ แปลว่ายังไม่เสร็จ

เช็คลิสต์นี้จึงใจทำให้สั้น เพราะด้านที่คุณข้ามไปก็ไม่ต่างจากไม่มีด้านเลย สิ่งที่น่าสนใจ โดยเฉพาะตอนที่คุณกำลังเร่งความเร็วและการเปลี่ยนแปลง “ดูเหมือนจะโอเค” คือการปล่อยผ่านไปเฉย ๆ จงต้านทานสิ่งนั้นเมื่อเป็นเรื่องที่ผู้ใช้เห็นได้ เทคนิคที่มีประโยชน์คือให้ AI กรอกเช็คลิสต์แทนคุณ — วางรายการแล้วถามว่า: “ตรวจเช็คลิสต์รีวิวนี้นี้สำหรับการเปลี่ยนแปลงที่คุณเพิ่งทำ สำหรับแต่ละข้อ บอกว่า PASS หรือ FAIL พร้อมหลักฐานหนึ่งบรรทัด” มันไม่สามารถเขียนอย่างซื่อสัตย์ว่า “เทสต์ผ่าน” ได้โดยไม่ได้รับมันจริง ๆ และการบังคับให้มันอ้างหลักฐานในแต่ละบรรทัดจะเปลี่ยนคำว่า “ดูโอเค” ที่คลุมเครือ ให้กลายเป็นสิ่งที่ตรวจสอบได้

พื้นฐานของ CI: ทำให้การตรวจสอบเป็นแบบอัตโนมัติ

CI (Continuous Integration — การผสานรวมต่อเนื่อง) คือหุ่นยนต์ที่รันการตรวจสอบของคุณทุกครั้งที่คุณ push โค้ด เพื่อไม่ให้คุณ ลืม รันมัน แพลตฟอร์มส่วนใหญ่ทำให้เรื่องนี้เหลือแค่ไฟล์เดียว นี่คือตัวอย่าง GitHub Actions แบบขั้นต่ำ:

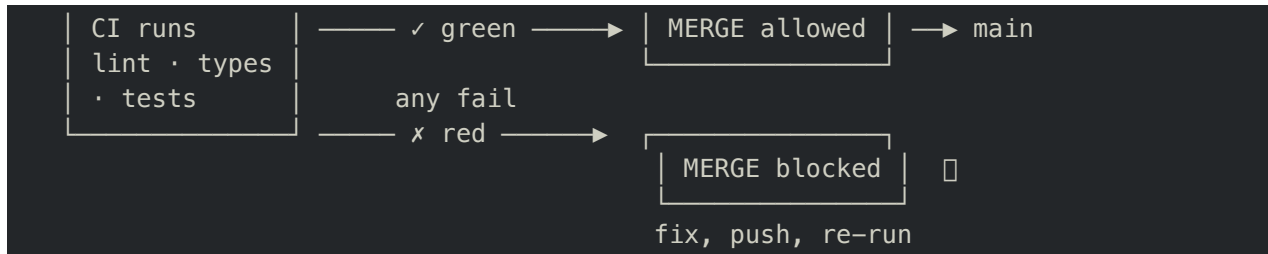
```
# .github/workflows/ci.yml
name: CI
on: [push, pull_request]

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci
      - run: npm run lint
      - run: npm run typecheck
      - run: npm test
```

ตอนนี้ทุก push จะถูก lint ตรวจสอบชนิดข้อมูล และทดสอบโดยอัตโนมัติ ถ้ามีอะไรล้มเหลว คุณจะเห็นเครื่องหมายสีแดง ก่อนที่โค้ดจะถูก merge ได้ คุณสามารถขอให้ AI เขียนไฟล์นี้ให้เข้ากับสแต็กของคุณได้ด้วย — “เพิ่ม GitHub Actions workflow ที่รัน lint, การตรวจสอบชนิดข้อมูล และเทสต์ของเราทุกครั้งที่มีการ pull request”

เมื่อเปิดใช้งานการป้องกัน branch (branch protection) แล้ว การตรวจสอบที่ขึ้นสีแดงจะไม่ใช่แค่การเตือนคุณเฉย ๆ — มันจะล็อกปม merge ไว้จริง ๆ จนกว่า build จะเขียวอีกครั้ง:





จุดสำคัญของ CI ไม่ใช่การทำให้เป็นอัตโนมัติเพื่อตัวมันเอง แต่คือการย้ายการตรวจสอบออกจากความทรงจำของคุณ ไปอยู่ในท่อลำเลียงที่ไม่สามารถถูกข้ามได้แม้ในคืนที่คุณเหนื่อยล้า มีการอัปเดตสองอย่างที่คุ้มค่าเมื่อพื้นฐานเริ่มเขียวแล้ว อย่างแรก เปลี่ยนการตรวจสอบให้เป็นสถานะ บังคับ (required) บน branch หลักของคุณ (กฎ “branch protection rule” บน GitHub) เพื่อให้ build สีแดงล็อกปุ่ม merge จริง ๆ แทนที่จะแค่แสดงคำเตือนที่คุณเพิกเฉยได้ อย่างที่สอง เมื่อ CI ล้มเหลว ให้วาง log ที่ล้มเหลวไปให้ AI โดยตรง: “CI ล้มเหลวในรอบนี้ นี่คือ output — วินิจฉัยและแก้ไขให้หน่อย” log คือรายงานบั๊กที่สร้างโดยเครื่องมืออย่างแม่นยำ ซึ่งเป็นอินพุตแบบที่ AI ทำได้ดีที่สุดพอดี

นิสัยที่ปลอดภัยโดยค่าเริ่มต้น

รวบรวมทุกอย่างเข้าเป็นปฏิริยาตอบสนองที่ทำให้คุณเคลื่อนที่เร็วได้โดยไม่ต้องกลัว:

- ทำงานบน branch ไม่ใช่บนโปรดักชันโดยตรง branch นั้นไม่มีต้นทุน แต่โปรดักชันที่พังนั้นมีต้นทุน
- คอมมิตสถานะที่ใช้งานได้เล็ก ๆ บ่อย ๆ การคอมมิตบ่อย ๆ ทำให้การย้อนกลับเป็นเรื่องเล็กน้อย
- ขอให้ AI รันสิ่งที่มันเขียนเสมอ — เทสต์ ตัวแอป ตัวตรวจสอบชนิดข้อมูล
- เปิดเทอร์มินัลไว้หนึ่งช่องสำหรับ AI และอีกหนึ่งช่องสำหรับตัวคุณเอง เพื่อให้คุณตรวจสอบได้อย่างอิสระ
- อย่าปล่อยให้การเปลี่ยนแปลงที่คุณไม่ได้อ่านถูกส่งออกไป ความเร็วมาจากความรู้กันความปลอดภัยที่ดี ไม่ใช่จากการข้ามการรีวิว

รู้กันความปลอดภัยไม่ได้ทำให้คุณช้าลง — มันคือสิ่งที่ทำให้คุณพูดได้ว่า “เอา ส่งเลย” โดยไม่รู้สึกลังเลในท้อง ตั้งค่ามันไว้ครั้งเดียว แล้ว AI จะวิ่งได้เต็มที่ในขณะที่ตาข่ายนิรภัยคอยรองรับการล้ม

เมื่อรู้กันความปลอดภัยกริดรอง: คัดแยกก่อนแก้

รู้กันความปลอดภัยจะมีประโยชน์ก็ต่อเมื่อคุณตอบสนองต่อมันอย่างถูกต้อง และรูปแบบความล้มเหลวก็คาดเดาได้: build ขึ้นสีแดง คุณตื่นตระหนกเล็กน้อย แล้ววาง error ไปให้ AI พร้อมคำว่า “แก้ที” บางครั้งมันก็ได้ผล แต่บ่อยครั้งมันสร้างการแก้ไขที่ทำให้ข้อความสีแดงหายไป โดยไม่ได้แก้ปัญหานั้นแท้จริง — AI อาจทำให้เทสต์อ่อนลง ขยายชนิดข้อมูลให้กว้างขึ้น หรือห่อการเรียกที่ล้มเหลวไว้ใน try/catch ที่กลืนปัญหาไปเงียบ ๆ

ก่อนที่คุณจะปล่อยให้มัน “แก้” อะไรก็ตาม ให้ใช้เวลาสิบวินาทีคัดแยกก่อน จริง ๆ แล้วความล้มเหลวของรู้กันความปลอดภัยมีแค่สามแบบ:

- บั๊กจริงที่คุณเพิ่งสร้างขึ้น การตรวจสอบกำลังทำหน้าที่ของมัน แก่ที่โค้ด ไม่ใช่แก๊ที่ตัวตรวจสอบ
- เทสต์ที่ล้าสมัยไปแล้ว คุณจงใจเปลี่ยนพฤติกรรม และเงื่อนไขยืนยันเดิมไม่ตรงกับพฤติกรรมใหม่อีกต่อไป ให้อัปเดตเทสต์ให้ยืนยันพฤติกรรมที่ถูกต้อง แบบใหม่ — อย่างจงใจ ไม่ใช่แบบสัญชาตญาณตอบโต้
- การตรวจสอบที่ไม่เสถียรหรือผิดพลาดเอง เกิดขึ้นได้จริงแต่ค่อนข้างหายาก ให้ระแวงตัวเองในจุดนี้เป็นพิเศษ เพราะนี่คือสิ่งที่คนมักโยนบั๊กจริง ๆ ทั้งลงไป เพราะมันเป็นทางที่ทำให้เดินหน้าต่อไปได้ง่ายที่สุด

คำสั่งที่ช่วยให้ AI ซื่อสัตย์คือการบังคับให้มันวินิจฉัยก่อนที่จะเปลี่ยนแปลงอะไร:

The test "never discounts below zero" is failing.
First, tell me WHY it fails – what does the code do now vs. what the test expects. Then tell me which is correct: the code or the test. Do NOT change anything until I confirm.

การหยุดคิดหนึ่งจังหวะนั้นคือความแตกต่างระหว่างรู้กันความปลอดภัยที่ปกป้องคุณ กับรู้กันความปลอดภัยที่คุณแอบฝึกให้ AI เอาชนะมันได้อย่างเจียบ ๆ การตรวจสอบที่ขึ้นสีแดงคือข้อมูล สิ่งที่แย่ที่สุดที่คุณทำได้คือง่ายเพื่อให้มันเขียว โดยไม่ได้อ่านว่ามันกำลังบอกอะไรคุณ

สรุปและแบบฝึกหัด

ประเด็นสำคัญที่ควรจำ

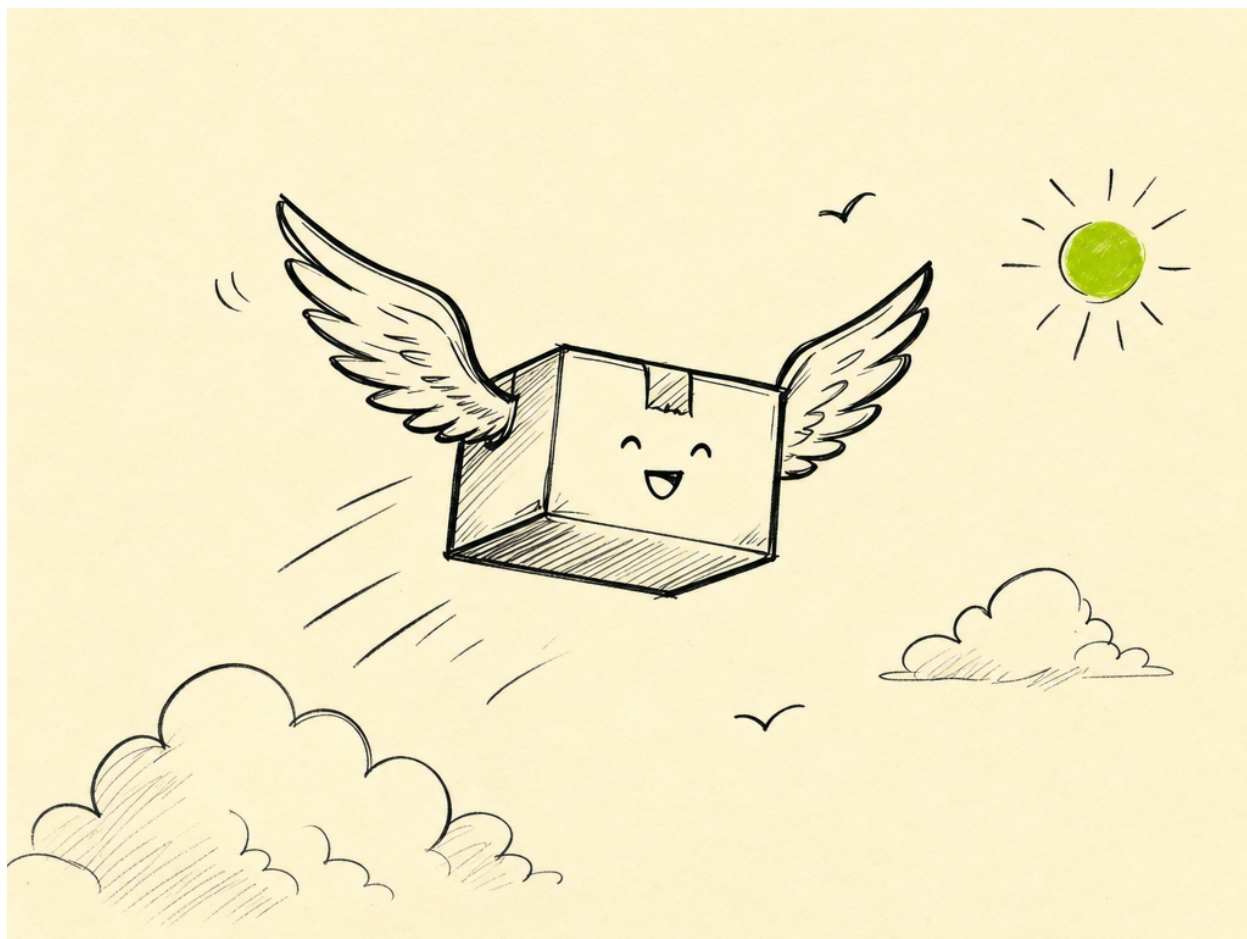
- รู้กันความปลอดภัยคือสิ่งที่ทำให้ AI ร่วงได้เต็มที่ในขณะที่ตาข่ายนิรภัยคอยรองรับการล้ม — มันเร่งความเร็วให้คุณ ไม่ใช่ทำให้ช้าลง
- เทสต์ การตรวจสอบชนิดข้อมูล และ linting จับความผิดพลาดได้ฟรี ให้เชื่อมมันเข้ากับ CI เพื่อให้การตรวจสอบทำงานอัตโนมัติ
- จำกัดขอบเขตการเปลี่ยนแปลงของ AI ให้แคบ และอ่านทุก diff อย่างมีวิจารณญาณก่อน merge
- การตรวจสอบที่ขึ้นสีแดงคือข้อมูล — คัดแยกก่อน (บั๊กจริง / เทสต์ล้าสมัย / การตรวจสอบไม่เสถียร) ก่อนปล่อยให้ AI “แก้” มัน
- อย่าส่งมอบการเปลี่ยนแปลงที่คุณไม่ได้อ่าน

ลองทำดู

ในโปรเจกต์ของคุณเอง ตั้งค่าการตรวจสอบอัตโนมัติที่คุณยังไม่มีสักอย่าง — เทสต์เดียว linter หรือขั้นตอนตรวจสอบชนิดข้อมูล — และเพิ่มมันเข้าไปใน CI workflow เพื่อให้มันรันทุกครั้งที่คุณ push จากนั้นจงใจทำให้บางอย่างพังเล็กน้อย และยืนยันว่าการตรวจสอบขึ้นสีแดงก่อนที่คุณจะแก้มัน

```
Add a CI workflow to this repo that runs the test suite, the linter, and the type checker on every push and pull request. Generate the config, explain each step, and tell me how to read the output when a check fails so I can tell a real bug from an out-of-date test.
```

การนำแอปขึ้นใช้งานจริง (Shipping to Production)



คุณสร้างแอปมาจนถึงจุดนี้แล้ว แอปทำงานไต่บนเครื่องของคุณ เดโมทำให้เพื่อนต้องทึ่ง และความอยากปล่อยมันออกสู่โลกจริงก็เริ่มคันไม้คันมือ นี่คือการจุดที่นักสร้างแอปคนเดียวจำนวนมากสะดุด — ไม่ใช่เพราะโค้ดพัง แต่เพราะการ “ขึ้นใช้งานจริง” (going live) ให้ความรู้สึกเหมือนเป็นศาสตร์อีกแขนงที่มีเวทมนตร์มืดของตัวเอง: DNS (Domain Name System หรือระบบชื่อโดเมน สมุดที่อยู่ของอินเทอร์เน็ตที่แปลงชื่ออย่าง yourdomain.com ให้กลายเป็นหมายเลขที่เซิร์ฟเวอร์ตั้งอยู่), เซิร์ฟเวอร์, ความลับต่าง ๆ (secrets), เว็บสคริปต์ของการชำระเงิน ข่าวดีคืองานเหล่านี้เป็นงานเชิงกลไกที่มีเอกสารอธิบายไว้อย่างดี ซึ่งเป็นสิ่งที่ AI ถนัดที่สุด หน้าที่ของคุณคือการกำกับทิศทาง ส่วนหน้าที่ของ AI คือการพิมพ์คำสั่ง

แต่คำว่า “กำกับ” คือคำสำคัญที่แบกรับน้ำหนักไว้ทั้งหมด ฮีมนของทั้งเล่มนี้คือ เข้าใจสิ่งที่คุณส่งมอบ คุณไม่จำเป็นต้องท่องจำแฟล็กทุกตัว แต่คุณต้องรู้ว่าแต่ละขั้นตอนมีไว้เพื่ออะไร — เว็บสคริปต์ปกป้องคุณจากอะไร ทำไมความลับถึงต้องไม่หลุดเข้าไปในรีโพของคุณเด็ดขาด เพดานการใช้จ่าย (spend cap) นั้นกักขังอะไรไว้ เมื่อคุณเข้าใจรูปทรงของสิ่งนั้น คุณจะสังเกตเห็นได้ว่าเมื่อไหร่ที่คำตอบอันมั่นใจของ AI นั้นผิดอย่างเจียบ ๆ และคุณจะกู่สถานการณ์ได้เมื่อการ deploy ครั้งแรกระเบิด นักสร้างแอปที่ถูกไฟไหม้ตอนเปิดตัวมักเป็นคนที่ว่าคำสั่งที่ตัวเองอ่านไม่ออกเสมอ

บทนี้จะพาเดินตลอดไมล์สุดท้าย จากแอปที่ทำงานอยู่บนเครื่องของคุณ ไปจนถึง URL (Uniform Resource Locator หรือที่อยู่เว็บที่คนพิมพ์ใส่เบราว์เซอร์) จริงที่ผู้คนสามารถจ่ายเงินให้คุณผ่านมันได้

เลือกโฮสต์และ Deploy

สำหรับแอปที่สร้างด้วย vibe coding ส่วนใหญ่ คุณไม่จำเป็นต้องมีเซิร์ฟเวอร์ที่ต้องคอยเลี้ยงดูเอง แพลตฟอร์มยุคใหม่รับโค้ดของคุณแล้วให้ URL ที่ใช้งานได้จริงภายในไม่กี่นาที ตัวเลือกที่พบบ่อย:

- เว็บไซต์แบบ static และ frontend: Cloudflare Pages, Vercel, Netlify, GitHub Pages
- แอปแบบ full-stack และ API (Application Programming Interface หรือประตูที่โปรแกรมหนึ่งใช้คุยกับอีกโปรแกรมหนึ่ง): Cloudflare Workers, Vercel, Render, Fly.io, Railway
- แอปเอนต์ที่หนักกว่านั้น: แพลตฟอร์ม container หรือ VPS ขนาดเล็ก (Virtual Private Server หรือคอมพิวเตอร์ส่วนแบ่งที่คุณเช่าไว้เอง ทำงานตลอดเวลาในศูนย์ข้อมูล)

ทางที่ถูกต้องคือบอก AI ว่าคุณสร้างอะไรมาแล้วให้มันแนะนำ ลองพิมพ์ว่า: “นี่คือแอป Next.js ที่มีฐานข้อมูล Postgres ฉันต้องการโฮสต์ที่ถูกที่สุดแต่เชื่อถือได้ และมี free tier ที่ให้เยอะ ช่วยบอกขั้นตอนการ deploy พร้อมไฟล์คอนฟิกให้หน่อย” ให้ระบุสิ่งที่จำกัดตัวเลือกจริง ๆ ให้ชัดเจน: เฟรมเวิร์กของคุณ มีฐานข้อมูลหรือไม่ ต้องมี background job หรือการอัปโหลดไฟล์หรือไม่ และงบประมาณของคุณ หน้าเว็บการตลาดแบบ static กับแอป full-stack ที่มีคิวนั้นมีบ้านที่ต่างกันมาก

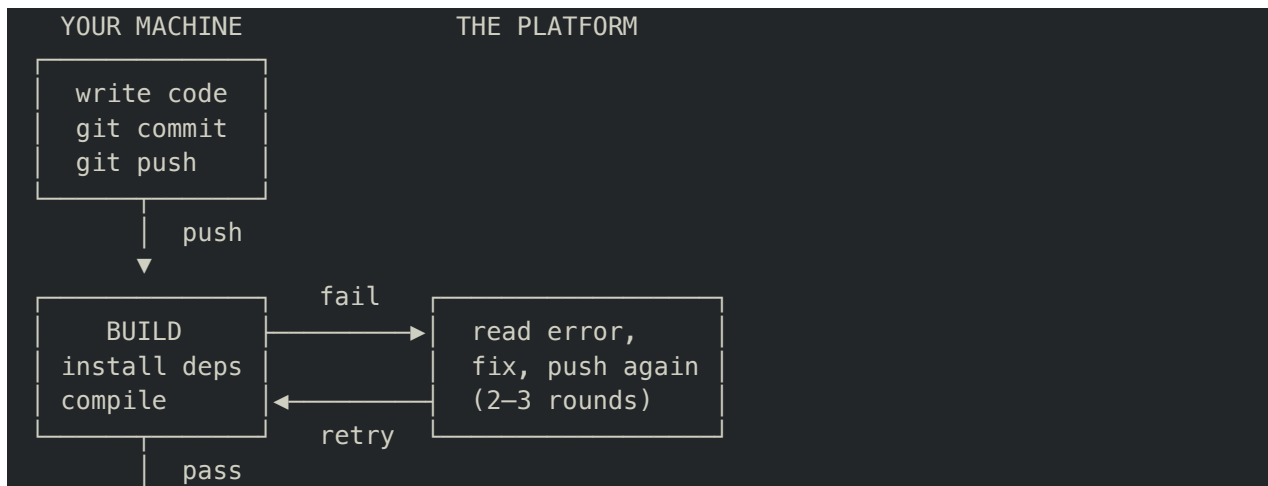
เมื่อเลือกได้แล้ว การ deploy มักจะเป็นแค่คำสั่งไม่กี่บรรทัด:

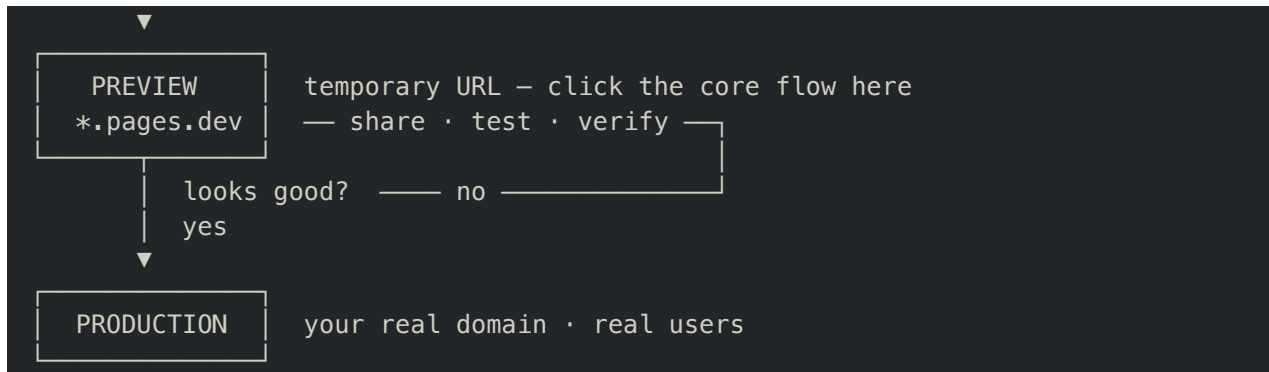
```
# Example: Cloudflare Workers via Wrangler
npm install -g wrangler
wrangler login
wrangler deploy

# Example: Vercel
npm install -g vercel
vercel --prod
```

ให้ AI สร้างไฟล์คอนฟิกของแพลตฟอร์ม (wrangler.toml, vercel.json ฯลฯ) และอธิบายข้อผิดพลาดใด ๆ ที่การ deploy ครั้งแรกโยนออกมา การ deploy ครั้งแรกแทบจะโยนข้อผิดพลาดออกมาเสมอ — คำสั่ง build ที่หายไป ไดรกทอรีเอาต์พุตที่ผิด เวอร์ชัน Node ที่ไม่ตรงกัน หรือตัวแปรสภาพแวดล้อม (environment variable หรือค่าคอนฟิกที่มีชื่อกำกับ ซึ่งถูกส่งให้แอปของคุณจากภายนอกโค้ด เหมือนกุญแจที่ซ่อนไว้ได้พรมเช็ดเท้าที่แอปรู้ว่าต้องไปหา) ที่ build ต้องการแต่หาไม่เจอ ไม่มีข้อไหนหมายความว่า คุณทำอะไรโง่ ๆ มันแค่หมายความว่าแพลตฟอร์มต้องการให้บอกอีกสักอย่าง วางข้อความ error กลับไปแบบคำต่อคำแล้วขอวิธีแก้ พร้อมถามว่า ทำไม มันถึงเกิดขึ้น เพื่อให้ครั้งต่อไปคุณประหลาดใจน้อยลง สักสองสามรอบคุณก็จะขึ้นออนไลน์บน URL ชั่วคราวได้ เปิด URL นั้นแล้วลองคลิกไล่ดูโฟลว์หลักก่อนที่จะฉลอง — “การ deploy สำเร็จ” กับ “แอปทำงานได้จริง” ไม่ใช่คำกล่าวอ้างเดียวกัน

ทุกการ deploy เดินตามไปป์ไลน์เดียวกัน ขั้นตอน preview คือตาข่ายนิรภัยของคุณ: มันให้ทุกการเปลี่ยนแปลง URL ของตัวเองที่คลิกลูกได้ก่อนที่มันจะแตะต้องผู้ใช้จริง





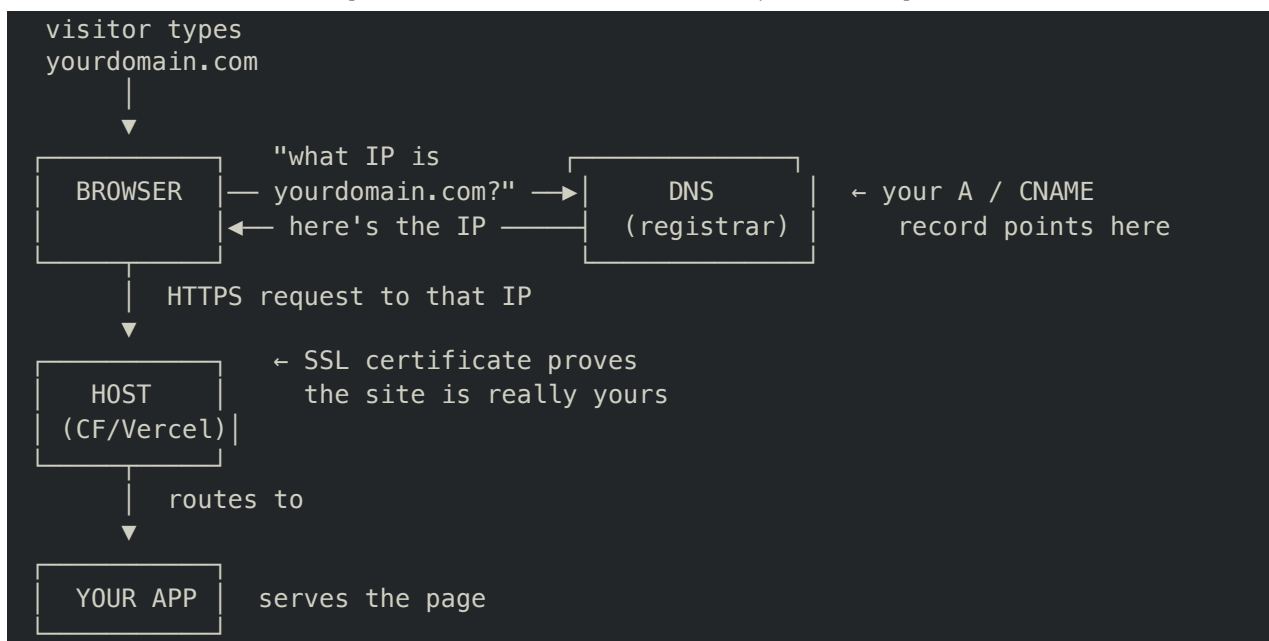
โดเมนกำหนดเองและ DNS

URL แบบ `*.workers.dev` หรือ `*.vercel.app` นั้นใช้ทดสอบได้ดี แต่คุณต้องการชื่อของตัวเอง ขั้นตอนมีดังนี้:

1. ซื้อโดเมน (Namecheap, Cloudflare Registrar, Porkbun — เลือกเจ้าใดเจ้าหนึ่ง)
2. ชี้ DNS ของโดเมนไปยังโฮสต์ของคุณ ปกติแล้วหมายถึงการเพิ่มเรคคอร์ด CNAME หรือ A หรือเปลี่ยนเนมเซิร์ฟเวอร์ (nameserver) ของโดเมนทั้งหมด
3. เพิ่มโดเมนในแดชบอร์ดของโฮสต์แล้วปล่อยให้มันออกใบรับรอง SSL (บัตรประจำตัวดิจิทัลที่พิสูจน์ว่าเว็บไซต์นี้เป็นของคุณจริง ๆ และเปิดใช้งานสัญลักษณ์แม่กุญแจกับ HTTPS ซึ่งเป็นเวอร์ชันที่ปลอดภัยและเข้ารหัสของโปรโตคอลการถ่ายโอนข้อมูลเว็บ) ให้โดยอัตโนมัติ

มาทำความเข้าใจแบบง่าย ๆ กันเล็กน้อย เพื่อให้เรคคอร์ดพวกนี้เล็กลงเหมือนเวทมนตร์: A record ชี้ชื่อโดเมนตรงไปยังที่อยู่ IP; CNAME ชี้ชื่อหนึ่งไปยัง ชื่ออื่น (มีประโยชน์เมื่อ IP ของโฮสต์อาจเปลี่ยนได้โดยที่คุณไม่รู้ตัว); และ nameserver คือการมอบโดเมนทั้งหมดให้ผู้ให้บริการจัดการเรคคอร์ดทุกตัวให้แทนคุณ โฮสต์สมัยใหม่ส่วนใหญ่ชอบให้ใช้ CNAME หรือเปลี่ยนเนมเซิร์ฟเวอร์ไปเลย เพราะพวกเขาต้องการจัดการใบรับรอง SSL และการเรดที่ edge ด้วยตัวเอง

นี่คือห่วงโซ่ทั้งหมดที่คำขอของผู้เยี่ยมชมเดินทางผ่าน และแต่ละส่วนที่คุณตั้งค่าเองอยู่ตรงไหนในนั้น:



DNS คือ ส่วนที่ทำให้ ผู้เริ่มต้น สับสน ที่สุด ส่วนใหญ่ เป็น เพราะ ศัพท์ เฉพาะ และ ความ ล่าช้า ใน การก กระจาย ข้อมูล (propagation delay) ให้ AI ช่วยแปลให้ ลองถามว่า: “โดเมนของฉันจดทะเบียนไว้ที่ Namecheap และโฮสต์บน

Cloudflare Pages ช่วยเดินทีละขั้นให้หน่อยว่าต้องเพิ่มเรคคอร์ดอะไร ค่าเป็นอะไร และตรวจสอบได้ยังไงว่ามันใช้ได้แล้ว” จากนั้นตรวจสอบผลด้วยตัวเองด้วยการเช็คแบบเร็ว ๆ แทนที่จะรีเฟรชเบราว์เซอร์เฉย ๆ:

```
# Look up the records the world actually sees
dig yourdomain.com
nslookup yourdomain.com

# Confirm the certificate is live and not expired
curl -I https://yourdomain.com
```

การเปลี่ยนแปลง DNS อาจใช้เวลากระจายตั้งแต่ไม่กี่นาทีไปจนถึงหนึ่งวัน และแคชของเบราว์เซอร์ของคุณเองคือตัวโกหกที่แย่ที่สุด — มันจะแสดงคำตอบเก่าให้คุณดูอย่างมีความสุขหลังจากที่อินเทอร์เน็ตส่วนที่เหลืออัปเดตไปแล้ว ถ้ามันยังไม่ขึ้นทันที นั่นเป็นเรื่องปกติ ลองเช็คจากมือถือผ่านเน็ตมือถือ หรือใช้เครื่องมือตรวจสอบ “DNS propagation” ออนไลน์ ก่อนที่จะสรุปว่ามีอะไรพัง

ตัวแปรสภาพแวดล้อมและความลับ (Environment Variables and Secrets)

โค้ดของคุณต้องการ API key, URL ฐานข้อมูล และโทเคนต่าง ๆ สิ่งเหล่านี้ ต้อง ไม่ถูก commit เข้ารีโพสิทอรีเด็ดขาด เมื่อความลับหลุดเข้าไปในประวัติ git แล้ว มันก็เท่ากับเปิดเผยต่อสาธารณะตลอดไปโดยพฤตินัย — การลบบรรทัดนั้นออกใน commit ที่หลังไม่ได้ ลบ มันออกจากประวัติจริง ๆ และบอตที่คอยสแกนหารีโพสิทอรีสาธารณะใหม่ ๆ เพื่อหา key ที่หลุดออกมานั้นทำงานเร็วภายในไม่กี่นาที กฎสองข้อนี้จะพาคุณไปได้ไกล:

- เก็บความลับไว้ในไฟล์ .env บนเครื่องเท่านั้น และตรวจให้แน่ใจว่า .env อยู่ใน .gitignore ของคุณ
- ตั้งค่าค่าเดียวกันเป็นตัวแปรสภาพแวดล้อมในแดชบอร์ดของโฮสต์คุณ (หรือผ่าน CLI — Command-Line Interface หรือหน้าต่างพร้อมป้อนข้อความที่คุณพิมพ์คำสั่งแทนการคลิก) สำหรับ production

```
# .env (local only — never commit this)
DATABASE_URL=postgres://localhost:5432/app
STRIPE_SECRET_KEY=sk_test_xxx
SESSION_SECRET=change-me

# Set the production equivalents on the host
wrangler secret put STRIPE_SECRET_KEY
vercel env add STRIPE_SECRET_KEY production
```

รักษาไฟล์ .env.example ที่ commit ไว้ ซึ่งแสดงรายการ ชื่อ (ไม่ใช่ค่า) ของตัวแปรทุกตัว เพื่อให้ตัวคุณในอนาคตและ AI รู้ว่าแอปคาดหวังอะไรบ้าง สิ่งเก๋าคำน่าหน้า sk_test_ ด้านบน — ให้แยก key ทดสอบกับ key จริงให้แยกแยะออกจากกันชัดเจน และอย่าปล่อยให้ key จริงอยู่ใน .env บนเครื่องที่คุณอาจเผลอลงในแชท ถ้าคุณเคยวางความลับลงในแชท หรือ commit มันเข้าไปโดยไม่ตั้งใจ ให้ถือว่ามันถูกเปิดเผยแล้วและหมุนเวียน (rotate) มันทันที: สร้าง key ใหม่ในแดชบอร์ดของผู้ให้บริการ อัปเดตทุกที่ที่ใช้มัน แล้วเพิกถอน key เก่า การหมุนเวียนใช้เวลาแค่ห้านาที แต่ URL ฐานข้อมูลที่หลุดออกไปอาจทำให้คุณเสียทุกอย่างที่อยู่ในฐานข้อมูลนั้น

การเชื่อมต่อบริการชำระเงิน

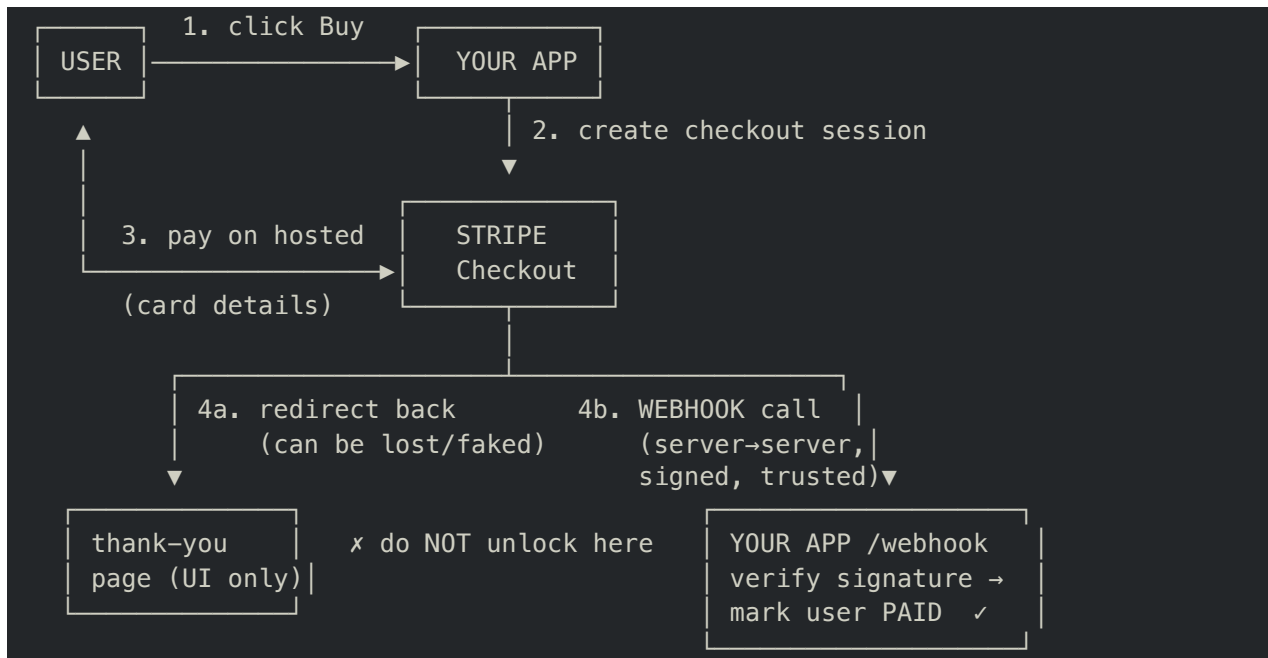
ถ้าคุณอยากได้รับเงิน เส้นทางมาตรฐานสำหรับนักสร้างแอปคนเดียวคือ Stripe (หรือ Lemon Squeezy / Paddle ถ้าคุณอยากให้พวกเขาจัดการภาษีขายให้ — เพราะในฐานะ merchant of record พวกเขาหักส่วนแบ่งมากกว่า แต่ทำให้ปัญหาปวดหัวเรื่องภาษีหายไป) กลไกเหล่านี้ทำซ้ำ ๆ กันและมีเอกสารอธิบายไว้ดี ทำให้เหมาะแก่การมอบหมายให้ AI ทำ ส่วนประกอบที่คุณต้องมี:

- โฟลว์ checkout — Stripe Checkout เป็นวิธีที่เร็วที่สุด มันมอบหน้าชำระเงินที่โฮสต์ไว้ให้คุณ ทำให้คุณไม่ต้องแตะข้อมูลบัตรเครดิตเลย
- ลองนึกภาพเว็บสืคเหมือนบริษัทรับชำระเงินโทรบอกแอปของคุณทันทีที่การขายสำเร็จ แทนที่แอปของคุณจะต้องคอยถามซ้ำ ๆ ว่า “ผ่านหรือยัง” เว็บสืค (webhook) ก็คือสิ่งนั้นเป๊ะ ๆ: เอนด์พอยต์ (URL บนเซิร์ฟเวอร์ของคุณ) ที่

ผู้ให้บริการชำระเงินเรียกใช้เพื่อให้แอปของคุณรู้ว่าการชำระเงินสำเร็จจริง นี่เป็นส่วนที่ผู้เริ่มต้นมักข้ามไปแล้วก็มาสงสัยทีหลังว่าทำไมออเดอร์ไม่เคยถูกส่งมอบ การ redirect กลับมายังเว็บไซต์ของคุณอาจถูกปิด ถูกบล็อก หรือถูกลบมได้ ส่วนเว็บสืคคือแหล่งความจริงระหว่างเซิร์ฟเวอร์กับเซิร์ฟเวอร์ อย่าเชื่อ redirect เพียงอย่างเดียว จงเชื่อเว็บสืค

- วิธี เก็บสิทธิ์การใช้งาน (entitlements) — บันทึกกว่าผู้ใช้จ่ายเงินแล้วในฐานข้อมูลของคุณ เพื่อให้การรีเฟรชหรือการเข้าจากอุปกรณ์ใหม่ยังคงเห็นสิ่งที่พวกเขาซื้อไว้

ประเด็นสำคัญคือ ลูกศรเส้นไหน ที่คุณควรเชื่อ การ redirect ของผู้ใช้กลับมายังเว็บไซต์คุณอาจสูญหายหรือถูกลบมได้ ส่วนเว็บสืคคือเส้นทางระหว่างเซิร์ฟเวอร์กับเซิร์ฟเวอร์ที่ปลอดภัยสินค้าจริง ๆ:



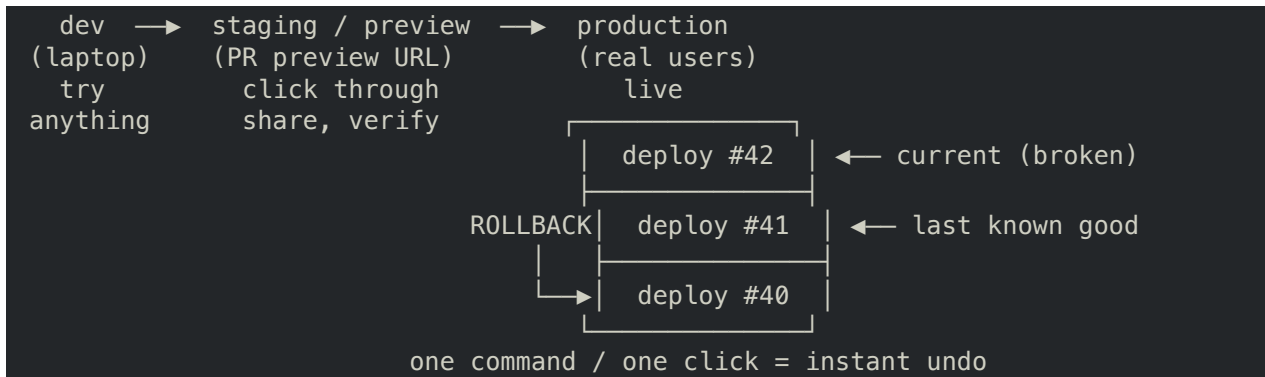
ขอให้ AI สร้าง โครงสำหรับทั้งสามส่วนนี้ให้ และอธิบายการตรวจสอบลายเซ็นของเว็บสืค (webhook signature verification) เพราะนี่คือขั้นตอนเดียวที่อ่อนไหวต่อความปลอดภัย — ถ้าไม่มีการตรวจสอบนี้ ใครก็ตามที่หา URL เว็บสืคของคุณเจอก็สามารถปลอมเหตุการณ์ “ชำระเงินสำเร็จ” และปลดล็อกสินค้าของคุณได้ฟรี ๆ ทดสอบทุกอย่างในโหมดทดสอบของ Stripe ด้วยหมายเลขบัตรทดสอบของพวกเขา (4242 4242 4242 4242 คือหมายเลขที่โด่งดัง) ก่อนที่จะเปลี่ยนไปใช้ key จริง และใช้ Stripe CLI เพื่อส่งต่อเหตุการณ์เว็บสืคมายังเครื่องของคุณ เพื่อที่คุณจะได้เห็นมันทำงานจริง อย่าข้ามการซ้อมในโหมดทดสอบ เงินจริงเงินแรกทีไหลผ่านแอปของคุณควรเป็นธุรกรรมที่คุณเคยเห็นมันทำงานสำเร็จมาแล้วนับสิบครั้ง

Staging และ Rollback

นิสัยสองอย่างแบ่งแยกการเปิดตัวที่กู้คืนได้อย่างสง่างามออกจากการเปิดตัวที่กลายเป็นความตื่นตระหนกตอนเที่ยงคืน: การทดสอบการเปลี่ยนแปลงในที่ที่ปลอดภัยก่อนที่มันจะถึงมือผู้ใช้จริง และความสามารถในการยกเลิกการ deploy ที่แย่ได้ทันที

Staging environment คือสำเนาที่สองของแอปคุณ — โค้ดเดียวกัน แต่ URL แยก ฐานข้อมูลแยก (หรือแบบใช้แล้วทิ้ง) — ที่คุณลองอะไรก่อนขึ้น production โสสต์ส่วนใหญ่ทำให้สิ่งนี้แทบไม่มีค่าใช้จ่ายเลย: ทุก pull request จะได้ preview URL ของตัวเองโดยอัตโนมัติ ทำให้คุณคลิกไล่ดูการเปลี่ยนแปลงและแชร์ให้คนอื่นดูก่อน merge ได้ ใช้มันโดยเฉพาะกับสิ่งที่ย้อนกลับได้ยาก เช่น การ migrate ฐานข้อมูล

ลองนึกภาพสามสภาพแวดล้อมเป็นบันไดที่การเปลี่ยนแปลงต้องปีนขึ้นไป โดยมี rollback เป็นเชือกที่ดึง production กลับไปยังขั้นที่ต่ำสุดได้ทันที:



Rollback คือปุ่ม undo ของแอปทั้งตัวคุณ เพราะแพลตฟอร์มเหล่านี้เก็บการ deploy ก่อนหน้าทุกครั้งไว้ การย้อนกลับไปยังเวอร์ชันที่ล่าสุดมักเป็นแค่คำสั่งเดียวหรือคลิกเดียวบนแดชบอร์ด:

```
# Cloudflare Pages: redeploy a known-good build
wrangler pages deployment list
wrangler pages deployment rollback <DEPLOYMENT_ID>

# Vercel: promote a previous deployment back to production
vercel rollback <DEPLOYMENT_URL>
```

รู้จักคำสั่งนี้ ก่อน ที่คุณจะต้องใช้มัน ไม่ใช่ระหว่างที่ระบบล่ม ประเด็นทั้งหมดของความเร็วแบบ vibe-coded คือการ ship นั้นถูกและเร็ว — แต่การ ship ที่ถูกจะรู้สึกปลอดภัยก็ต่อเมื่อการ un-ship นั้นถูกพอ ๆ กัน

การเฝ้าติดตามและติดตามข้อผิดพลาด

เมื่อผู้คนเริ่มใช้แอปของคุณ คุณต้องรู้ว่ามันพังตอนไหน — ควรรู้ก่อนที่พวกเขาจะมาบอกคุณ ชุดเครื่องมือเริ่มต้นแบบเบา ๆ:

- การติดตามข้อผิดพลาด (error tracking): Sentry จะดักจับและจัดกลุ่ม exception พร้อม stack trace และบอกคุณว่าผู้ใช้คนไหนและบรรทัดไหนที่ทำให้เกิดข้อผิดพลาดแต่ละอัน ติดตั้งแค่ไม่กี่บรรทัด
- การเฝ้าติดตามเวลาทำงาน (uptime monitoring): ตัว pinger ฟรี (UptimeRobot, Better Stack) จะยิงเข้าเว็บไซต์ของคุณทุก ๆ ไม่กี่นาทีและอีเมลบอกคุณเมื่อมันล่ม
- Logs: รู้ว่าโฮสต์ของคุณเก็บ log ไว้ที่ไหน (wrangler tail, แดชบอร์ดของ Vercel ฯลฯ) เพื่อให้คุณตรวจสอบได้เมื่อมีอะไรผิดปกติ

ให้ AI เพิ่ม Sentry และเชื่อมต่อเอนด์พอยต์ health-check พื้นฐาน — เส้นทางเล็ก ๆ อย่าง /health ที่คืนค่า 200 OK และยืนยันว่าเข้าถึงฐานข้อมูลได้ ซึ่ง uptime monitor ของคุณสามารถ ping ได้ เป้าหมายไม่ใช่ระบบสังเกตการณ์ระดับองค์กร แต่เป็นเพียงแค่ว่าไม่ต้องมารู้เรื่องระบบล่มจากทวีตค่า

ประสิทธิภาพและการแคช

แอปที่ช้าจะเสียผู้ใช้และได้อันดับแย่ลงในการค้นหา คุณไม่จำเป็นต้องปรับแต่งละเอียดยิบในวันแรก แต่คว่ำชัยชนะที่ได้มาง่าย ๆ:

- ลองนึกภาพการเก็บสำเนาไฟล์ของคุณไว้ในคลังสินค้าทั่วโลก เพื่อให้ผู้เยี่ยมชมแต่ละคนได้รับบริการจากคลังที่ใกล้ที่สุด นั่นคือ CDN (Content Delivery Network) — ให้บริการ static asset ผ่าน CDN (โฮสต์ส่วนใหญ่ทำสิ่งนี้ให้อัตโนมัติ)
- แคช คำตอบและค่าคำนวณข้อมูลที่มีค่าใช้จ่ายสูงและไม่ค่อยเปลี่ยนแปลง
- บีบอัดและโหลดรูปภาพแบบ lazy-load ลด JavaScript ที่ส่งออกไป
- รัน Lighthouse ใน Chrome DevTools แล้วส่งรายงานให้ AI เพื่อจัดลำดับความสำคัญของสิ่งที่ต้องแก้

ลองถามว่า: “นี่คือรายงาน Lighthouse ของฉัน ช่วยบอกสามการเปลี่ยนแปลงที่ส่งผลกระทบมากที่สุดโดยใช้ความพยายามน้อยที่สุด” ให้การวัดผล ไม่ใช่การเดา เป็นตัวขับเคลื่อนสิ่งที่คุณจะปรับแต่ง — เป็นเรื่องปกติจริง ๆ ที่จะใช้เวลาทั้งวันปรับแต่งสิ่งผิด ๆ เพราะมัน รู้สึก ช้ำ ทั้งที่ต้นตอที่แท้จริงคือค่าคั่นเดียวที่ไม่มี index หรือรูปภาพหน้าปกขนาด 4 MB ที่คุณลืมบีบอัด

รายการตรวจสอบก่อนเปิดตัว

ก่อนที่คุณจะประกาศอะไรก็ตาม ให้ไล่ตามรายการนี้ ให้ AI ช่วยตรวจสอบแต่ละรายการแทนที่จะเชื่อความจำของตัวเอง

- ☐ แอป deploy ได้อย่างราบรื่นและ URL production โหลดขึ้นมาได้
- ☐ โดเมนกำหนดเองแปลงค่าได้และมี HTTPS ที่ใช้งานได้
- ☐ ความลับทั้งหมดถูกตั้งค่าใน production แล้ว ไม่มีตัวไหนถูก commit เข้ารีโพ
- ☐ .env อยู่ใน gitignore และ .env.example เป็นเวอร์ชันล่าสุด
- ☐ การชำระเงินถูกทดสอบครบวงจร (รวมถึงเว็บสค) ในโหมดทดสอบแล้ว
- ☐ การสมัครสมาชิก การล็อกอิน และเส้นทางการใช้งานหลักทำงานได้ใน production ทั้งหมด
- ☐ การติดตามข้อผิดพลาดและการเฝ้าติดตามเวลาทำงานทำงานอยู่แล้ว
- ☐ คุณรู้วิธี rollback และได้ทดสอบมันแล้วอย่างน้อยหนึ่งครั้ง
- ☐ เลย์เอาต์บนมือถือถูกตรวจสอบบนโทรศัพท์จริงแล้ว
- ☐ มีหน้านโยบายความเป็นส่วนตัวและข้อกำหนดการใช้งานพื้นฐานอยู่แล้ว
- ☐ เปิดใช้งานการสำรองข้อมูลฐานข้อมูลแล้ว
- ☐ มีช่องทางที่ชัดเจนให้ผู้ติดต่อคุณได้ (อีเมลหรือแบบฟอร์ม)

หลังจากที่คุณเปิดตัว

การ ship คือจุดเริ่มต้น ไม่ใช่จุดจบ เวอร์ชันแรกคือสมมติฐาน ผู้ใช้จริงคือการทดลอง เฝ้าดูแดชบอร์ดข้อผิดพลาดของคุณ อ่านทุกข้อความ และ ship การแก้ไขเล็ก ๆ ให้เร็ว ด้านทานความอยากที่จะรี้อสร้างทุกอย่างใหม่ทันทีที่เจอความผิดพลาดครั้งแรก — ปะแต่ละจุดที่คม เพิ่มการลงทุนในสิ่งที่ผู้คนใช้จริง และปล่อยให้ส่วนที่เหลือรอไปก่อน

นิสัยที่ทรงพลังที่สุดตรงนี้เป็นรูปที่แน่นอน: สังเกตปัญหาหรือไอเดีย อธิบายให้ AI ฟัง ship การเปลี่ยนแปลง ดูผลลัพธ์ เพราะงานเชิงกลไกถูกมอบหมายไปแล้ว คุณจึงทำซ้ำได้ภายในไม่กี่ชั่วโมงแทนที่จะเป็นสัปดาห์ ความเร็วนี้ — ไม่ใช่พีเจอาร์เดียว ๆ โหนดเลย — คือข้อได้เปรียบที่แท้จริงของการสร้างด้วย vibe คุณขึ้นออนไลน์แล้ว ที่นี้ก็เดินหน้าต่อไป

สรุปและฝึกปฏิบัติ

ประเด็นสำคัญที่ควรจำ

- การ ship คือ workflow ไม่ใช่ช่วงเวลาเดียว: deploy สังเกต ปะแก้ ทำซ้ำ — และทำให้ rollback เป็นสิ่งที่คุณเคยทดสอบมาแล้ว
- รันรายการตรวจสอบก่อนเปิดตัวก่อนที่คุณจะประกาศอะไรก็ตาม และให้ AI ตรวจสอบแต่ละรายการแทนที่จะเชื่อความจำของตัวเอง
- ให้การวัดผลเป็นตัวขับเคลื่อนการปรับแต่ง — รายงาน Lighthouse ชนะการเดาว่าอะไร “รู้สึก” ช้ำเสมอ
- ความลับอยู่ในคอนฟิกของ production เท่านั้น ไม่เคยอยู่ในรีโพ .env อยู่ใน gitignore และ .env.example เป็นปัจจุบันเสมอ
- เวอร์ชันแรกคือสมมติฐาน ผู้ใช้จริงคือการทดลอง ดังนั้นให้ทำซ้ำด้วยการแก้ไขเล็ก ๆ ที่รวดเร็ว

ลองทำดู

หยิบแอปที่คุณ deploy ไว้แล้ว (หรือ deploy อันเล็ก ๆ ขึ้นมาตอนนี้เลย) แล้วรันรายการตรวจสอบก่อนเปิดตัวทั้งหมดกับมันทีละบรรทัด สำหรับทุกช่องที่คุณติ๊กไม่ได้อย่างสัตย์จริง วางรายการที่ล้มเหลวนั้นให้ AI ดูแล้วขอการเปลี่ยนแปลงที่เล็กที่สุดที่จะปิดช่องว่างนั้น ปิดท้ายด้วยการคลิก deep link บนเว็บไซต์จริง — ไม่ใช่แค่หน้าแรก — เพื่อยืนยันว่า production ทำงานเหมือนกับ dev

พรมปีประจำบท

```
Here is my deployed app and its stack: [describe app + hosting + services].  
Act as my launch reviewer. Walk this pre-launch checklist item by item  
and, for each, tell me how to verify it on the LIVE site (not in dev):  
- production URL loads over valid HTTPS on the custom domain  
- all secrets set in production; none committed to the repo  
- core happy path (sign-up, login, main action) works in production  
- payments tested end-to-end including the webhook (if any)  
- error tracking and uptime monitoring are live  
- I can roll back, and have tested it once  
- mobile layout works on a real phone  
- database backups are enabled  
For every item that fails, give me the SMALLEST change that fixes it.  
Do not assume anything passes – make me prove each one.
```

กรณีศึกษา: จากศูนย์สู่การเปิดตัว



ทฤษฎีมีประโยชน์ แต่ไม่มีอะไรสอนการทำ vibe coding ได้ดีเท่ากับการดูมันเกิดขึ้นจริง บทนี้จะพาไปดูโปรเจกต์เล็ก ๆ แต่จริงจังสี่โปรเจกต์ ที่สร้างและเปิดตัวแบบครบวงจรด้วยการสั่งงาน AI ไม่มีโปรเจกต์ไหนเป็นของเล่น แต่ละอันมีผู้ใช้จริง มีชื่อโดเมนจริง และมีเงินหรือเวลาเป็นเดิมพัน

ในแต่ละกรณี คุณจะได้เห็นไอเดียตั้งต้น สเปกที่เรามอบให้ AI สดักที่เราเลือกใช้ พรอมต์ที่ทำให้เกิดความคืบหน้าจริง

ๆ อุปสรรคที่เกือบทำให้โปรเจกต์พังไปกลางทาง และวิธีที่มันขึ้นใช้งานจริง ให้อ่านสิ่งเหล่านี้เหมือนคู่มือปฏิบัติการ ไม่ใช่ไฮไลต์โชว์ผลงาน ประเด็นไม่ได้อยู่ที่โปรเจกต์เหล่านี้จะสำคัญแค่ไหน แต่อยู่ที่ ท่าทาง การทำงานที่ซ้ำ ๆ กัน พออ่านถึงกรณีนี้ที่สี่ คุณจะเริ่มจับจิ้งหะได้ นั่นคือสเปกสั้น ๆ สดุดีที่มีคนดูแลให้ การดีบั๊กด้วยหลักฐาน และการเปิดตัวที่ทดสอบกับความเป็นจริง

กรณีศึกษาที่ 1: หน้า Landing Page หลายภาษา + ระบบชำระเงิน

ไอเดีย

เพื่อนคนหนึ่งขายคู่มือ PDF ราคา 29 ดอลลาร์ให้กับชาวต่างชาติที่ย้ายมาอยู่เกาหลี เธอต้องการหน้าเว็บเพจเดียวที่โหลดเร็วทั่วโลก พูดได้ทั้งภาษาอังกฤษ เกาหลี และจีน และรับชำระเงินด้วยบัตรเครดิตได้โดยที่เธอไม่ต้องแตะต้องเซิร์ฟเวอร์เลย

สเปก

เราเขียนสเปกให้อยู่ในหนึ่งย่อหน้าก่อนจะเขียนโค้ดสักบรรทัด:

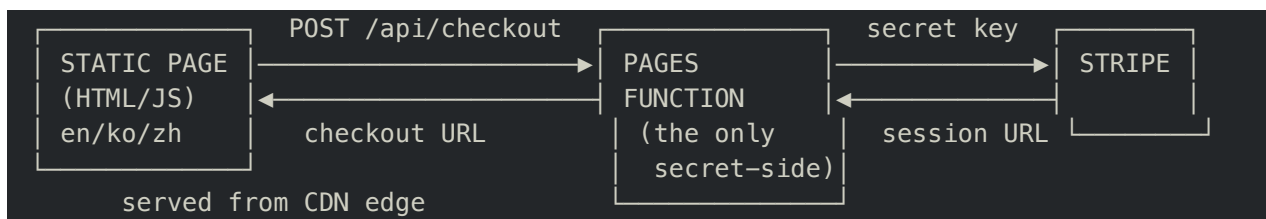
```
Build a single-page marketing site for a $29 digital product.
Three languages (en/ko/zh), auto-detected from the browser but
switchable with a toggle. One "Buy" button that opens Stripe
Checkout and redirects to a thank-you page on success. No backend
server I have to maintain. Must load fast from anywhere.
```

สังเกตสิ่งที่สเปกนี้ ไม่ได้ พูดถึง ไม่มีเฟรมเวิร์ก ไม่มีดีไซน์ระบบ ไม่มีระบบวิเคราะห์ข้อมูล ไม่มีการเก็บอีเมล แต่ละสิ่งเหล่านั้นล้วนเป็นฟีเจอร์ที่สมเหตุสมผลได้ แต่ทุกอันก็จะทำให้การเปิดตัวครั้งแรกช้าลงเช่นกัน วินัยของสเปกหนึ่งย่อหน้าส่วนใหญ่แล้วก็คือวินัยของการ “ตัดออก” นั่นเอง

สแต็กที่เลือกใช้

“ไม่มีแบ็กเอนด์ที่ต้องดูแลเอง” บวกกับ “เร็วทั่วโลก” ชี้ตรงไปที่การโฮสต์แบบสแตติกบน edge CDN (Content Delivery Network — เครือข่ายกระจายเนื้อหา คือสำเนาไฟล์ของคุณที่เก็บไว้ตามศูนย์ข้อมูลทั่วโลก เพื่อให้ผู้เข้าชมแต่ละคนโหลดจากศูนย์ที่ใกล้ที่สุด) พร้อมฟังก์ชัน serverless เล็ก ๆ หนึ่งตัวสำหรับสิ่งเดียวที่ต้องการ ความลับ นั่นคือการสร้าง Stripe session เราเลือกเว็บไซด์สแตติกบน Cloudflare Pages พร้อม Pages Function เดียว สถาปัตยกรรมทั้งหมดอธิบายได้ในหนึ่งประโยค คือหน้าเว็บสแตติกที่เรียกใช้ฟังก์ชันเดียวที่คุยกับ Stripe และความเรียบง่ายนี้แหละที่ทำให้ดีบั๊กได้ง่ายในภายหลัง

ทั้งระบบมีสามกล่อง และมีเพียงกล่องตรงกลางเท่านั้นที่แตะต้องความลับ:



พรมแดนสำคัญ

เราเริ่มแบบกว้าง ๆ แล้วให้ AI เสนอโครงสร้างเอง:

```
Scaffold a static landing page (plain HTML/CSS/JS, no framework)
with a language toggle for en/ko/zh. Store the copy in a single
JS object keyed by language. Detect the default from
navigator.language. Keep it to one index.html plus one main.js.
```


จากนั้นเป็นเส้นทางการชำระเงิน ซึ่งเป็นจุดที่ความลับอาศัยอยู่:

```
Add a Cloudflare Pages Function at /api/checkout that creates a Stripe Checkout session for a single $29 product and returns the redirect URL. Read STRIPE_SECRET_KEY from the environment, never hardcode it. The Buy button should POST to this function and then window.location to the returned URL.
```

วลี “never hardcode it” (อย่าฝังค่าตายตัวลงในโค้ด) นั้นคุ่มค่าที่จะพูดออกไปตรง ๆ ถ้าปล่อยให้เป็นไปตามค่าเริ่มต้นของมันเอง AI จะเปลอสคีย์ตัวอย่างลงในโค้ดโดยตรงเพื่อให้ตัวอย่าง “ใช้งานได้” และคีย์ตัวอย่างก็มีวิธีกลายเป็นคีย์จริงที่หลุดออกไปพร้อมกับการเปิดตัว การพูดออกมาให้ชัดเจนในพรอมต์เพียงครั้งเดียว ถูกกว่าการมาจับผิดตอนรีวิวโค้ดทีหลังมาก

อุปสรรค

การทดสอบใช้งานจริงครั้งแรกล้มเหลว ปุ่ม Buy ไม่ทำอะไรเลย และคอนโซลของเบร้าวเซอร์แสดงข้อผิดพลาด CORS (Cross-Origin Resource Sharing — กฎของเบร้าวเซอร์ว่าเว็บไซต์ไหนได้รับอนุญาตให้เรียกเว็บไซต์ไหนได้บ้าง) เราไม่ได้เดา แต่วางข้อความ error ตัวจริงกลับไปให้ AI ดู:

```
Clicking Buy logs: "Access to fetch at '/api/checkout' blocked by CORS policy." The function and page are on the same domain. Here's the function code: [pasted]. What's actually wrong?
```

AI จับจุดได้ทันที ฟังก์ชันกำลังส่ง Stripe URL กลับมาเป็น JSON (JavaScript Object Notation — รูปแบบข้อความง่าย ๆ ที่อ่านได้ ซึ่งโปรแกรมใช้แลกเปลี่ยนข้อมูลแบบมีโครงสร้างกัน) แต่ตัว fetch ถูกเรียกก่อนที่ preflight แบบ OPTIONS ของฟังก์ชันจะถูกจัดการ เพราะเราดันตีพลอยหน้าเว็บสแตติกกับฟังก์ชันไปคนละโปรเจกต์ Pages โดยไม่ตั้งใจ ทางแก้ที่แท้จริงคือเรื่องโครงสร้างการตีพลอย ไม่ใช่โค้ด เราย้ายฟังก์ชันเข้าไปอยู่ในโฟลเดอร์ /functions ของโปรเจกต์เดียวกัน แล้ว error ก็หายไป

บทเรียนตรงนี้ควรค่าแก่การครุ่นคิด อาการ ที่เห็นอยู่ในโค้ด (error CORS จาก fetch) แต่ สาเหตุ อยู่ที่โครงสร้างพื้นฐาน (สองโปรเจกต์แทนที่จะเป็นหนึ่ง) ถ้าเราปล่อยให้ AI “แก้ error CORS” โดยไม่วางบริบทการตั้งค่าให้ดู มันคงจะเปะปะเสดเดอร์ Access-Control-Allow-Origin เข้าไปแทน ซึ่งเป็นการแก้ที่คอมไพล์ผ่านแต่กลบอาการไว้เฉย ๆ ทั้งบักตัวจริงไว้เหมือนเดิม วางข้อความ error ตัวจริง พร้อมกับ บริบทรอบข้าง และอย่ารับคำอธิบายแรกที่ฟังดูสมเหตุสมผลถ้ามันไม่ตรงกับสถานการณ์จริงของคุณ

การเปิดตัว

เราเพิ่มคีย์ Stripe ตัวจริงเป็นตัวแปรสภาพแวดล้อมแบบเข้ารหัสในแดชบอร์ดของ Pages (ไม่เคยอยู่ใน repo) ชี้ DNS (Domain Name System สมุดที่อยู่ของอินเทอร์เน็ตที่แปลงชื่อโดเมนให้กลายเป็นเซิร์ฟเวอร์) ของโดเมนเธอไปที่ Pages และทำการซื้อจริงหนึ่งครั้งด้วยบัตรจริงมูลค่า 29 ดอลลาร์ แล้วคืนเงินทีหลัง เรายังลองคลิกผ่านเส้นทางที่ล้มเหลวโดยตั้งใจ ด้วยบัตรทดสอบที่ถูกปฏิเสธ เพื่อยืนยันว่าเธอไม่ถูกเรียกเก็บเงิน และเห็นข้อความที่สมเหตุสมผลแทนที่จะเป็นหน้าจอว่างเปล่า ขึ้นใช้งานจริงได้ภายในพายวันเดียว เธอขายได้ครั้งแรกในเย็นวันนั้นเอง

กรณีศึกษาที่ 2: เครื่องมือ SaaS ขนาดเล็กที่มีระบบล็อกอินและฐานข้อมูล

ไอดี

นักออกแบบฟรีแลนซ์คนหนึ่งต้องการแดชบอร์ดส่วนตัวสำหรับบันทึกชั่วโมงทำงานที่เรียกเก็บเงินได้ต่อลูกค้าแต่ละราย และส่งออกเป็น CSV (Comma-Separated Values — ไฟล์สเปรดชีตแบบข้อความส่วนที่แอปสเปรดชีตไหนก็เปิดได้) รายเดือน ไม่ต้องหุรหุระอะไร แต่ต้องมีระบบบัญชี (เพื่อให้ข้อมูลของเธอเป็นของเธอคนเดียว) และต้องเก็บข้อมูลถาวรได้

สเปก

เมื่อมีระบบล็อกอินและฐานข้อมูลเข้ามาเกี่ยวข้อง เดิมพันก็สูงขึ้น สเปกจึงต้องระบุ ขอบเขต ให้ชัดเจนกว่าเดิม:

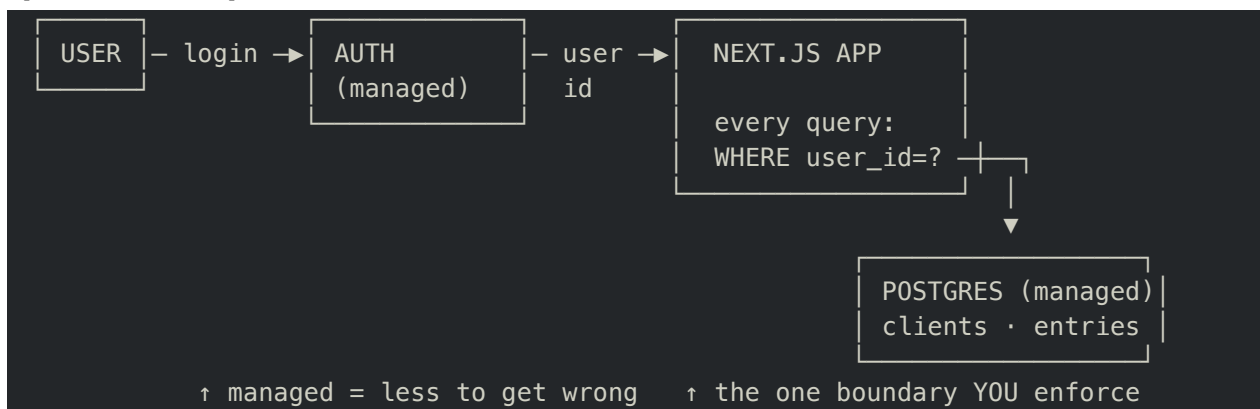
```
A logged-in web app where a user can: sign up / log in with email,
create clients, log time entries (date, client, hours, note),
see a table of entries filtered by month, and download that month
as CSV. Each user only ever sees their own data. Mobile-friendly.
```

ประโยค “Each user only ever sees their own data” (ผู้ใช้แต่ละคนเห็นได้เฉพาะข้อมูลของตัวเองเท่านั้น) ดูเหมือนจะเป็นแค่เรื่องประสบการณ์ผู้ใช้ (UX) แต่จริง ๆ แล้วมันคือแบบจำลองความปลอดภัยของทั้งแอปที่ถูกอัดแน่นอยู่ในเก้าอี้ คำการระบุมันไว้ในสเปกทำให้เราสามารถซึกกลับไปทีประโยคนี้ได้ทุกครั้งที่ AI เริ่มหลุดออกนอกเส้นทาง

สแต็กที่เลือกใช้

สำหรับคนที่สร้างแอปคนเดียว ทำที่ชนะคือการเลือกสแต็กที่ระบบล็อกอินและฐานข้อมูลเป็นบริการที่มีคนดูแลให้ ไม่ใช่โค้ดที่คุณต้องเขียนเอง เราเลือกแอป Next.js ที่ดีพลอยบนโฮสต์แบบ serverless พร้อมฐานข้อมูล Postgres แบบมีคนดูแล และผู้ให้บริการล็อกอินสำเร็จรูปที่จัดการเรื่องล็อกอินด้วยอีเมล เซสชัน และการรีเซตรหัสผ่านให้เราทั้งหมด โค้ดที่ต้องเขียนน้อยลงก็หมายถึงโค้ดที่ AI จะทำพลาดแทนเราได้น้อยลงตามไปด้วย ระบบล็อกอินโดยเฉพาะเป็นหมวดที่แทบไม่ควรให้ AI เขียนเองตั้งแต่ต้นเลย เพราะรูปแบบความล้มเหลวมันเจียบเจิบ รัศมีความเสียหายคือบัญชีของทุกคน และผู้ให้บริการแบบมีคนดูแลก็ผ่านการเจอกรณีสุดขั้วมานับล้าน ๆ ครั้งของการล็อกอินมาแล้ว

ส่วนที่มีคนดูแลอยู่รอบนอก ส่วนกฎเดียวที่เราเป็นเจ้าของอยู่ตรงกลาง นั่นคือทุกคิวรีต้องกรองด้วย user_id เพื่อไม่ให้ผู้ใช้คนหนึ่งอ่านข้อมูลของอีกคนได้เด็ดขาด:



พรอมต์สำคัญ

เราให้เทมเพลตของผู้ให้บริการล็อกอินทำงานหนักส่วนใหญ่ให้ แล้วสั่งให้ AI เพิ่มตรรกะเฉพาะทางธุรกิจลงไปทับ:

```
We're using [auth provider]'s Next.js starter. Add a Postgres
schema with two tables: clients (id, user_id, name) and
time_entries (id, user_id, client_id, date, hours, note). Every
query MUST filter by the logged-in user's id from the session.
Generate the migration and the typed data-access functions.
```

บรรทัด “MUST filter by user_id” คือประโยคที่สำคัญที่สุดในโปรเจกต์ทั้งหมด เราย้ำเงื่อนไขนี้ในแทบทุกพรอมต์ที่แตะต้องข้อมูล เพราะบีกที่หนักที่สุดในแอปหลายผู้ใช้เพียงหนึ่งเดียวคือการที่ผู้ใช้คนหนึ่งเห็นแถวข้อมูลของอีกคน การพูดซ้ำ ๆ อาจรู้สึกว่ามันเพื่อยตอนพิมพ์ แต่นั่นแหละคือความเพื่อยที่ช่วยชีวิตคุณไว้ เพราะโมเดลไม่มีความจำว่าเงื่อนไขสำคัญแค่ไหนข้ามพรอมต์แต่ละครั้ง

สำหรับการส่งออกข้อมูล:

```
Add a /api/export route that takes a month (YYYY-MM), pulls the logged-in user's time_entries for that month joined to client names, and streams a CSV download. Reject the request if there's no valid session.
```

อุปสรรค

ระหว่างการทดสอบ เราสร้างสองบัญชีขึ้นมา และพบว่าบัญชี B สามารถเห็นรายชื่อลูกค้าของบัญชี A ในครอบครองได้ นี่คือการที่เรากลัวมากที่สุดพอดี แทนที่จะขอให้ AI “แก้ไขหน่อย” เราให้มันพิสูจน์ปัญหาก่อน:

```
Account B is seeing Account A's clients. Show me every database query in the codebase that reads the clients table, and for each one tell me whether it filters by the session user_id. Don't fix anything yet – just audit.
```

การตรวจสอบพบคิวรีหนึ่งตัว คือตัวโหลดครอบครอง ที่ถูกเขียนขึ้นก่อนที่เราจะเพิ่มเงื่อนไข และหลุดรอดผ่านมาได้ เราให้ AI เพิ่มฟิลเตอร์ที่ขาดหายไป แล้วขอให้มันสร้างตัวป้องกันเพิ่มเติม:

```
Add a single helper that every read goes through, which takes the session and injects the user_id filter, so no future query can forget it. Refactor the existing queries to use it.
```

นั่นเปลี่ยนการแก้ไขแบบครั้งเดียวจบให้กลายเป็นหลักประกันเชิงโครงสร้าง จากนั้นเราทำให้หลักประกันนี้ทดสอบได้จริง เพราะการวัดที่ตรวจสอบไม่ได้ก็เป็นแค่ความหวังลม ๆ แล้ง ๆ:

```
Write a test that creates two users, has each create a client, then asserts that user A's session can never read user B's client through any of the data-access functions.
```

บทเรียน: เมื่อ AI สร้างบักด้านความปลอดภัยขึ้นมา อย่าแค่แก้จุดนั้นจุดเดียว แต่สั่งให้มันกำจัด หมาดหมู ของความผิดพลาดนั้นทั้งหมด แล้วล็อกหมาดหมูนั้นไว้ด้วยเทสต์ที่จะฟังเสียงดังทันทีถ้าใครเผลอเปิดช่องนั้นขึ้นมาใหม่

การเปิดตัว

เราใส่ข้อมูลทดสอบของหนึ่งเดือน ส่งออกเป็น CSV แล้วเปิดในสเปรดชีตเพื่อยืนยันว่าตัวเลขและการเข้ารหัสถูกต้อง จากนั้นตั้งรหัสผ่านฐานข้อมูลให้แข็งแรง และหมุนเวียนข้อมูลรับรองออกจากไฟล์ในเครื่องทุกไฟล์ เราตีพลอยไปที่โฮสต์แบบ serverless เพิ่มตัวแปรสภาพแวดล้อมสำหรับใช้งานจริงในแดชบอร์ดของมัน แล้วส่ง URL ให้เธอ เธอสมัครใช้งานเองด้วยการล็อกอินจริง โปรเจกต์ทั้งหมดใช้เวลาแค่หนึ่งสัปดาห์

กรณีศึกษาที่ 3: เครื่องมือทำงานอัตโนมัติส่วนตัว

ไอดี

โปรเจกต์เพื่อความสะดวกส่วนตัวล้วน ๆ สคริปต์ที่ทุกเช้าจะดึงกิจกรรมในปฏิทินของวันนั้นและสภาพอากาศ จัดรูปแบบเป็นสรุปหนึ่งบรรทัด แล้วส่งไปยังแชท Telegram ส่วนตัว จุดประสงค์คือเพื่อไม่ต้องเปิดสามแอปก่อนดื่มกาแฟทุกเช้า

สเปก

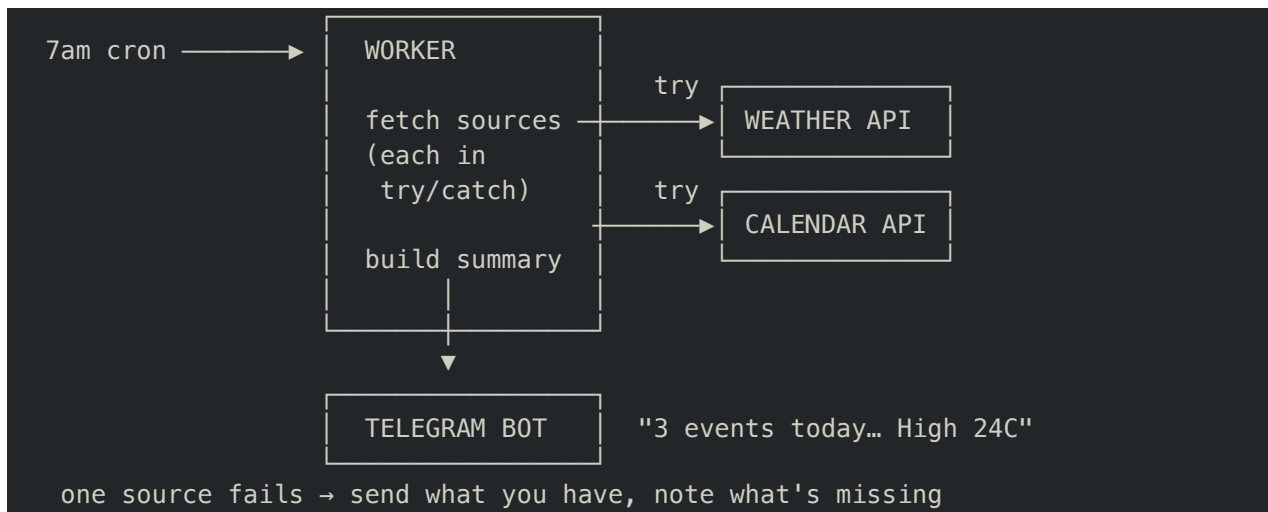
```
A scheduled job that runs at 7am my timezone. It reads today's events from my calendar, fetches the forecast for my city, and sends one Telegram message like: "3 events today, first at 9:30. High 24C, rain after 4pm." If any source fails, still send what you have and note what's missing.
```

ประโยชน์สุดท้าย ที่บอกให้ทำงานแบบ “ลดระดับอย่างสง่างาม” แทนที่จะพังไปเลย เป็นพฤติกรรมแบบที่คุณต้องขอให้ชัดเจน AI จะไม่คาดเดามันเอง ถ้าไม่พูดออกมา คำเริ่มต้นที่โมเดลมักเลือกใช้คือ “โยน error ออกไปเลย” ซึ่งสำหรับงาน cron ตอนตีเจ็ดแล้ว หมายถึงเช้าเจียบ ๆ ที่ไม่มีข้อความอะไรมาเลย และไม่มีเบาะแสว่าทำไม

สแต็กที่เลือกใช้

ฟังก์ชัน serverless แบบมีตารางเวลาเดียว (Worker ที่ทริกเกอร์ด้วย Cron) เพื่อไม่ต้องมีเครื่องอะไรที่ต้องคอยเปิดทิ้งไว้ ทั้งหมดคือไฟล์เดียวบวกกับตารางเวลาหนึ่งอัน สำหรับเครื่องมือส่วนตัว เรื่องนี้สำคัญกว่าที่ฟังดู เพราะอะไรก็ตามที่มีเซิร์ฟเวอร์ต้องคอยดูแลเป็นสิ่งที่ คุณจะลืมจ่ายเงินให้ในที่สุด และโปรเจกต์ก็จะตายเพราะถูกทอดทิ้ง ไม่ใช่เพราะความล้มเหลว

ทริกเกอร์หนึ่งตัวกระจายไปสองแหล่งข้อมูล แล้วรวมกลับมาเป็นข้อความเดียว try/catch รอบแต่ละแหล่งข้อมูลคือสิ่งที่ทำให้ความล้มเหลวจุดเดียวลดระดับอย่างสง่างามแทนที่จะทำให้การรันทั้งหมดตายไปด้วย:



พรอมต์สำคัญ

```
Write a Cloudflare Worker triggered by cron at 7am Asia/Seoul.
It calls a weather API and a calendar API (I'll provide both
tokens as secrets), builds a one-line summary, and POSTs it to
the Telegram Bot API. Wrap each external call in try/catch so one
failure doesn't kill the message. Read every token from env.
```

พอเราอยากปรับแต่งถ้อยคำโดยไม่ต้องรอถึงตีเจ็ด:

```
Add a manual trigger: if the Worker is hit with a GET request and
a ?test=1 query param, run the same logic immediately and return
the message text in the response instead of sending it. Keep this
behind a secret token so only I can trigger it.
```

ตัวทริกเกอร์ทดสอบนี้ช่วยเราหนีจากวงจรฟีดแบ็กที่โหดร้ายไปได้ เราสามารถเห็นผลลัพธ์ได้ตามต้องการ แทนที่จะได้เห็นแค่วันละครั้ง นี่เป็นท่าทางทั่วไปที่ควรขโมยไปใช้ เมื่อไรก็ตามที่โค้ดของคุณรันตามตารางเวลา ให้สร้างวิธีที่จะส่งให้มันรัน เดี๋ยวนี้ ตามต้องการ ก่อนที่คุณจะต้องรอตามนาฬิกา

อุปสรรค

ข้อความ Telegram มาถึงแล้ว แต่เวลาผิด กิจกรรมแสดงเป็นเวลา UTC ไม่ใช่เวลาเกาหลี ส่วนสภาพอากาศถูกต้องดี เราตีกรอบปัญหาให้แคบลง:

```
Calendar times are 9 hours off – showing UTC, not Asia/Seoul.
The weather time ("rain after 4pm") is correct. Here's how I
format both: [pasted]. Fix only the calendar formatting and tell
me why the weather path was already right.
```

AI อธิบายว่า weather API (Application Programming Interface — ประตูละการหนึ่งเปิดไว้ให้โปรแกรมอื่นดึงข้อมูลของมันไปใช้) ส่งค่ากลับมาเป็นสตริงที่แปลงเป็นเวลาท้องถิ่นแล้วอยู่แล้ว ในขณะที่ calendar API ส่งกลับเป็น timestamp แบบ UTC ที่เรานำมาแสดงตรง ๆ โดยไม่แปลง มันเพิ่มการแปลงเขตเวลาเข้าไปหนึ่งจุดตรงขั้นตอนจัดรูปแบบปฏิทิน การจำกัดขอบเขตของการแก้ไขให้เหลือแค่ “เฉพาะเส้นทางปฏิทินเท่านั้น” หยัดไม่ให้มันไปเขียนโค้ด weather ที่ใช้งานได้ดีอยู่แล้วซ้ำ ซึ่งเป็นวิธีที่ AI มักทำพั้งในสิ่งที่เคยดีอยู่แล้ว ส่วนครึ่งหลังของพรวดนั้น ที่บอกว่า “tell me why the weather path was already right” ทำหน้าที่สองอย่างพร้อมกัน คือให้สาเหตุที่แท้จริงแก่เราแทนที่จะเป็นการเดา และบังคับให้ AI ต้องอธิบายความแตกต่างของทั้งสองเส้นทาง เพื่อไม่ให้มันผลอ “ทำให้เหมือนกัน” ทั้งสองเส้นทางจนกลายเป็นพั้งทั้งคู่โดยไม่ตั้งใจ

การเปิดตัว

“การเปิดตัว” ในที่นี้หมายถึงแค่การดีพลอย Worker ตั้งตารางเวลา cron และเพิ่มโทเคน API เป็นความลับแบบเข้ารหัส เรายิงเข้าเอนด์พอยต์ ?test=1 สองสามครั้งเพื่อยืนยันว่าข้อความอ่านลื่นดี แล้วก็ปล่อยให้มันทำงานไป มันรันทุกเช้ามาตั้งแต่นั้น และบรรทัดที่บอกให้ลดระดับอย่างสง่างามก็พิสูจน์คุณค่าของมันในอีกหนึ่งเดือนต่อมา เมื่อ weather API ล่ม แต่ข้อความก็ยังส่งมาถึงอยู่ดี เพียงแต่ไม่มีพยากรณ์อากาศเท่านั้น

กรณีศึกษาที่ 4: เว็บไซต์คอนเทนต์ที่ไม่ต้องคอยดูแล

ไอดี

นักเขียนงานอดิเรกคนหนึ่งอยากเปลี่ยนโพลเดอร์ Markdown บันทึกลงของเขา ซึ่งเป็นรีวิวร้านอาหารจากหลายปีที่เดินทางไปมา ให้กลายเป็นเว็บไซต์สาธารณะที่ค้นหาได้ โดยไม่ต้องมี CMS ให้ล็อกอิน และไม่มีค่าใช้จ่ายรายเดือน แค่เพิ่มบันทึกใหม่ push ขึ้นไป จบ

สเปก

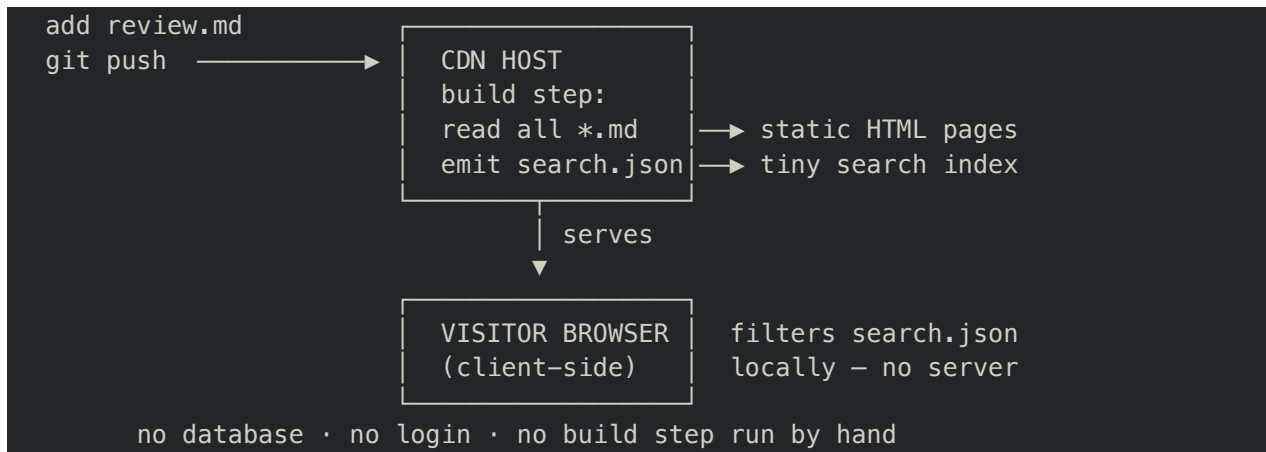
```
A static site generated from a folder of Markdown files, one file
per review (title, city, rating, body in frontmatter). Build a
homepage that lists reviews newest-first, a page per review, and
client-side search by city or name. No database, no login, no
build step I have to run by hand. Pushing a new .md file should
publish it.
```

ข้อกำหนดที่ซ่อนอยู่คือประโยคสุดท้าย “No build step I have to run by hand” หมายความว่าสเปกนี้จริง ๆ แล้วพูดถึงเวิร์กโฟลว์ ไม่ใช่แค่ผลลัพธ์เท่านั้น และข้อกำหนดด้านเวิร์กโฟลว์เป็นสิ่งที่ AI มักข้ามไปเว้นแต่คุณจะระบุมันไว้ชัดเจน เพราะมันไม่ปรากฏให้เห็นในตัวแอปที่รันอยู่

สแต็กที่เลือกใช้

ตัวสร้างเว็บไซต์สแตติกที่อ่าน Markdown ตอน build time ดีพลอยไปยังโฮสต์ CDN ที่ build อัตโนมัติเมื่อมีการ push เข้า Git push เข้า repo โฮสต์ก็ build ใหม่ CDN ก็เลิฟให้ ส่วนการค้นหาทำงานฝั่ง client บน JSON index เล็ก ๆ ที่สร้างขึ้นตอน build จึงไม่มีเซิร์ฟเวอร์และไม่มีค่าใช้จ่ายจากการคิวรี ซึ่งเป็นข้อแลกเปลี่ยนที่ดีสำหรับรีวิวสักไม่กี่ร้อยรายการ และเป็นสิ่งที่เราตั้งใจระบุไว้ตั้งแต่แรก ไม่ใช่ค้นพบทีหลัง

เวิร์กโฟลว์ทั้งหมดเป็นลูกศรเดียว การ push จะทริกเกอร์การ build ที่เผยแพร่เว็บไซต์ ไม่มีอะไรทำงานตอน request time เลย นอกจากเบราว์เซอร์ของผู้เข้าชมเองที่กรอง index:



พรอมต์สำคัญ

เรากำหนดให้ AI ยึดตามรูปแบบข้อมูลก่อนเป็นอันดับแรก ก่อนที่จะแตะต้อง UI เลย:

```
Set up a static site that reads every .md file in /reviews. Each file's frontmatter has title, city, rating (1-5), date. Parse all of them at build time into a sorted list (newest first) and generate: a homepage listing them, and one page per review at /reviews/<slug>. Show me the data-loading code first, before any styling.
```

“Before any styling” (ก่อนใส่สไตล์ใด ๆ) เป็นคั่นโยกที่จงใจดึงใช้ ถ้าคุณขอทุกอย่างพร้อมกันทีเดียว คุณจะได้น้ำเว็บที่สวยงามแต่เชื่อมกับข้อมูลที่โง่โง่ไม่ได้ แต่ถ้าขอขั้นข้อมูลก่อน คุณจะตรวจสอบรากฐานได้ก่อนที่ฟิกเชลแม้แต่ว่าเดียวจะมาถึงความสนใจไป จากนั้นจึงเป็นเรื่องการค้นหา ที่ดีกรอบให้ยังคงประหยัดต้นทุน:

```
Generate a search.json at build time with {title, city, slug} for every review. On the homepage, add a search box that filters the visible list client-side by matching city or title – no network calls, just filter the already-loaded index. Keep it under 50 lines of JS.
```

อุปสรรค

การ build ผ่านฉลุยตอนรันในเครื่อง แต่เว็บไซต์ที่ดีพลอยไปแล้วขึ้น 404 ในทุกหน้ารีวิวแต่ละรายการ หน้าแรกทำงานปกติ แต่หน้ารีวิวรายตัวไม่ทำงาน เราวางอาการที่เกิดขึ้นและการตั้งค่าไปให้ดู แทนที่จะทฤษฎีเดาไปเอง:

```
Homepage works live, but /reviews/<slug> pages all 404 in production while working in local dev. Here's my build output directory listing and my host's deploy config: [pasted]. Why does local dev serve these pages but production doesn't?
```

```
The AI traced it to the difference between dev-server routing (which resolves paths dynamically) and static hosting (which serves only files that physically exist). The build was generating the review pages into the wrong output folder, so they never got uploaded – dev had hidden the bug because its router faked the routes that production couldn't.
```

เราชี้โฟลเดอร์ output ให้ตรงกับสิ่งที่โฮสต์ดีพลอยจริง หน้าเว็บก็ปรากฏขึ้นมา และเราได้เพิ่มอีกหนึ่งบรรทัดในเช็กलिสด์

การเปิดตัว นั่นคือ คลินิกคลิกบนเว็บไซต์จริง ไม่ใช่แค่หน้าแรก บทเรียนที่กว้างกว่านั้นคือ “ใช้งานได้นอนพัฒนา” กับ “ใช้งานได้นอนดีพลอยจริง” เป็นข้อสรุปคนละเรื่องกัน และ static export คือจุดที่มันแยกทางกันชัดเจนที่สุด เซิร์ฟเวอร์สำหรับพัฒนามักจะยอมความกับเรื่อง routing ในแบบที่การโฮสต์ไฟล์จริงไม่มีวันยอม

การเปิดตัว

เรา push รีวิวจริงเข้าไปหนึ่งรายการ ดูโฮสต์ build ใหม่ แล้วโหลดลิงก์คลิกไปยังรีวิวนั้นบนโดเมนจริง ซึ่งเป็นกรณีเดียวกับที่ dev เคยโกหกเราไว้พอดี จากนั้นเราค้นหาชื่อเมืองของมันเพื่อยืนยันว่า index จับมันไว้แล้ว การเพิ่มรีวิว ถัดไปเป็นแค่คอมมิตหนึ่งบรรทัด ซึ่งคือประเด็นทั้งหมดของเรื่องนี้ การเปิดตัวไม่ใช่โมเมนต์เดียว แต่เป็นเวิร์กโฟลว์ที่ทำงานต่อไปได้เองโดยไม่ต้องมีเราคอยดูแล

สิ่งที่ทั้งสักรณีนี้อาจมีร่วมกัน

- สเปกมาก่อนเสมอ และมันสั้น เป็นแค่หนึ่งย่อหน้า ไม่ใช่เอกสารยาว ๆ แต่มันระบุข้อจำกัดที่สำคัญเอาไว้ ไม่ว่าจะเป็นภาษา “เห็นได้เฉพาะข้อมูลของตัวเองเท่านั้น” “ยังต้องส่งสิ่งที่มีอยู่” “push ไฟล์แล้วต้องเผยแพร่ได้เลย” รวมถึงข้อจำกัดด้านเวิร์กโฟลว์ที่ไม่เคยปรากฏในตัวแอปที่รันอยู่
- ความลับไม่เคยแตะต้อง repo เลย ทุกโปรเจกต์อ่านคีย์จากตัวแปรสภาพแวดล้อมหรือความลับที่เข้ารหัส ซึ่งตั้งค่าไว้ในแดชบอร์ด ไม่ใช่ในโค้ด เราพูดคำว่า “never hardcode it” ออกมาดัง ๆ เพราะค่าเริ่มต้นมักจะเป็นสิ่งตรงกันข้าม
- สดเด็กถูกเลือกเพื่อให้มีสิ่งที่ทำพลาดได้น้อยลง ระบบล็อกอินแบบมีคนดูแล ฐานข้อมูลแบบมีคนดูแล การโฮสต์แบบสแตติก ฟังก์ชัน serverless แต่ละตัวเลือกลดความยุ่งยากของโค้ดที่ AI จะทำพังแทนเราได้ และลดความยุ่งยากของสิ่งที่เราจะต้องดูแลรักษาเอง
- อุปสรรคถูกแก้ไขด้วยการป้อนหลักฐานที่แม่นยำให้ AI ทั้งข้อความ error ตัวจริง โค้ดจริง การตั้งค่าดีพลอย อาการที่เกิดขึ้นจริง และด้วยการจำกัดขอบเขตการแก้ไขให้แคบ เพื่อให้โค้ดที่ใช้งานได้ยังคงใช้งานได้ต่อไป มีสองครั้งที่อาการอยู่ในโค้ด แต่สาเหตุอยู่ที่โครงสร้างพื้นฐาน
- การเปิดตัวหมายถึงการทดสอบจริงกับความเป็นจริง ไม่ว่าจะเป็นการซื้อจริง (และการซื้อที่ถูกปฏิเสธ) บัญชีที่สอง ทริกเกอร์ตามต้องการ ลิงก์คลิกบนเว็บไซต์ที่ดีพลอยจริง ไม่ใช่แค่ “มันคอมไพล์ผ่าน”

คุณไม่จำเป็นต้องมีไอเดียใหญ่โตเพื่อเริ่มต้น คุณแค่ต้องมีไอเดียเล็ก ๆ หนึ่งอัน สเปกหนึ่งย่อหน้า และความเต็มใจที่จะส่งงานแทนที่จะพิมพ์เอง พร้อมกับความอดทนที่จะทดสอบมันกับความเป็นจริงก่อนที่จะประกาศว่าเสร็จแล้ว

สรุปและแบบฝึกหัด

ประเด็นสำคัญ

- สเปกมาก่อนเสมอ และมันสั้น เป็นย่อหน้าเดียวที่ระบุข้อจำกัดที่สำคัญเอาไว้ รวมถึงข้อจำกัดด้านเวิร์กโฟลว์ที่ไม่เคยปรากฏในตัวแอปที่รันอยู่
- ความลับไม่เคยแตะต้อง repo เลย ทุกโปรเจกต์อ่านคีย์จากตัวแปรสภาพแวดล้อมหรือค่าคอนฟิกที่เข้ารหัสซึ่งตั้งค่าไว้ในแดชบอร์ด
- สดเด็กถูกเลือกเพื่อให้มีสิ่งที่ทำพลาดได้น้อยลง ระบบล็อกอินแบบมีคนดูแล ฐานข้อมูลแบบมีคนดูแล การโฮสต์แบบสแตติก ฟังก์ชัน serverless
- อุปสรรคถูกแก้ไขด้วยการป้อนหลักฐานที่แม่นยำให้ AI ทั้งข้อความ error ตัวจริง โค้ดจริง การตั้งค่าดีพลอย และการจำกัดขอบเขตการแก้ไขให้แคบ
- การเปิดตัวหมายถึงการทดสอบจริงกับความเป็นจริง ไม่ใช่แค่ “มันคอมไพล์ผ่าน” ไม่ว่าจะเป็นการซื้อจริง บัญชีที่สอง ลิงก์คลิกบนเว็บไซต์ที่ดีพลอยจริง

ลองทำดู

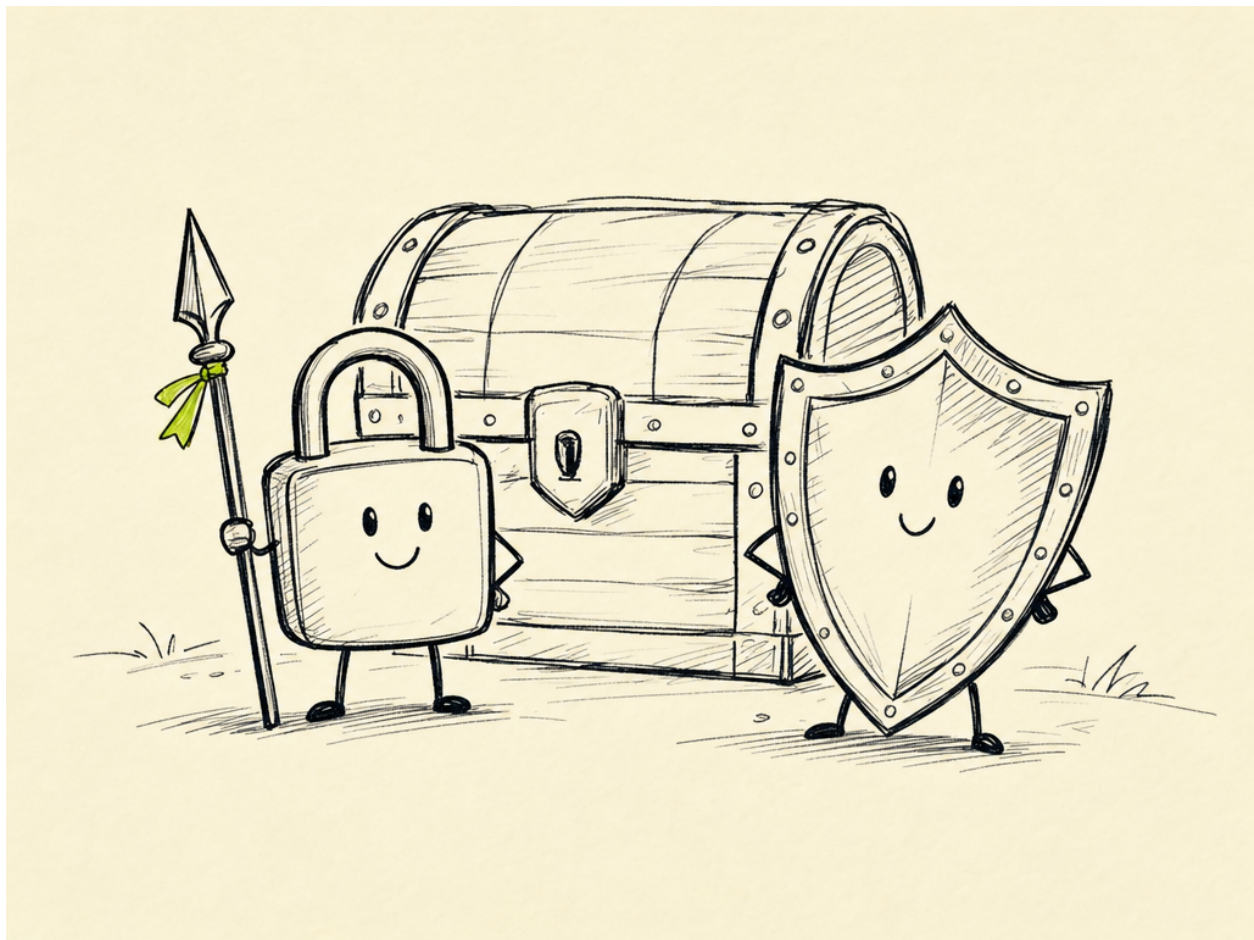
เลือกไอเดียของคุณเองสักหนึ่งอัน แล้วเขียนสเปกหนึ่งย่อหน้าแบบที่ทั้งสักรณีนี้อาจมีร่วมกัน ทำรายการกระทำหลักของแอป ข้อมูลที่ผู้ใช้แต่ละคนมองเห็นได้ และข้อจำกัดด้านเวิร์กโฟลว์อย่างน้อยหนึ่งข้อ (เนื้อหาถูกเผยแพร่ได้อย่างไร ใครทำอะไรได้

บ้าง) จากนั้นลิสต์บล็อกส่วนประกอบแบบมีคนดูแลที่คุณจะพึ่งพา เพื่อให้มีโค้ดที่เขียนเองน้อยลงที่อาจพัง จำกัดให้อยู่แค่หนึ่งย่อหน้าบวกกับลิสต์บัลเล็ตสั้น ๆ ถ้ามันยาวเกินกว่านั้น แปลว่าคุณกำลังบรรยายละเอียดมากเกินไป

พร้อมต์ประจำบท

```
I want to build: [one-paragraph description of the app].  
Before any code, help me tighten this into a buildable spec.  
1. Restate the single core action a user performs.  
2. List the data each user can see – and what they must NOT see.  
3. Name the workflow constraints that won't show up in the UI  
   (e.g. "pushing a file should publish it", "only their own data").  
4. Recommend a stack that MINIMIZES custom code: managed auth,  
   managed database, static or serverless hosting where possible.  
5. Flag the one part most likely to go wrong, and how we'll test it  
   against reality (a real purchase, a second account, a deep link).  
Keep the spec to a paragraph plus bullets. Push back if it's too big.
```

ความปลอดภัยสำหรับนัก Vibe Coding



ความปลอดภัยเป็นส่วนหนึ่งของ vibe coding ที่กีดกั้นคุณอย่างเจ็บปวด ๆ ฟีเจอร์ที่ฟังจะแสดงตัวออกมาทันที — ปุ่มกดไม่ทำงาน คุณก็แก้ไป แต่ช่องโหว่ด้านความปลอดภัยจะไม่แสดงตัวเลย จนกว่าจะถึงวันที่มีใครสักคนพบมันและนำข้อมูลผู้

ใช้ของคุณไปโพสต์ในฟอรัม ถึงตอนนั้น AI ที่เขียนบักตัวนี้ไปนานแล้ว และคุณคือคนที่ต้องอธิบายให้คนนับพันฟังว่าทำไมรหัสผ่านของพวกเขาถึงรั่วไหล

บทนี้ไม่ได้พูดถึงเรื่องความกลัว คุณไม่จำเป็นต้องกลายเป็นนักทดสอบเจาะระบบ (penetration tester) หรือท่องจำ OWASP Top 10 เพื่อจะสร้างแอปเล็ก ๆ ที่ปลอดภัยได้ สิ่งที่คุณต้องมีคือความเข้าใจในวิธีการไม่กี่แบบที่ซอฟต์แวร์มักถูกเจาะเข้าไป รู้จักรูปแบบที่ AI มักผลิตออกมาซ้ำ ๆ และเพิ่มขั้นตอนตรวจสอบหนึ่งจุดก่อนที่คุณจะปล่อยงาน ปรัชญาก็เหมือนกับส่วนอื่นของหนังสือเล่มนี้: อย่าปล่อยสิ่งที่คุณไม่เข้าใจออกไป — และความปลอดภัยคือจุดที่การไม่เข้าใจนั้นมีต้นทุนแพงที่สุด

ทำไมโค้ดที่ AI สร้างจึงไม่ปลอดภัยโดยค่าเริ่มต้น

AI จะปรับให้เหมาะกับพรมพื้ที่คุณให้ไป และพรมพื้ของคุณแทบทุกครั้งจะบอกว่า “ทำให้มันทำงานได้” ไม่ใช่ “ทำให้มันปลอดภัย” ดังนั้นมันจึงสร้างเส้นทางที่สั้นที่สุดไปสู่พีเจอร์ที่ใช้งานได้ เส้นทางที่สั้นที่สุดมักจะเป็นเส้นทางที่ไม่ปลอดภัย เช่น การต่อสตริงเพื่อสร้างคิวรี (string-concatenated queries), เอนด์พอยต์ (endpoint หนึ่งในประตูที่เข้าถึงได้ของแอปคุณ — URL เดียวที่แอปตอบรับคำขอ) ที่ไม่มีการตรวจสอบสิทธิ์ หรือความลับที่ถูกวางแทรกตรง ๆ เพราะนั่นคือตัวอย่างที่มันเรียนรู้มา

สถานการณ์ยิ่งแย่ลงไปอีก เพราะ AI เรียนรู้จากอินเทอร์เน็ตทั้งหมด — รวมถึงบทเรียนสอนโค้ดและโปรเจกต์ที่ถูกทิ้งร้างนับสิบปีที่ตัวมันเองก็ไม่ปลอดภัยอยู่แล้ว โค้ดที่ไม่ปลอดภัยพบได้บ่อยกว่า โค้ดที่ปลอดภัยในข้อมูลฝึกฝนของมัน เพราะโค้ดที่ปลอดภัยเขียนยากกว่าและถูกโพสต์น้อยกว่า เมื่อคุณขอ “แบบฟอร์มล็อกอิน” ในทางสถิติแล้วคุณมักจะได้แบบฟอร์มล็อกอินโดยเจลีย์ของอินเทอร์เน็ต และแบบฟอร์มล็อกอินโดยเจลีย์ของอินเทอร์เน็ตก็มีปัญหา

AI ยังไม่คำนึงถึงภัยคุกคามด้วย มนุษย์ที่กำลังสร้างพีเจอร์อัปโหลดไฟล์จะรู้สึกกังวลวูบหนึ่ง — ถ้ามีคนอัปโหลดอะไรที่ไม่ดีล่ะ? แต่ AI ไม่รู้สึกอะไรเลย มันเขียนตัวจัดการ (handler) ด้วยความมั่นใจเท่ากันไม่ว่าจะรัดกุมหรือเปิดโล่ง ไม่มีการลังเล ไม่มี “อืม ส่วนนี้ทำให้ผมกังวลนะ” คุณต้องเป็นคนใส่ความกังวลนั้นเข้าไปเอง

บทสรุปไม่ใช่ “โค้ด AI อันตราย อย่าใช้มัน” แต่คือ “มันทำงานได้” กับ “มันปลอดภัย” เป็นคำถามสองข้อที่ต่างกัน และ AI จะตอบแค่ข้อแรกเท่านั้น เว้นแต่คุณจะบังคับให้มันตอบข้อที่สองด้วย

Injection: เมื่อข้อมูลนำเข้ากลายเป็นโค้ด

Injection เป็นบักร้ายแรงที่เก่าแก่ที่สุดและยังคงพบบ่อยที่สุด มันเกิดขึ้นเมื่อข้อมูลจากผู้ใช้ถูกปฏิบัติเหมือนเป็น คำสั่งแทนที่จะเป็น ข้อมูล มีสองรูปแบบที่สำคัญสำหรับแอปส่วนใหญ่

SQL injection — SQL (Structured Query Language) คือภาษาที่แอปใช้ขอข้อมูลจากฐานข้อมูล — เกิดขึ้นเมื่อข้อมูลนำเข้าจากผู้ใช้แอบแทรกเข้าไปในคิวรีฐานข้อมูล รูปแบบคลาสสิกที่มีช่องโหว่:

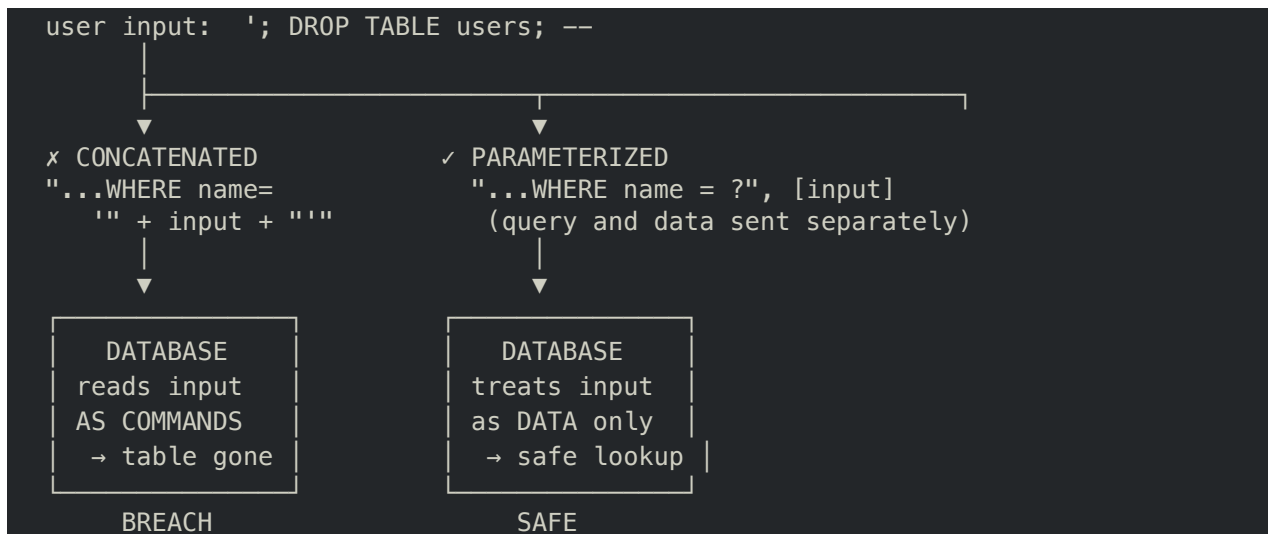
```
// VULNERABLE: the user's input is glued straight into the query
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query(`SELECT * FROM users WHERE name = '${name}'`);
});
// If someone passes name = '; DROP TABLE users; --
// your query becomes a command to delete the table.
```

วิธีแก้คือ parameterized query (หรือเรียกอีกอย่างว่า prepared statement) คุณส่งคิวรีและข้อมูลแยกกัน เพื่อให้ฐานข้อมูลไม่มีทางสับสนระหว่างทั้งสองอย่าง:

```
// SAFE: the value is passed as a parameter, never as code
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query("SELECT * FROM users WHERE name = ?", [name]);
});
```

กฎคือ: อย่าสร้างคิวรีด้วยการต่อสตริงเด็ดขาด ถ้าคุณเห็นแบ็กติกหรือ + กำลังประกอบ SQL รอบตัวแปรในดิฟฟ์ ให้หยุดและขอเวอร์ชัน parameterized

ทั้งสองเส้นทางนำข้อมูลนำเข้าของผู้ใช้เดียวกันเข้าสู่ฐานข้อมูล แต่มีเพียงเส้นทางเดียวเท่านั้นที่ป้องกันไม่ให้ฐานข้อมูลเข้าใจผิดว่าข้อมูลนั้นเป็นคำสั่ง:



XSS (cross-site scripting) ก็เป็นแนวคิดเดียวกันแต่เกิดในเบราว์เซอร์ ถ้าคุณนำข้อมูลนำเข้าจากผู้ไปวางลงในหน้าเว็บเป็น HTML ดิบ (HyperText Markup Language คือโค้ดที่บอกเบราว์เซอร์ว่าจะแสดงผลอะไร) ผู้โจมตีสามารถแทรกแท็ก `<script>` ที่ทำงานในเบราว์เซอร์ของผู้ใช้คนอื่นได้ — เช่น ขโมยเซสชันของพวกเขา วิธีแก้คือการ escape: แสดงเนื้อหาของผู้ใช้เป็น ข้อความ ไม่ใช่ HTML เฟรมเวิร์กสมัยใหม่อย่าง React จะ escape ให้โดยค่าเริ่มต้นซึ่งดีมากจนกว่า AI จะหยิบทางลัดอย่าง `dangerouslySetInnerHTML` หรือ `innerHTML` มาใช้เพื่อ “ทำให้มันทำงานได้” สิ่งเหล่านั้นจะข้ามการป้องกันไป ให้ตั้งข้อสงสัยกับการเรียกใช้แบบนี้ทุกครั้งที่พบในดิฟฟ์

หลักการรวมเบื้องหลังบั๊ก injection ทุกแบบคือ: แยกข้อมูลกับโค้ดออกจากกัน เมื่อใดที่ข้อมูลนำเข้าของผู้ใช้ข้ามเข้าไปในคิวรี คำสั่ง เทมเพลต หรือ HTML จะต้องมีการ escape หรือ parameterize มันเสมอ

ความลับและ API key: กับดักคีย์ที่ถูกเปิดเผย

ความลับ (secret) คือสิ่งใดก็ตามที่ให้สิทธิ์เข้าถึง เช่น API (Application Programming Interface) key รหัสผ่านฐานข้อมูล โทเคนของผู้ให้บริการชำระเงิน คีย์สำหรับเซ็นชื่อดิจิทัล ความผิดพลาดที่พบบ่อยที่สุด — และแพงที่สุด — ของการทำ vibe coding คือการทำความลับรั่วไหล

มีกับดักสองแบบที่ตักคนไว้อยู่เสมอ:

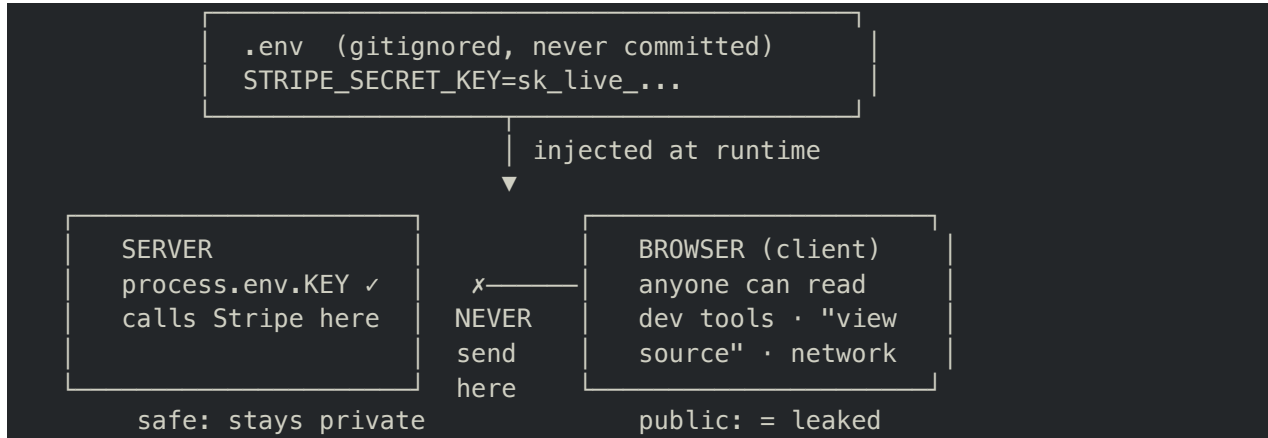
- ความลับในโค้ดฝั่งไคลเอนต์ อะไรก็ตามที่อยู่ใน frontend ของคุณ (JavaScript ที่ทำงานในเบราว์เซอร์) ถือว่าเปิดเผยต่อสาธารณะ ผู้ใช้สามารถเปิด dev tools แล้วอ่านมันได้ AI เมื่อถูกขอให้เรียก API จากคอมโพเนนต์ React มันจะยีนดิวาง secret key ของคุณตรงนั้นเลย — และตอนนี้ใครก็ตามที่เข้าเว็บไซต์ของคุณก็สามารถคัดลอกมันไปใช้จนบิลคุณพุ่งได้ Secret key ควรอยู่ที่ฝั่งเซิร์ฟเวอร์เท่านั้น ไม่ใช่ในโค้ดที่ถูกส่งไปยังเบราว์เซอร์
- ความลับในรีพอ คีย์ที่ถูกฝังตายตัวในไฟล์จะถูก commit เข้า git แม้ว่า你会ลบมันในภายหลัง มันก็ยังคงอยู่ในประวัติของ git ตลอดไป และบอตต่าง ๆ ก็สแกนหาทรัพย์สินสาธารณะเพื่อหาสิ่งนี้ภายในไม่กี่นาทีหลังจาก push

บ้านที่ปลอดภัยสำหรับความลับคือ environment variable — ค่าที่ถูกส่งให้แอปตอนรันใหม่ โดยไม่อยู่ในโค้ดเลย:

```
// VULNERABLE: key is in the source, will be committed to git
const stripe = new Stripe("sk_live_51H8xQ2eZvK...");
```

```
// SAFE: key is read from the environment, never written in code
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
```

ความลับมีบ้านที่ปลอดภัยเพียงที่เดียวเท่านั้น — ฝั่งเซิร์ฟเวอร์ ที่อ่านมาจาก environment เบริวเซอร์คือพื้นที่สาธารณะ อะไรก็ตามที่ถูกส่งไปที่นั่นสามารถถูกอ่านได้โดยใครก็ตามที่เปิด dev tools



เพิ่ม .env (และ .env.local ฯลฯ) เข้าไปใน .gitignore ของคุณ ก่อน ที่จะเขียนความลับสักตัวเดียว ถ้าคีย์ตัวใด ตกไปอยู่ในโค้ดหรือประวัติของคุณ ให้ถือว่ามันถูกเผาแล้ว: หมุนมันใหม่ (rotate — สร้างตัวใหม่และเพิกถอนตัวเก่า) เพราะแค่ลบบรรทัดนั้นไม่พอ เนื่องจากคีย์เก่ายังคงใช้งานได้และยังอยู่ข้างนอกนั้น

Authentication vs Authorization: เอนด์พอยต์ที่ใครก็เรียกได้

สองคำนี้ฟังดูคล้ายกัน และความแตกต่างระหว่างมันคือจุดที่การเจาะระบบจริง ๆ มักเกิดขึ้น

- Authentication คือ คุณเป็นใคร? — การล็อกอิน การพิสูจน์ตัวตน
- Authorization คือ คุณได้รับอนุญาตให้ทำอะไรได้บ้าง? — ว่าผู้ใช้ที่ล็อกอินอยู่ คนนี้ สามารถทำ การกระทำนี้ กับ ข้อมูลนี้ ได้หรือไม่

AI ทำ authentication ได้ดีพอสมควร เพราะโลบราจัดการส่วนใหญ่ให้อยู่แล้ว Authorization ต่างหากที่มันมักล้นหลามเป็นประจำ เพราะ authorization ขึ้นอยู่กับกฎเฉพาะของแอปคุณ ซึ่ง AI ไม่รู้จัก นี่คือหายนะในตำราเรียน:

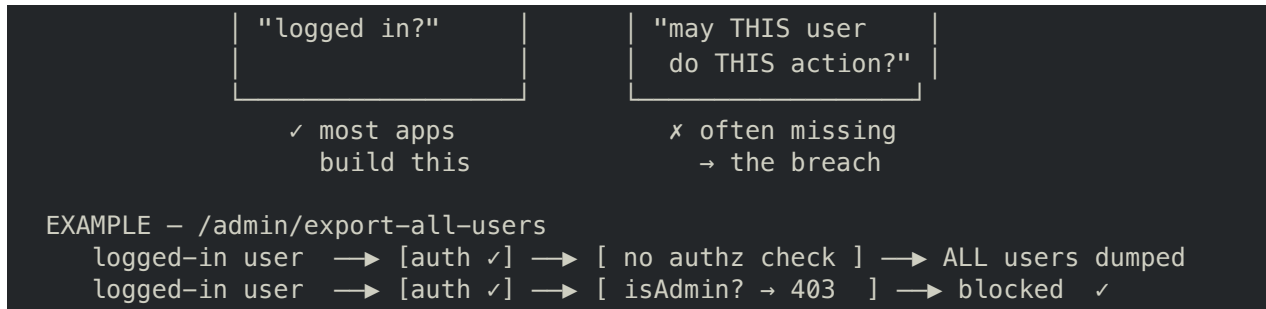
```
// VULNERABLE: checks you're logged in, but not WHO you are
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  res.json(db.getAllUsers()); // any logged-in user can hit this
});
```

เอนด์พอยต์นั้น “ได้รับการป้องกัน” แล้ว — คุณต้องล็อกอินก่อน แต่ผู้ใช้ที่ล็อกอินอยู่ คนไหนก็ได้ รวมถึงคนที่เพิ่งสมัคร ไปเมื่อสามสัปดาห์ก่อน สามารถเรียกมันและดาวน์โหลดข้อมูลของผู้ใช้ทุกคนได้ มี authentication แต่ไม่มี authorization วิธีแก้คือตรวจสอบสิทธิ์บนการกระทำนั้นโดยตรง:

```
// SAFE: confirms this user is actually an admin
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  if (!req.user.isAdmin) return res.status(403).send("Forbidden");
  res.json(db.getAllUsers());
});
```

ลองนึกภาพประตูดงสองบานที่คำขอต้องผ่านให้ได้ Authentication ถามว่า “คุณล็อกอินอยู่ไหม?”; authorization ถามว่า “คุณคนนี้ได้รออนุญาตให้ทำสิ่งนี้ไหม?” การเจาะระบบเกิดขึ้นเมื่อแอปสร้างประตูแรกแต่ลืมสร้างประตูที่สอง:





กับดักเดียวกันปรากฏในรูปแบบย่อ ๆ ได้ทุกที่ เช่น เอนด์พอยต์ที่คืนค่าออเดอร์หมายเลข #1234 โดยไม่ตรวจสอบว่าออเดอร์นั้นเป็นของผู้ใช้ที่ร้องขอหรือไม่ ใครก็สามารถเปลี่ยนหมายเลขใน URL แล้วอ่านออเดอร์ของคนอื่นได้ กฎที่ไม่หรูหราแต่เด็ดขาดคือ: ตรวจสอบสิทธิ์บนทุกเอนด์พอยต์ที่แตะข้อมูล และอย่าไว้ใจ ID ที่มาจากไคลเอนต์ อย่าคิดว่า URL ที่ซ่อนอยู่จะปลอดภัยเพราะมันถูกซ่อนไว้ — “ไม่มีใครรู้ว่าสิ่งนี้มีอยู่” ไม่ใช่มาตรการรักษาความปลอดภัย

การจัดการไฟล์และการอัปโหลด

การให้ผู้ใช้อัปโหลดไฟล์คือทุ่นระเบิดที่เจียบสงบ และร่างแรกของ AI แทบไม่เคยปิดช่องโหว่เหล่านี้เลย:

- ตรวจสอบชนิดและขนาดของไฟล์ ถ้าไม่มีการจำกัด ใครสักคนอาจอัปโหลดไฟล์ขนาด 10 GB หรือไฟล์ที่รันได้ (executable) ตรวจสอบเนื้อหาจริง ไม่ใช่แค่นามสกุลไฟล์ ซึ่งใครก็เปลี่ยนชื่อได้
- อย่าไว้ใจชื่อไฟล์ ชื่อไฟล์อย่าง ../../etc/passwd สามารถทำให้ตัวจัดการที่ไม่รัดกุมเขียนไฟล์ออกนอกโฟลเดอร์ที่ตั้งใจไว้ (เรียกว่า path traversal) ให้สร้างชื่อที่ปลอดภัยของคุณเองแทนที่จะใช้ชื่อที่ผู้ใช้ให้มา
- อย่าให้บริการไฟล์ที่อัปโหลดจากต้นทาง (origin) เดียวกับแอปของคุณ และอย่าปล่อยให้ไฟล์ที่ผู้ใช้อัปโหลดถูกรันเป็นโค้ด ไฟล์ที่ผู้ใช้ส่งเข้ามาแล้วถูกวางไว้ในโฟลเดอร์ที่เซิร์ฟเวอร์ของคุณจะ รัน คือเส้นทางตรงสู่การถูกยึดครอง

คุณไม่จำเป็นต้องสร้างสิ่งเหล่านี้ทั้งหมดเอง — บริการจัดเก็บข้อมูลจะจัดการส่วนใหญ่ให้ — แต่คุณต้อง ขอ เพราะ AI จะไม่เสนอมันให้เองโดยสมัครใจ “เพิ่มฟีเจอร์อัปโหลด” จะได้เพียงเส้นทางที่ราบรื่นที่สุด “เพิ่มฟีเจอร์อัปโหลดที่ตรวจสอบชนิดและขนาดไฟล์ ปฏิเสธทุกอย่างยกเว้นรูปภาพ และเก็บไฟล์ภายใต้ชื่อที่สร้างขึ้น” จะได้บางอย่างที่ใกล้เคียงความปลอดภัยมากกว่า

ความเสี่ยงจากดีเพนเดนซี: แพ็กเกจที่ AI คิดขึ้นมาเอง

แอปสมัยใหม่พึ่งพาแพ็กเกจของบุคคลที่สามหลายสิบตัว และ AI ก็แนะนำแพ็กเกจเหล่านี้อย่างรวดเร็ว ความเสี่ยงสองอย่างมาของมัน

อย่างแรกคือ typosquatting และแพ็กเกจที่ถูกคิดขึ้นเอง (hallucinated) ผู้โจมตีเผยแพร่แพ็กเกจอันตรายด้วยชื่อที่ต่างจากของจริงนิดหน่อย (เช่น regeusts แทนที่จะเป็น requests) โดยพนันว่าจะมีคนพิมพ์ผิด บางครั้ง AI ก็นำเข้าแพ็กเกจที่ไม่มีอยู่จริงด้วยความมั่นใจ — และผู้โจมตีที่สังเกตเห็นพฤติกรรมนี้สามารถจดทะเบียนชื่อนั้นพร้อมมัลแวร์ข้างในได้ ก่อนที่จะติดตั้งอะไรก็ตามที่ AI แนะนำ ให้ลองดูก่อนว่า มันมีอยู่จริงไหม มียอดดาวน์โหลดและรีโพลจริงไหม ชื่อสะกดตามที่คุณคาดหวังไหม

อย่างที่สองคือ ดีเพนเดนซีที่ไม่ได้รับการตรวจสอบโดยทั่วไป แพ็กเกจทุกตัวที่คุณเพิ่มเข้ามาคือโค้ดที่ทำงานด้วยสิทธิ์การเข้าถึงเต็มรูปแบบของแอปคุณ ยิ่งดีเพนเดนซีเยอะ พื้นที่เสี่ยงต่อทั้งบั๊กและการโจมตีห่วงโซ่อุปทาน (supply-chain attack) ก็ยิ่งมากขึ้น เลือกลอบรรีที่มีชื่อเสียงและมีน้อยกว่า แทนที่จะใช้แพ็กเกจแปลกหน้าจำนวนมาก และถามตัวเองว่า “เราต้องการแพ็กเกจสำหรับสิ่งนี้จริงหรือ หรือแค่สับสนบรรทัดก็พอ?” ก่อนที่จะเพิ่มมันเข้ามา

รันเครื่องมือตรวจสอบ (audit tool) ของระบบนิเวศของคุณเป็นระยะ ๆ (npm audit, pip-audit และเครื่องมือแบบเดียวกัน) — มันจะแจ้งช่องโหว่ที่รู้จักแล้วในสิ่งที่คุณติดตั้งไปแล้ว และ AI ก็เก่งในการแก้ไขสิ่งที่มันรายงาน

ประตูละเอียดตรวจสอบความปลอดภัย

นี่คือนิสัยเดียวที่จะเปลี่ยนทุกอย่างข้างต้นจากความกังวลให้กลายเป็นกระบวนการ: ก่อนที่คุณจะปล่อยงาน ให้ AI โจมตีโค้ดของมันเอง โมเดลที่เขียนพีเจอรันนั้นมักจะหาช่องโหว่ในมันได้ — เพียงแต่มันจะไม่ทำ เว้นแต่คุณจะขอ พลิกบทบาทมันจากผู้สร้างเป็นศัตรู:

You wrote this endpoint. Now act as an attacker trying to break it.

List every way a malicious user could:

- read or modify data they shouldn't (authorization holes)
- inject code via input (SQL injection, XSS, command injection)
- abuse missing validation or rate limits

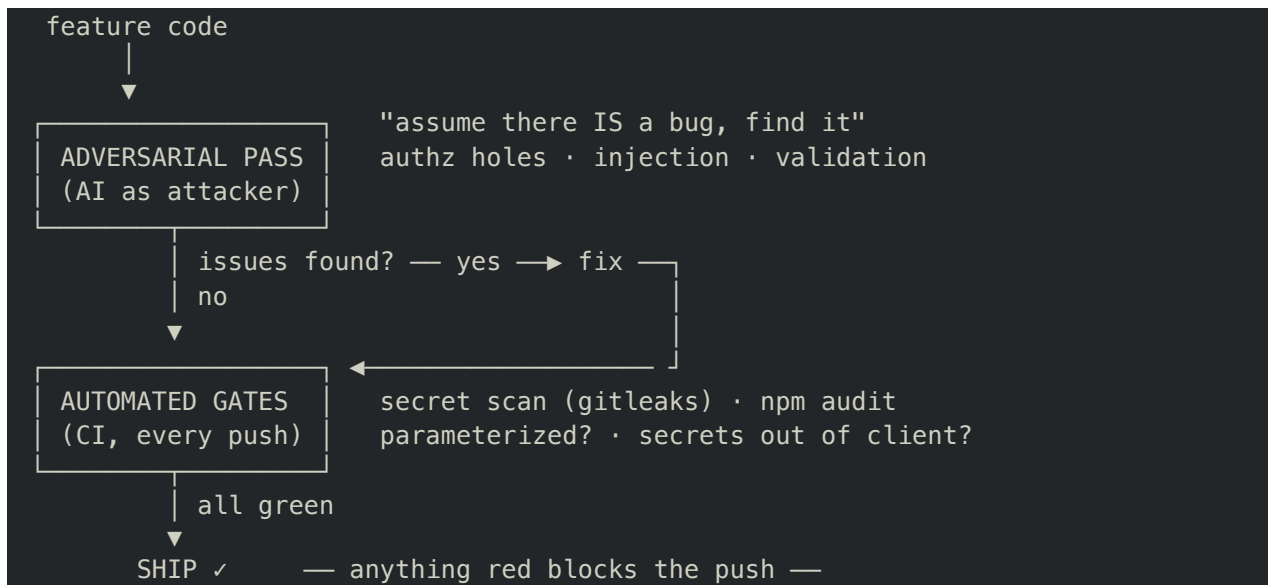
For each, show the exact request that exploits it, then the fix.

Don't reassure me – assume there IS a vulnerability and find it.

บรรทัดสุดท้ายนั้นสำคัญมาก: ถ้าปล่อยเป็นกลาง AI มักจะบอกว่า “ดูปลอดภัยดี!” แต่เมื่อสั่งให้สมมติว่ามีข้อบกพร่องอยู่ มันก็จะไปค้นหาจริง ๆ จับคู่การตรวจสอบเชิงปฏิบัติกับเช็กลิสต์สั้น ๆ ก่อนปล่อยงานที่คุณรันกับทุกอย่างที่ผู้ใช้ต้องเจอ:

- ☐ ทุกเอนด์พอยต์ตรวจสอบสิทธิ์ ไม่ใช่แค่ว่าผู้ใช้ล็อกอินอยู่
- ☐ ฐานข้อมูลทั้งหมดเป็นแบบ parameterized — ไม่มี SQL ที่สร้างด้วยการต่อสตริง
- ☐ ข้อมูลนำเข้าจากผู้ใช้ที่แสดงผลบนหน้าเว็บถูก escape แล้ว (ไม่มีการแทรก HTML ดิบ)
- ☐ ไม่มีความลับในโค้ดฝั่งไคลเอนต์ และไม่มีอันไหนถูก commit เข้ารีโพอ
- ☐ .env อยู่ใน gitignore แล้ว และคีย์ที่เคยรั่วไหลได้ถูกหมุนใหม่แล้ว
- ☐ การอัปโหลดไฟล์ตรวจสอบชนิดและขนาด และใช้ชื่อที่สร้างขึ้น
- ☐ ดีเพนเดนซีใหม่ ๆ ถูกตรวจสอบด้วยตัวว่ามีอยู่จริงและมีชื่อเสียงดี

ลองนึกภาพประตุนี้เป็นลำดับขั้นที่ไม่มีอะไรจะถูกปล่อยออกไปได้โดยไม่ผ่าน AI สลับหมวกจากผู้สร้างมาเป็นผู้โจมตี จากนั้นการสแกนอัตโนมัติจะคอยรองรับสิ่งที่การตรวจสอบด้วยมนุษย์อาจพลาดไป:



และรัน secret scanner ก่อนที่คุณจะ push — เครื่องมืออย่าง gitleaks (หรือการสแกนในตัวของคุณ) จะ grep โค้ดและประวัติของคุณเพื่อหาสิ่งที่มีรูปร่างเหมือนคีย์ มันคือเซฟตี้เน็ตแบบคำสั่งเดียวสำหรับความผิดพลาดที่แพงที่สุดในรายการนี้ และคุณสามารถให้ AI เชื่อมมันเข้ากับ CI (Continuous Integration — หน้าที่รันการตรวจสอบของคุณโดยอัตโนมัติทุกครั้งที่คุณ push โค้ด) เพื่อให้มันรันทุกครั้งที่คุณ push

ความประมาทจากระบบอัตโนมัติ ในเวอร์ชันความปลอดภัย

ช่วงเวลาที่ยันตรายที่สุดไม่ใช่วันแรก — แต่เป็นวันที่แก้ไข ในวันแรกคุณระมัดระวัง คุณอ่านทุกดีพฟ์ คุณรันพรอมปีแบบผู้โจมตี คุณตรวจสอบตัวจัดการการอัปเดต แล้วเมื่อการเปลี่ยนแปลงร้อยครั้งถูกปล่อยออกไปโดยไม่มีอะไรเลวร้ายเกิดขึ้น การตรวจสอบก็เริ่มรู้สึกเหมือนพิธีกรรม คุณข้ามประตูตรวจสอบไป “แค่ครั้งนี้ครั้งเดียว” สำหรับการเปลี่ยนแปลงเล็ก ๆ การเปลี่ยนแปลงเล็กนั้นเพิ่มเอนด์พอยต์ที่ไม่มีการตรวจสอบสิทธิ์เข้าไป ไม่มีอะไรเกิดขึ้นสักพัก เพราะการไม่มีอะไรเกิดขึ้นนั่นแหละคือลักษณะของช่องโหว่ด้านความปลอดภัยเมื่อมองจากภายใน

นี่คือความประมาทจากระบบอัตโนมัติที่เข้าไปที่ความปลอดภัย และมันแย่กว่าที่ไหน ๆ เพราะความล้มเหลวด้านความปลอดภัยนั้นเงียบและมาช้า พีเจอร์ทที่ฟังจะลงโทษคุณภายในไม่กี่นาที แต่การตรวจสอบสิทธิ์ที่ฟังจะให้รางวัลคุณด้วยความสำเร็จที่ดูเหมือนจริงไปเรื่อย ๆ จนกว่าจะถึงจุดที่ถูกเจาะระบบ การไม่มีหายนะไม่ใช่หลักฐานของความปลอดภัย — มันคือสภาวะปกติของแอปที่มีช่องโหว่ที่ยังไม่ถูกพบเท่านั้นเอง

การป้องกันคือทำให้ประตูตรวจสอบนั้นถูกจนคุณไม่ข้ามมันไป และเป็นอัตโนมัติมากพอจนคุณไม่สามารถข้ามมันได้ ใส่การสแกนความลับและ audit ไว้ใน CI เพื่อให้มันรันโดยไม่ต้องอาศัยความจำของคุณ และเตรียมพรอมปีแบบผู้โจมตีไว้ให้พร้อมสำหรับสิ่งใดก็ตามที่แตะ auth ข้อมูล หรือการอัปเดต ยึดเส้นเดียวไว้ให้มัน: การเปลี่ยนแปลงที่จัดการข้อมูลผู้ใช้หรือสิทธิ์จะไม่ถูกปล่อยออกไป จนกว่าจะมีใครสักคน — คุณเอง พร้อมความช่วยเหลืออย่างซื่อสัตย์จาก AI — ได้ลองโจมตีมันอย่างจริงจัง ความเร็วใน vibe coding มาจากประตูตรวจสอบที่ดี ไม่ใช่จากการเชื่อว่าความเงียบหมายถึงทุกอย่างโอเค

สรุปและฝึกฝน

ประเด็นสำคัญ

- โค้ดที่ AI สร้างไม่ปลอดภัยโดยค่าเริ่มต้น ให้สมมติว่ามีช่องโหว่อยู่และไปค้นหามัน แทนที่จะเชื่อความประทับใจแรกที่คุณสะดุด
- หมวดหมู่ใหญ่ ๆ ซ้ำเดิมเสมอ: injection, ความลับที่เปิดเผย, การตรวจสอบสิทธิ์ที่ขาดหาย, การอัปเดตไฟล์ที่ไม่ปลอดภัย และดีเพนเดนซีที่ถูกคิดขึ้นเอง
- Authorization เป็นเรื่องต่อคำขอ ไม่ใช่ต่อการล็อกอิน — ทุกเอนด์พอยต์ต้องตรวจสอบว่าผู้ใช้ คนนี้ สามารถทำการกระทำนี้ ได้หรือไม่ ไม่ใช่แค่ว่าพวกเขาล็อกอินอยู่
- รัน secret scanner อย่าง git leaks ก่อนทุกครั้งที่จะ push และหมุนเคียวใดก็ตามที่เคยรั่วไหล เชื่อมการสแกนเข้ากับ CI เพื่อที่คุณจะข้ามมันไม่ได้
- วันที่แก้ไขอันตรายกว่าวันแรก — ทำให้ประตูตรวจสอบความปลอดภัยถูกจนไม่ข้าม และเป็นอัตโนมัติมากพอจนข้ามไม่ได้

ลองทำดู

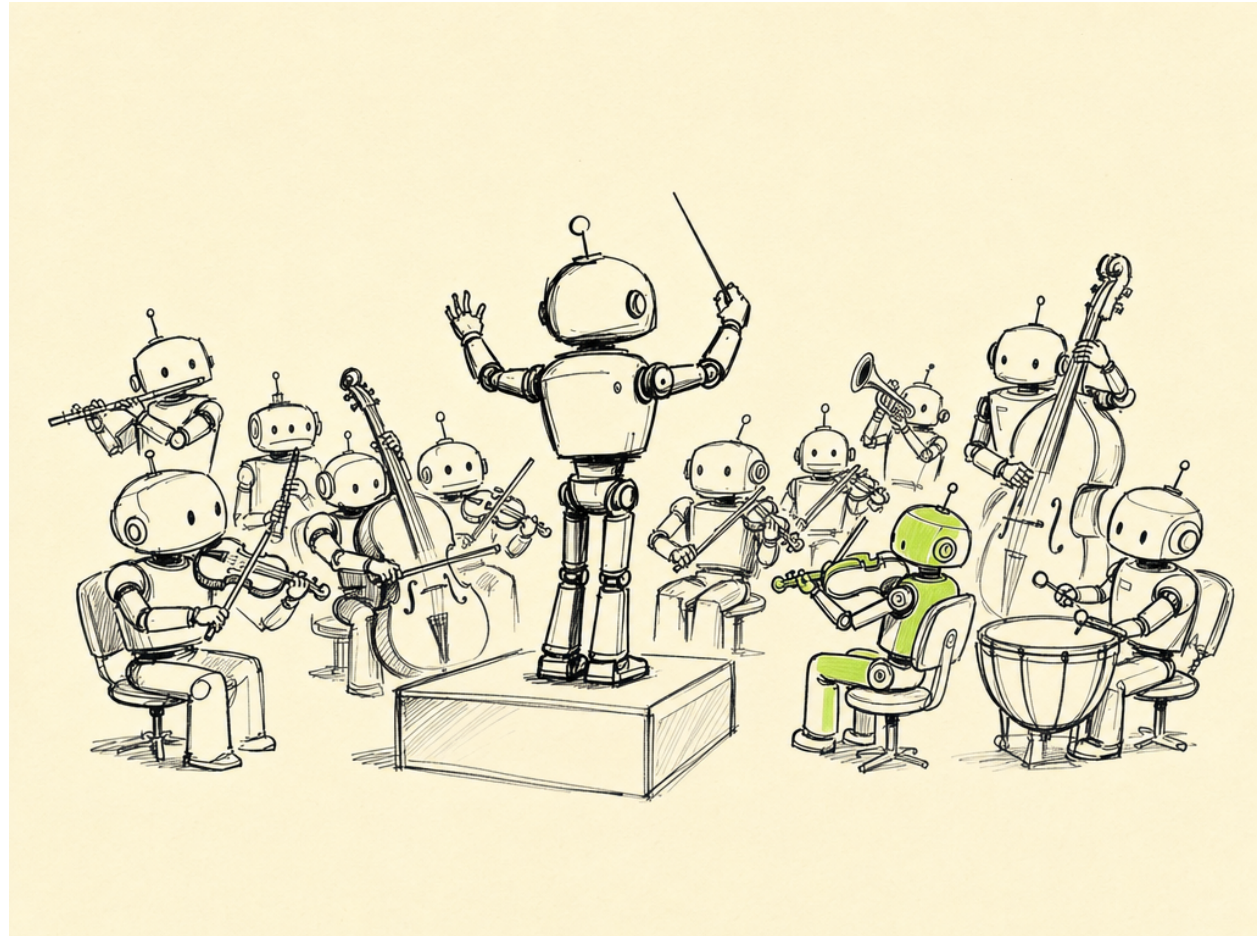
หยิบพีเจอร์ทใดก็ได้ของคุณที่แตะ auth ข้อมูลผู้ใช้ หรือการอัปเดต แล้วรันพรอมปีตรวจสอบเชิงปฏิบัติด้านล่างนี้กับมัน สำหรับแต่ละช่องโหว่ที่ AI รายงาน ให้ขอคำขอที่ใช้โจมตีมันจริง ๆ และวิธีแก้ที่น้อยที่สุด จากนั้นนำวิธีแก้ไปใช้และรันพรอมปีอีกครั้ง ปิดท้ายด้วยการรัน git leaks (หรือการสแกนความลับของแพลตฟอร์มคุณ) กับรีโพและประวัติของมัน

พรอมปีประจำบท

```
Here is the code for a feature that handles [auth / user data / uploads]:  
[paste the code].  
Act as an attacker reviewing this for security holes. Specifically check:  
- broken authorization (can a user act on another user's data?)  
- injection via input (SQL injection, XSS, command injection)  
- exposed secrets or keys in code or responses  
- unsafe file handling (type, size, path, generated names)
```

```
- missing validation or rate limits
For EACH issue, show the exact request that exploits it, then the fix.
Do NOT reassure me - assume there IS a vulnerability and find it.
```

การจัดการเอเจนต์ (Agent Orchestration)



ตลอดทั้งเล่มนี้ AI ทำหน้าที่เหมือนเครื่องมือที่คุณถืออยู่ในมือ: คุณถาม มันตอบ คุณตรวจสอบผลลัพธ์ นั้นยังคงเป็นภาพจำลองที่ถูกต้องในเวลาส่วนใหญ่ แต่เครื่องมือเหล่านี้ได้พัฒนาขึ้นมาอีกโหมดหนึ่ง — โหมดที่คุณส่งมอบ เป้าหมายแทนที่จะเป็น ขั้นตอน และ AI จะทำงานเป็นเวลาหลายนาที (บางครั้งนานกว่านั้น) ก่อนที่มันจะกลับมาหาคุณ

บทนี้พูดถึงการเปลี่ยนแปลงนั้น และการไม่หลุดจากความเข้าใจเมื่อมันเกิดขึ้น สิ่งที่น่าสนใจคือการตีความคำว่า “เอเจนต์” (agent) ว่า “ตอนนี้ฉันหยุดใส่ใจได้แล้ว” นั่นคือการตีความที่ทำให้คนฟังพินาศ เอเจนต์คือแรงยกสำหรับงานที่คุณเข้าใจอยู่แล้ว — ไม่ใช่ใบอนุญาตให้ส่งมอบโค้ดที่คุณเลิกดูมันแล้ว

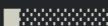
สามระดับ

การมองความช่วยเหลือของ AI เป็นสามระดับจะช่วยให้ แต่ละระดับมอบเชื่อทำให้คุณมากขึ้นเรื่อยๆ

- การเติมโค้ดในตัวแก้ไข (in-editor completions) AI จะแนะนำบรรทัดหรือบล็อกถัดไปขณะที่คุณพิมพ์ คุณควบคุมได้เต็มที่ คุณยอมรับหรือปฏิเสธข้อเสนอแนะแต่ละอันเอง หน่วยของงานคือไม่ก็บรรทัด และโดยปกติคุณจะตรวจสอบทุกบรรทัดเพราะมันปรากฏอยู่ที่เคอร์เซอร์ของคุณโดยตรง

- ผู้ช่วยระดับทั้งโปรเจกต์ (repo-level assistants) คุณคุยกับผู้ช่วยที่มองเห็นทั้งโปรเจกต์ของคุณ แก้ไขหลายไฟล์พร้อมกัน รันคำสั่ง และอ่านผลลัพธ์ นี่คืองานส่วนใหญ่ที่เกิดขึ้นในหนังสือเล่มนี้ หน่วยของงานคือหนึ่งขั้นตอนหรือฟีเจอร์เล็กๆ และคุณตรวจสอบ diff (ความแตกต่างของโค้ดที่เปลี่ยนไป) หลังจากแต่ละรอบ
- เอเจนต์อัตโนมัติ (autonomous agents) คุณให้เป้าหมาย — “ทำให้ชุดทดสอบผ่านทั้งหมด” “ย้ายโมดูลนี้ไปใช้ API (Application Programming Interface — ชุดคำสั่งที่ซอฟต์แวร์ชั้นหนึ่งเปิดให้ซอฟต์แวร์อื่นเรียกใช้) ใหม่” — แล้วเอเจนต์จะวางแผน แก้ไข รัน อ่านข้อผิดพลาด และลองใหม่ซ้ำแล้วซ้ำเล่าหลายขั้นตอน คุณตรวจสอบผลลัพธ์ ไม่ใช่การกระทำแต่ละอย่าง หน่วยของงานคืองานทั้งชิ้น

แต่ละระดับมอบเชื่อให้คุณมากขึ้น: หน่วยของงานที่ใหญ่ขึ้น การบังคับทิศทางที่ต่อเนื่องน้อยลง และจุดที่คุณตรวจสอบที่ช้าลง:

TIER	UNIT OF WORK	YOU REVIEW	CONTROL
completions ▼	a few lines	each suggestion	 tight
repo assistant ▼	a step/feature	the diff/turn	
autonomous agent	a whole task	the RESULT	 loose

more rope ▼ — steer continuously → set loose, check outcome

การกระโดดที่สำคัญคือขั้นสุดท้าย ในสองระดับแรก คุณบังคับทิศทางอย่างต่อเนื่อง แต่กับเอเจนต์อัตโนมัติ คุณปล่อยมันไปแล้วตรวจสอบผลลัพธ์ นั่นเป็นทักษะที่แตกต่างออกไป และเป็นเนื้อหาของบทนี้ที่เหลือ

เมื่อไรควรปล่อยให้มันรัน vs. บังคับทิศทางด้วยตัวเอง

การตัดสินใจว่าจะปล่อยให้เอเจนต์รันโดยไม่มีคนดูแลนั้นขึ้นอยู่กับคำถามสามข้อ — เช็กลิสต์ที่คุณต้องทำ ก่อน ที่จะเดินจากไป

- รัศมีความเสียหาย (blast radius) ถ้าสิ่งนี้ผิดพลาด จะเสียหายมากแค่ไหน? การจัดรูปแบบโค้ดทั้งโปรเจกต์มีรัศมีความเสียหายเล็ก การย้ายฐานข้อมูล (database migration) หรือการเปลี่ยนตรรกะการยืนยันตัวตน (auth logic) มีรัศมีใหญ่ ยิ่งรัศมีใหญ่เท่าไร ยิ่งต้องบังคับทิศทางด้วยมือมากเท่านั้น
- ความสามารถในการย้อนกลับ (reversibility) คุณสามารถยกเลิกมันได้ในราคาถูกหรือไม่? งานบน git branch ใหม่ที่มี working tree สะอาดสามารถย้อนกลับได้ — git restore . แล้วมันก็หายไป งานที่ลบข้อมูล ส่งอีเมลเรียกเก็บเงินบัตรเครดิต หรือ push ไปยังระบบจริง (production) ไม่สามารถย้อนกลับได้ อย่าปล่อยให้เอเจนต์รันโดยไม่มีคนดูแลกับสิ่งใดที่คุณเอากลับคืนไม่ได้
- ความสามารถในการตรวจสอบ (verifiability) คุณบอกได้ไหมว่าผลลัพธ์ถูกต้อง? ถ้า “เสร็จแล้ว” หมายถึงชุดทดสอบสีเขียวหรือหน้าเว็บที่ใช้งานได้จริงต่อหน้าคุณ คุณตรวจสอบได้ในไม่กี่วินาที และเอเจนต์ก็เหมาะสมมาก แต่ถ้า “เสร็จแล้ว” ขึ้นอยู่กับการตัดสินใจที่ละเอียดอ่อน — การคำนวณเงิน กฎความปลอดภัย เงื่อนไขการแข่งขันของกระบวนการ (race conditions) — คุณจะตรวจสอบด้วยการเช็คเร็วๆ ไม่ได้ คุณต้องอยู่ใกล้ๆ

วิธีง่าย ๆ ในการรวมทั้งสามข้อเข้าด้วยกัน: ปล่อยให้มันรันเมื่องานย้อนกลับได้และผลลัพธ์ตรวจสอบได้ในราคาถูก บังคับทิศทางด้วยมือเมื่อไม่ใช่ นำกฎจากบทเรื่องการสร้างฟีเจอร์มาใช้ต่อ — ยิ่งรัศมีความเสียหายใหญ่ ขั้นตอนยิ่งต้องเล็ก — แล้วเพิ่มกฎคู่กันของมัน: ยิ่งรัศมีความเสียหายใหญ่ ยิ่งต้องผูกสายจูงให้แน่นขึ้น

กำหนดขอบเขตงานให้เดินหลงไม่ได้

เอเจนต์อัตโนมัติล้มเหลวในแบบเฉพาะตัว: มันเดินหลงทาง คุณขอสิ่งหนึ่ง มันกลับ “ช่วยเหลือ” ด้วยการปรับโครงสร้าง (refactor) อีกสามอย่าง แต่ไฟล์ที่คุณไม่เคยพูดถึง แล้วกลับมาพร้อม diff ที่คุณอ่านไม่ออก การแก้ปัญหาไม่ใช่เอเจนต์ที่ฉลาดกว่า — แต่คือบริฟ (brief) ที่รัดกุมกว่า

งานอัตโนมัติที่มีสี่ส่วน: เป้าหมาย (goal) ที่ชัดเจน ข้อจำกัด (constraints) ที่ระบุไว้ชัด เกณฑ์ความสำเร็จ (done-criterion) ที่เป็นรูปธรรม และ รั้วกันความปลอดภัย (guardrails) ที่เอเจนต์รันตรวจสอบเองได้ รั้วกันความปลอดภัยเป็นส่วนสำคัญที่สุด เพราะมันคือรั้วที่เอเจนต์ปีนข้ามไม่ได้โดยที่คุณไม่รู้ตัว

Goal: Make the failing tests in tests/checkout/ pass.

Constraints:

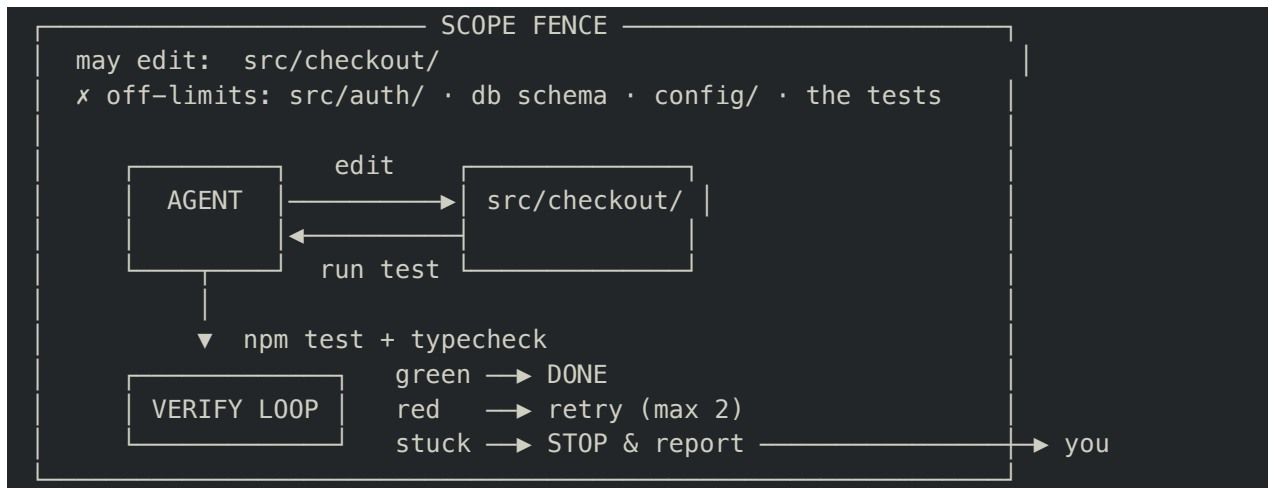
- Only edit files under src/checkout/. Do not touch src/auth/, the database schema, or anything in config/.
- Do not add new dependencies.
- Do not change the tests themselves – the tests define correct behavior.

Done when:

- `npm test tests/checkout/` is fully green.
- `npm run typecheck` passes.
- `git diff --stat` shows changes only under src/checkout/.

If you get stuck after two real attempts, stop and report what you tried and what's still failing. Don't paper over a failure to make the suite go green.

บริพนี้ใช้ได้ผลเพราะมันซิงเอเจนต์ไว้ในกล่อง: ข้อจำกัดกักกำแพงว่าจะอะไรที่มันแตะได้ เกณฑ์ความสำเร็จคือคำสั่งที่มันรันได้จริง และวงจรการตรวจสอบ (verify loop) ทำให้มันซื่อสัตย์:



มีสามสิ่งที่ทำให้บริพนี้ยากที่จะเดินหลงออกไป ข้อจำกัดวาดกำแพงล้อมรอบไฟล์ที่เอเจนต์แตะได้ เกณฑ์ความสำเร็จคือคำสั่งที่ใครก็รันได้ ไม่ใช่ความรู้สึก และย่อหน้าสุดท้ายให้สิทธิ์มันที่จะ หยุดและถาม แทนที่จะดันต่อไป — ซึ่งป้องกันหายนะแบบสโลว์โมชันที่เอเจนต์ยัง “แก้ไข” สิ่งต่างๆ ต่อไปเรื่อยๆ นานยี่สิบนาทีแล้วทั้งต้นไม้ (tree) ไว้ในสภาพแย่กว่าตอนที่มันพบ

ชุดทดสอบคือรั้วกันตัวจริง เอเจนต์ที่รันชุดทดสอบของตัวเองได้จะมีวงจรป้อนกลับ (feedback loop) ที่แน่นและซื่อสัตย์ และส่วนใหญ่จะอยู่ในนั้น ส่วนตัวที่ไม่มีวิธีตรวจสอบงานตัวเองจะส่งมอบสิ่งที่พังให้คุณอย่างมั่นใจ ถ้าคุณจะจำนัยเดียวจากบทนี้: มอบวิธีให้เอเจนต์ตรวจสอบตัวเองได้ และทำให้การผ่านมันเป็นนิยามของ “เสร็จแล้ว”

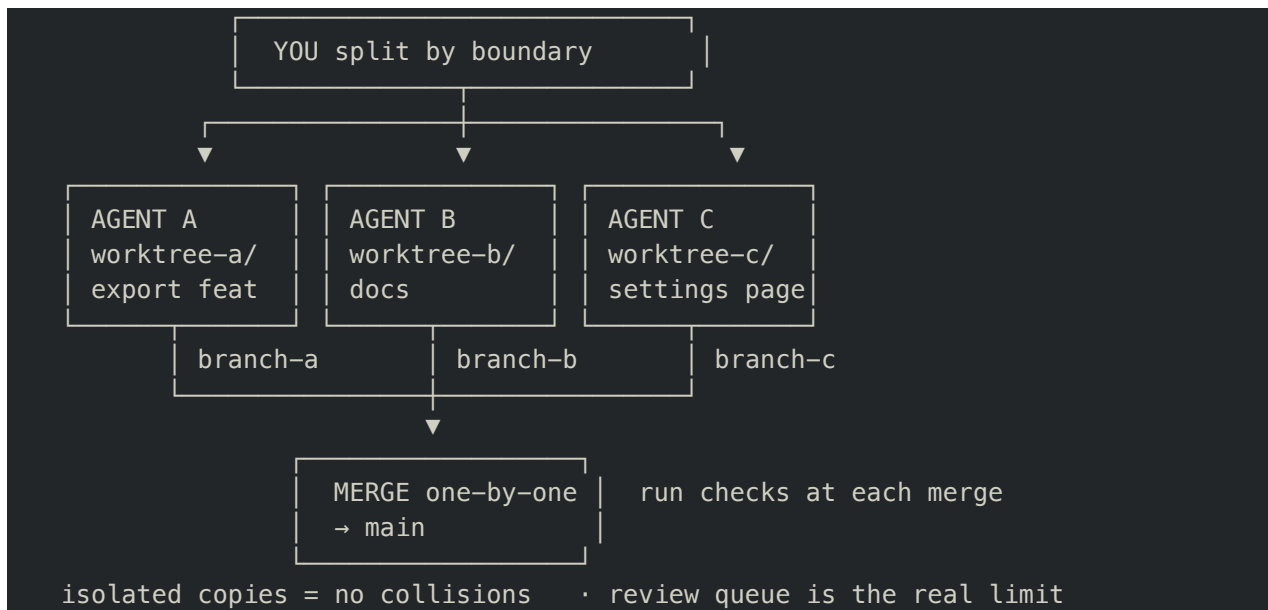
เอเจนต์แบบพื้นหลังและแบบขนาน

เมื่อคุณเริ่มไว้ใจเอเจนต์กับงานที่กำหนดขอบเขตแล้ว ขั้นตอนถัดไปก็ชัดเจน: รันหลายตัวพร้อมกัน เครื่องมือสมัยใหม่ให้คุณเริ่มเอเจนต์แบบพื้นหลัง (background agents) — แต่ละตัวอยู่บน branch หรือ worktree ของตัวเอง (สำหรับการทำงานแยกต่างหากของโปรเจกต์ เพื่อให้การแก้ไขของเอเจนต์ตัวหนึ่งไม่ชนกับอีกตัว) — ทำให้งานที่กำหนดขอบเขตไว้สามสิ่งงานคืบหน้าไปพร้อมกันในขณะที่คุณทำอย่างอื่น

นี่คือการเร่งความเร็วที่แท้จริง แต่ใช้ได้เฉพาะกับงานที่เป็นอิสระ จากกันเท่านั้น สิ่งที่คุณแลกมาคือการประสานงาน (coordination) เอเจนต์สองตัวที่แก้ไขไฟล์เดียวกันจะสร้าง diff ที่ขัดแย้งกัน และการรวมกันด้วยมืออาจมีต้นทุนสูงกว่าที่คุณประหยัดได้

- แบ่งตามขอบเขต ไม่ใช่ตามจำนวนบรรทัด มอบชิ้นงานให้เอเจนต์แต่ละตัวโดยไม่ทับซ้อนกับตัวอื่น — โมดูลแยก ไฟล์แยก “เอเจนต์ A ทำฟีเจอร์ส่งออกข้อมูล เอเจนต์ B เขียนเอกสาร” เป็นวิธีที่สะอาด “ทั้งสองเอเจนต์แก้ไข app.js” คือปัญหาการรวม (merge) ที่รอเกิดขึ้น
- แยกพื้นที่ทำงาน รันเอเจนต์แต่ละตัวบน git branch หรือ worktree ของตัวเอง เพื่อไม่ให้เหยียบการเปลี่ยนแปลงที่ยังไม่ได้ commit ของกันและกัน รวมกลับทีละตัว โดยรันการตรวจสอบของคุณในแต่ละครั้งที่รวม
- รักษาจำนวนให้ซื้อสัตย์ คุณยังต้องตรวจสอบผลลัพธ์ทุกอย่าง เอเจนต์สี่ตัวที่กำลังรันคือ diff สี่ชุดที่ต้องอ่านและชุดทดสอบสี่ชุดที่ต้องเชื่อถือ เกินกว่าจำนวนหยิบมือหนึ่ง วิศวกรรมตรวจสอบ — ไม่ใช่เอเจนต์ — จะกลายเป็นคอขวดของคุณ

เอเจนต์แต่ละตัวทำงานใน worktree ที่แยกจากกันของตัวเอง เพื่อให้การแก้ไขของพวกเขาไม่มีวันชนกัน คุณรวมมันกลับทีละตัว โดยรันการตรวจสอบที่แต่ละด้าน:



ข้อผิดพลาดคือการปฏิบัติต่อการทำงานแบบขนานเหมือนว่ามันฟรี ทั้งที่ไม่ใช่ — คุณแค่ย้ายงานจาก การเขียน ไปเป็นการตรวจสอบและรวมเข้าด้วยกัน บ่อยครั้งมันเป็นการแลกเปลี่ยนที่ดีเยี่ยม แต่เฉพาะเมื่อคุณตรวจสอบจริงๆ เท่านั้น

หลายเอเจนต์: กระจายงาน, ฝ่ายตรวจสอบปฏิปักษ์, การสังเคราะห์

มี “เอเจนต์หลายตัว” อีกแบบหนึ่งที่ทำงาน ภายใน งานเดียวกัน แทนที่จะทำงานคนละงาน แทนที่จะให้เอเจนต์ตัวเดียวทำทุกอย่าง คุณแบ่งบทบาทออกจากกัน:

- กระจายงานเพื่อความครอบคลุม (fan-out for breadth) เอเจนต์หลายตัวจัดการปัญหาเดียวกันจากมุมมองที่ต่างกัน — หรือสำรวจส่วนต่างๆ ของโค้ดเบสขนาดใหญ่ — แล้วรายงานกลับ มีประโยชน์เมื่อคุณต้องการความครอบคลุม: “หาทุกที่ที่เราเรียกใช้ฟังก์ชันที่เลิกใช้แล้วนี้” ขนานได้ดี
- ฝ่ายตรวจสอบปฏิปักษ์ (adversarial reviewer) เอเจนต์ตัวหนึ่งเขียน ส่วนอีกตัวมีหน้าที่เดียวคือ โจมตี ผลลัพธ์ของตัวแรก — หาบั๊ก กรณีขอบ (edge case) ที่ตกหล่น ข้อผิดพลาดที่ไม่ได้จัดการ ฝ่ายตรวจสอบที่ได้รับคำสั่งชี้ไปที่ diff ว่า “สมมติว่าสิ่งนี้ผิด แล้วหาสาเหตุ” จะจับสิ่งที่เอเจนต์ผู้เขียนหลอกตัวเองว่าถูกต้องได้
- การสังเคราะห์ (synthesis) รอบสุดท้ายรวบรวมสิ่งที่ค้นพบและสร้างผลลัพธ์ที่สอดคล้องกันหนึ่งเดียว — โค้ดที่ใช้งานได้พร้อมกับบันทึกสั้นๆ ว่าฝ่ายตรวจสอบพบอะไรและได้รับการแก้ไขอย่างไร

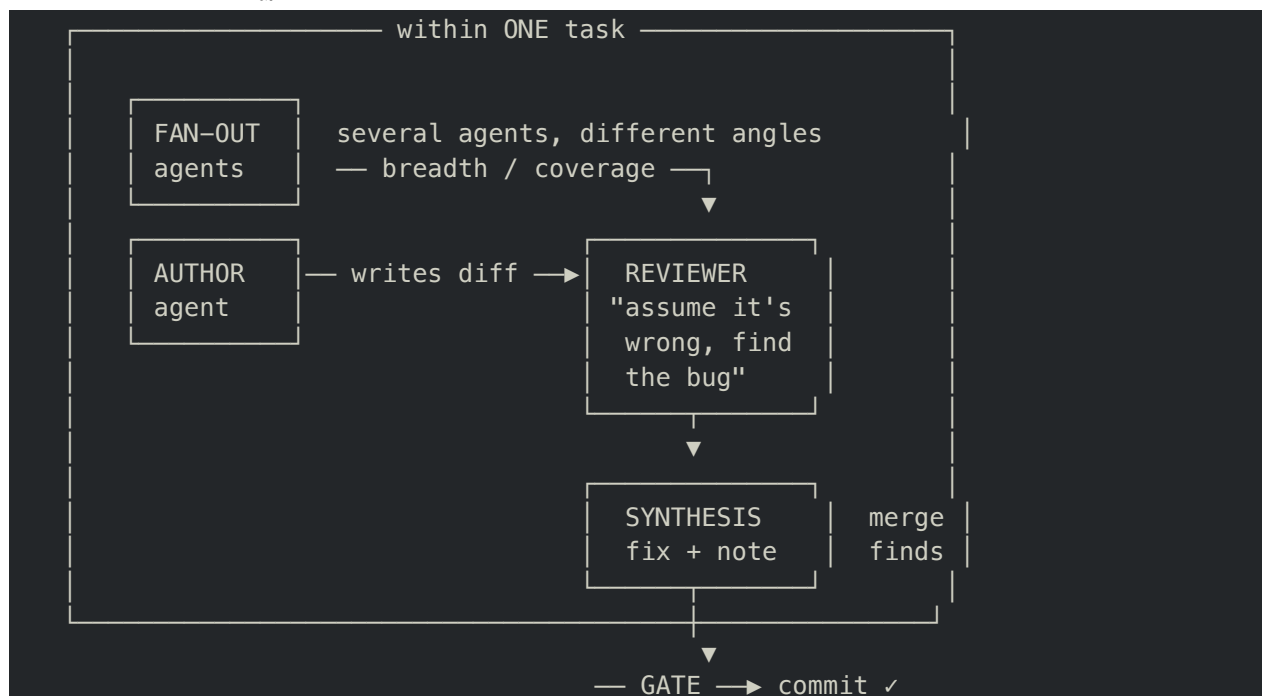
Reviewer agent brief:

You are reviewing a diff written by another agent. Assume it contains at least one bug and find it. Specifically check:

- Does it actually satisfy the original goal, or just pass the tests?
- Edge cases: empty input, nulls, the largest realistic input.
- Anything deleted or changed that the task didn't ask for.

Report concrete problems with file and line. Do not rewrite the code — just report. If you find nothing after a genuine look, say so plainly.

ทั้งสามบทบาทนี้สร้างเป็นสายพานการผลิต (pipeline) ที่มีด่านตรวจ (gate) อยู่ท้ายสุด: จะไม่มีอะไรไปถึง commit จนกว่าฝ่ายตรวจสอบปฏิบัติจะได้ตรวจแล้ว และการสังเคราะห์ได้แก้ไขสิ่งที่พบแล้ว



ฝ่ายตรวจสอบปฏิบัติจะคุ้มค่าที่จะตั้งขึ้นมา แม้ว่าคุณจะเป็นคนเดียวที่ไม่มีกองทัพอเจนต์ — เพราะผู้เขียนของการเปลี่ยนแปลงนั้นเป็นคนที่แย่ที่สุดในการหาข้อบกพร่องของตัวเอง ไม่ว่าจะเป็นมนุษย์หรือ AI การตรวจสอบรอบสองด้วยคำสั่งแบบเป็นปฏิบัติคือประกันภัยราคาถูกลง

การตรวจสอบผลลัพธ์ในระดับใหญ่

นี่คือปัญหาที่ต้องยอมรับตรงๆ: เมื่อเอเจนต์กำลังเขียนโค้ดหลายร้อยบรรทัดในหลายงานพร้อมกัน คุณไม่สามารถอ่านทุกบรรทัดแบบที่คุณตรวจสอบ diff สิบบรรทัดได้ การเสแสร้งว่าคุณทำได้นำไปสู่การประทับตราอนุมัติโดยไม่ดู (rubber-stamping) หรือไม่ก็อัมพาต (paralysis — ไม่เคยไว้ใจอะไรเลย) ไม่มีทางไหนที่จะส่งมอบงานได้จริง

ทางออกคือการตั้งด่านตรวจ (gate) เป็นชั้นๆ แทนการอ่านทุกอย่าง:

- ชุดทดสอบคือพื้นฐานขั้นต่ำ ผลลัพธ์ของเอเจนต์จะไม่ถูกรวมเข้ากับโค้ดหลักโดยไม่มีชุดทดสอบที่เกี่ยวข้องเป็นสีเขียวและตัวตรวจสอบชนิดข้อมูล (type-checker) สะอาด นี่คือนี่สิ่งที่ต้องรอไม่ได้ และมันเป็นแบบอัตโนมัติ — เครื่องจักรทำส่วนนี้ให้
- ตรวจสอบจุดเสี่ยงอย่างละเอียด คุณอ่านทั้งหมดไม่ได้ ดังนั้นให้อ่านส่วนที่รัศมีความเสียหายสูง: อะไรก็ตามที่แตะการยืนยันตัวตน เงิน การลบข้อมูล หรือการเรียกใช้ระบบภายนอก ให้อ่านทุกบรรทัดที่เหลือนเพื่อหาสัญญาณเตือนจากบทเรื่องการสร้างฟีเจอร์ — ไฟล์ที่ไม่ควรถูกแตะ สิ่งที่ถูกกลับไปเจียบๆ ค่าขอลึกๆ ที่กลับสร้าง diff ขนาดใหญ่

- อ่าน diff ไม่ใช่โค้ดเบสทั้งหมด `git diff --stat` บอกคุณได้ในไม่กี่วินาทีว่า อะไร เปลี่ยนไปและเปลี่ยนมากแค่ไหน งานที่กำหนดขอบเขตไว้แค่โมดูลเดียวแต่แสดงการเปลี่ยนแปลงในหกโมดูลคือสัญญาณให้ดูให้ละเอียดขึ้น ก่อนที่คุณจะอ่านแม้แต่บรรทัดเดียว
- รันมันดู ชุดทดสอบสีเขียวไม่ใช่แอปที่ใช้งานได้จริง คลิกลิงก์นั้น การตรวจสอบขั้นสุดท้ายเป็นอันเดียวกับทุกบทย่อยอื่นเสมอ: มันทำในสิ่งที่คุณขอจริงๆ หรือไม่ ต่อหน้าคุณ?

การกำหนดขอบเขตให้ผลตอบแทนสองเท่าตรงนี้ งานที่คุณกำหนดขอบเขตไว้แน่นอนจะได้ diff ที่คุณตรวจสอบได้จริง ส่วนงานที่คุณปล่อยหลวมจะได้ diff ที่คุณตรวจสอบไม่ได้ ปัญหาการตรวจสอบถูกแก้ไขไปแล้วเป็นส่วนใหญ่ที่ต้นทาง คือในรีพอนันเอง

ค่าใช้จ่ายและวงจรที่วิ่งไม่หยุด

เอเจนต์อัตโนมัติทำงานด้วยการวนซ้ำ (loop): ลอง เช็ก ลองอีกครั้ง วงจรนั้นคือแก่นของมัน — และมันก็เป็นสิ่งที่อาจทำให้บิลพุ่งหรือวิ่งไม่หยุดโดยที่คุณไม่รู้ตัวเช่นกัน

เรื่องนี้เชื่อมโยงกับแนวคิดความปลอดภัยด้านค่าใช้จ่ายจากก่อนหน้านี้: เครื่องมือแบบสมัครสมาชิกราคาคงที่ (flat-rate subscription) ล้มเหลวด้วยการหยุดทำงาน ส่วนเอเจนต์ที่ใช้ paid API แบบวัดปริมาณ (metered) ล้มเหลวด้วยการเรียกเก็บเงิน เมื่อผู้ขายเขียนโค้ดแบบสมัครสมาชิกถึงขีดจำกัด มันก็แค่หยุดทำงานเพิ่มจนกว่าช่วงเวลาจะรีเซ็ต แต่เมื่อสคริปต์ที่คุณเขียนขึ้นเรียกใช้ paid API ติดอยู่ในวงจรการลองใหม่ (retry loop) มันจะเรียกต่อไป — และทุกการเรียกมีค่าใช้จ่าย — จนกว่าคุณจะสังเกตเห็นหรือบัตร์ของคุณเต็มวงเงิน

ดังนั้นกฎจึงตรงไปตรงมา:

- อย่าปล่อยไว้วงจรเอเจนต์ที่ไม่มีการควบคุมรันกับ paid API ถ้าคุณเขียนวงจรของตัวเองที่เรียกใช้ API ของ LLM (Large Language Model — โมเดลภาษาขนาดใหญ่ ซึ่งเป็นชนิดของ AI ที่อยู่เบื้องหลังเครื่องมือเขียนโค้ดเหล่านี้) มันต้องมีเพดานที่แน่นอน — จำนวนรอบสูงสุด วงเงินใช้จ่าย เวลาหมดอายุ (timeout) ไม่มีข้อยกเว้น ไม่มี “เดี๋ยวฉันจะเฝ้าดูมัน” คุณจะไม่ทำแบบนั้นหรอก
- ให้ทุกวงจรมีทางออก “หยุดหลังจาก N ครั้ง” “หยุดถ้าชุดทดสอบยังล้มเหลวสองครั้งติดกัน” “หยุดถ้ารันมาสิบนาทีแล้ว” เอเจนต์ที่ไม่มีเงื่อนไขหยุดคือบัก ไม่ใช่ฟิเจอร์
- เฝ้าดูการรันครั้งแรก ก่อนที่คุณจะไว้ใจให้เอเจนต์รันในพื้นหลัง ให้เฝ้าดูการรันในพื้นหน้า (foreground) สองสามครั้งตั้งแต่ต้นจนจบ คุณกำลังตรวจสอบว่ามัน ลู่เข้า (converges) — คือมันเข้าใกล้ความสำเร็จล้นมากขึ้น ไม่ใช่ติดวนซ้ำแก้จุดเดิม
- เลือกเครื่องมือที่มีรั้วกันความปลอดภัย ผู้ช่วยที่มีการจัดการพร้อมขีดจำกัดในตัวปลอดภัยกว่าที่จะปล่อยให้รันทิ้งไว้ เมื่อเทียบกับวงจร API ดิบๆ ที่คุณเขียนตอนที่ยังคีน การที่ระบบสมัครสมาชิกล้มเหลวแบบปิด (fail closed) คือข้อดี ส่วน API ที่ล้มเหลวแบบเปิด (fail open) คืออันตราย

ข้อจำกัดที่ต้องยอมรับจริงๆ

เอเจนต์คือรูปแบบแรงยกที่ทรงพลังที่สุดในหนังสือเล่มนี้ ซึ่งเป็นเหตุผลว่าทำไมจึงคุ้มค่าที่จะมองให้ชัดว่ามันคืออะไรและไม่ใช่อะไร

พวกมันไม่ใช่ความเป็นอิสระที่คุณหยุดทำความเข้าใจได้ ในวินาทีที่คุณตอบคำถาม “โค้ดนี้ควรทำอะไร และฉันจะรู้ได้อย่างไรว่ามันผิด” ไม่ได้อีกต่อไป คุณก็ได้ข้ามจากการใช้แรงยกไปเป็นการหวังพึ่งโชคแล้ว ทุกชั้นในบทย่อย — การกำหนดขอบเขต รั้วกันความปลอดภัย ชุดทดสอบ การตรวจสอบจุดเสี่ยง เพดานค่าใช้จ่าย — มีอยู่เพื่อให้คุณอยู่ฝั่งของความเข้าใจ ในขณะที่ปล่อยให้เอเจนต์ทำงานพิมพ์โค้ดให้มากขึ้นมาก

เพดานทักษะยังเหมือนกับทั้งเล่ม: ความสามารถของคุณในการบอกสิ่งที่คุณต้องการอย่างแม่นยำ และการรู้จำเมื่อคุณได้มันมาแล้ว เอเจนต์ขยายความสามารถนั้น มันไม่ได้แทนที่มัน คนที่โดนไฟลวกไม่ใช่คนที่ใช้เอเจนต์ — แต่เป็นคนที่ใช้เอเจนต์เพื่อหยุดคิด อย่าเป็นแบบนั้น ส่งมอบงานไป แต่รักษาการตัดสินใจไว้กับตัวเอง

สรุปและแบบฝึกหัด

ประเด็นสำคัญ

- เอเจนต์ขยายความสามารถของคุณในการบอกสิ่งที่ต้องการอย่างแม่นยำและรู้จำเมื่อคุณได้มันมาแล้ว — พวกมันไม่ได้แทนที่ความสามารถนั้น
- การกำหนดขอบเขตให้ผลตอบแทนสองเท่า: งานที่กำหนดขอบเขตไว้แน่นจะได้ diff ที่คุณตรวจสอบได้จริง ส่วนงานที่ปล่อยหลวมจะได้ diff ที่คุณตรวจสอบไม่ได้
- อ่าน diff ไม่ใช่โค้ดเบสทั้งหมด — `git diff --stat` ตรวจสอบขอบเขตที่บานปลายได้ในไม่กี่วินาที และชุดทดสอบสีเขียวไม่มีวันแทนที่การรันจริง
- อย่าปล่อยให้อัปเดตเอเจนต์ที่ไม่มีการควบคุมรันกับ paid API ให้ทุกวงจรมีเพดานที่แน่นอน — จำนวนรอบสูงสุด วงเงินใช้จ่าย หรือเวลาหมดอายุ
- ฝึการรันครั้งแรกๆ ในพื้นหน้าเพื่อยืนยันว่าเอเจนต์ ลู่เข้า ก่อนที่คุณจะไว้วางใจให้มันรันในพื้นที่หลัง

ลองทำดู

หยิบงานเล็กๆ ที่กำหนดขอบเขตไว้ดี แล้วมอบให้เอเจนต์พร้อมเงื่อนไขหยุดที่ชัดเจน (“หยุดหลังจาก 3 ครั้ง” หรือ “หยุดถ้าชุดทดสอบยังล้มเหลวสองครั้งติดกัน”) ฝึการรันตั้งแต่ต้นจนจบในพื้นที่หน้า จากนั้นตรวจสอบเฉพาะ diff ด้วย `git diff --stat` ตามด้วย `git diff` ถ้าการเปลี่ยนแปลงแต่ละไฟล์มากกว่าที่คุณกำหนดขอบเขตไว้ นั่นคือสัญญาณให้คุณรีดรีฟให้แน่นขึ้นแล้วรันใหม่

พรมปิระจําบท

```
I'm giving you an autonomous task. Obey these guardrails exactly:
- SCOPE: only touch [list the files/modules]. If the fix needs more,
  STOP and tell me instead of expanding scope.
- GOAL: [one concrete, checkable outcome].
- STOP CONDITIONS: stop after 3 attempts, OR if tests fail twice in a
  row, OR if you've made changes outside the scope above.
- COST: do not call any paid API in a loop without a hard cap.
- REPORT: when you stop, show me `git diff --stat`, the test result,
  and one sentence on whether you converged or got stuck.
Begin, and narrate each attempt so I can watch you converge.
```