FREE

$ vibe-coding --for noncoders

Vibe Coding สำหรับคนไม่ใช่โปรแกรมเมอร์

ใช้ Claude Code สร้าง landing page, mini app, และ prototype จริงโดยไม่ต้องเขียนโค้ด

11 บท · 8 PLAYBOOKS · 42 ภาพประกอบ

```
claude-code · session
$ claude-code build landing-page
✓ Created index.html · ปลื้มไหม
✓ Deployed → vercel.app
$ ship it
```

$ VIBECODINGTHAILAND.COM

2026 EDITION

สารบัญ

// บทนำ

เริ่มอ่านตรงนี้ — วิธีใช้หนังสือเล่มนี้

คำนำสั้น ๆ

หนังสือเล่มนี้เขียนสำหรับคนที่มีไอเดีย อยากสร้างเว็บ แอปเล็ก ๆ เครื่องมือภายใน หรือ prototype แต่ไม่ได้เป็น programmer

คุณไม่จำเป็นต้องเริ่มจากการเรียนเขียนโค้ดหลายเดือนก่อน

แต่คุณต้องเรียนรู้บทบาทใหม่:

> คุณไม่ใช่คนพิมพ์โค้ดเองทุกบรรทัด
> คุณคือคนกำกับ AI ให้สร้าง software อย่างมีขอบเขต ตรวจได้ และไม่เสี่ยงเกินไป

ในเล่มนี้ เราใช้ Claude Code เป็นเครื่องมือหลัก

คุณจะได้เรียนรู้วิธี:

- เปลี่ยนไอเดียให้เป็น spec
- เขียน context ให้ AI ไม่ต้องเดา
- ให้ AI build ของเล็ก ๆ ที่ใช้ได้
- test และ debug โดยไม่ต้องอ่าน code ลึก
- deploy ขึ้นออนไลน์
- ตรวจ security ขั้นพื้นฐาน
- รู้ว่าเมื่อไหร่ควรหยุดและให้ developer/security reviewer ช่วย

หนังสือเล่มนี้ไม่ขายฝันว่า AI จะทำให้ software ทุกอย่างง่ายและปลอดภัยโดยอัตโนมัติ

เป้าหมายคือทำให้คุณสร้างของจริงขนาดเล็กได้อย่างรับผิดชอบ

หนังสือนี้เหมาะกับใคร

เหมาะกับคุณถ้า:

- คุณไม่ใช่ programmer แต่กล้าใช้เครื่องมือใหม่
- คุณมีไอเดียอยากทำ landing page, waitlist, dashboard, internal tool หรือ MVP เล็ก ๆ

- คุณอยากใช้ Claude Code เป็นคนลงมือเขียน code ให้
- คุณยอมเรียนรู้การเขียน requirement, test, review diff, debug และ deploy
- คุณอยากเข้าใจความเสี่ยงเรื่อง data, secret, auth, permission และ production แบบภาษาคน

ไม่เหมาะถ้าคุณต้องการ:

- หนังสือสอน JavaScript/React/Postgres ตั้งแต่ศูนย์แบบละเอียด
- คู่มือสร้างระบบ production ขนาดใหญ่
- คู่มือ security professional หรือ pentest
- สูตรลัด “ให้ AI ทำทั้งหมดโดยไม่ต้องตรวจอะไรเลย”

คุณต้องรู้ code ไหม

ไม่ต้องรู้ code ลึก

แต่ต้องรู้ 5 อย่างนี้:

1. คุณกำลังสร้างอะไร
2. user ต้องทำอะไรได้
3. data อะไรถูกเก็บ
4. จะรู้ได้อย่างไรว่ามันทำงานถูก
5. จุดไหนเสี่ยงเกินกว่าจะทำคนเดียว

ถ้าคุณทำ 5 ข้อนี้ได้ คุณใช้ Claude Code ได้ดีกว่าคนที่สั่ง AI แบบคลุมเครือแล้วกด accept ทุกอย่าง

วิธีอ่านแบบไม่หลง

เล่มนี้มี 2 ส่วนใหญ่ ๆ

1. บทหลัก

บทหลักมีไว้ให้อ่านต่อเนื่อง เพื่อเข้าใจ mindset และ workflow

อ่านตามลำดับได้ดีที่สุด:

```
บท 1 → บท 11
```

แต่ถ้าเวลาน้อย ให้อ่านอย่างน้อย:

```
บท 1, 2, 4, 5, 6, 7, 9, 10, 11
```

ถ้ามีเวลาแค่ 2 ชั่วโมง ให้เปิด `front-matter/03-two-hour-path.md` ก่อน แล้วค่อยกลับมาอ่านบทหลักตามลำดับ

2. Appendices / Playbooks

Appendix ไม่ได้มีไว้ให้อ่านรวดเดียวทั้งหมด

ให้ใช้เหมือนคู่มือเปิดดูตอนติดงานจริง

ตัวอย่าง:

```
ติดตั้งไม่เป็น → เปิด Setup Playbook
จะ deploy → เปิด Vercel Playbook
จะใช้ Supabase → เปิด Supabase Waitlist Playbook
app พัง → เปิด Debug Evidence Playbook
จะ merge/change ใหม่ → เปิด GitHub PR Workflow
```

อย่าพยายามอ่าน appendix ทั้งหมดก่อนเริ่ม เพราะจะรู้สึกเยอะเกินไป

Reader Roadmap: อยากทำอะไร ให้อ่านอะไร

ถ้าคุณเริ่มจากศูนย์จริง ๆ

อ่าน:

1. บท 1 — Vibe Coding คืออะไร
2. บท 2 — Claude Code ทำงานยังไง
3. บท 3 — เตรียมเครื่อง
4. Appendix H — Setup Playbook
5. Appendix B — Starter Files

เป้าหมาย:

> ติดตั้งพร้อม มี project folder มี Git save point และเริ่ม Claude Code ได้

ถ้าคุณอยากทำ landing page

อ่าน:

1. บท 4 — Project Context / `CLAUDE.md`
2. บท 5 — เขียน Spec
3. บท 6 — Project Landing Page
4. Appendix A — Prompt Templates
5. Appendix D — Vercel Deploy Playbook

เป้าหมาย:

> ได้ landing page ที่ test ได้ และ deploy ขึ้น Vercel ได้

ถ้าคุณอยากเก็บ email / waitlist

อ่าน:

1. บท 7 — Waitlist App with Supabase
2. บท 10 — Security Checklist
3. Appendix E — Supabase Waitlist Playbook
4. Appendix D — Vercel Deploy Playbook
5. Appendix F — Debug Evidence Playbook ถ้า submit แล้วพัง

เป้าหมาย:

> มี form ที่เก็บข้อมูลลง Supabase โดยไม่ expose secret และไม่เปิดข้อมูลคนอื่นให้ public อ่าน

ถ้าคุณอยากทำ internal dashboard

อ่าน:

1. บท 8 — Internal Dashboard
2. บท 10 — Security Checklist
3. Appendix E — Supabase Waitlist Playbook ถ้า dashboard ใช้ข้อมูล waitlist
4. Appendix G — GitHub PR + Preview Workflow

ข้อควรจำ:

> `dashboard ที่มีข้อมูล user ไม่ควร deploy public ถ้ายังไม่มั่นใจเรื่อง auth/permission`

ถ้า app พังหรือ deploy fail

อ่าน:

1. บท 9 — Debug แบบ non-coder
2. Appendix F — Debug Evidence Playbook
3. Appendix D — Vercel Deploy Playbook ถ้าเกี่ยวกับ deploy
4. Appendix E — Supabase Waitlist Playbook ถ้าเกี่ยวกับ database/RLS

เป้าหมาย:

> `เก็บหลักฐานให้ครบก่อนให้ Claude แก้ ไม่ใช่สั่ง “มันพัง แก้หน่อย”`

ถ้าจะเปิดให้คนอื่นใช้จริง

อ่าน:

1. บท 10 — Security Checklist
2. บท 11 — Launch, Iterate, and Hand Off
3. Appendix D — Vercel Deploy Playbook
4. Appendix F — Debug Evidence Playbook
5. Appendix G — GitHub PR + Preview Workflow

เป้าหมาย:

> เปิดแบบ small launch, มี rollback plan, มี feedback loop, และรู้ว่าความเสี่ยงอยู่ระดับไหน

ลำดับ appendix ที่แนะนำให้ใช้จริง

ถึงชื่อไฟล์จะเรียง A–H ตามตอนเขียน แต่เวลาใช้งานจริง แนะนำคิดตามลำดับนี้:

1. Appendix H — Setup Playbook
2. Appendix A — Prompt Templates
3. Appendix B — Starter Files
4. Appendix D — Vercel Deploy Playbook
5. Appendix E — Supabase Waitlist Playbook
6. Appendix F — Debug Evidence Playbook
7. Appendix G — GitHub PR + Preview Workflow

พูดง่าย ๆ:

> Setup ก่อน
> Prompt/Starter ต่อ
> Deploy/Data/Debug/Release ตามงานจริง

โปรเจกต์ที่แนะนำให้ทำระหว่างอ่าน

ถ้าอยากได้ผลลัพธ์จริงจากเล่มนี้ ให้ทำ project เดียวพอ:

> Landing page + waitlist form

Version แรกควรมีแค่นี้:

- หน้า landing page 1 หน้า
- CTA ชัด

- form เก็บ email
- success/error state
- deploy บน Vercel
- เก็บข้อมูลใน Supabase
- RLS/policy แบบ public insert-only
- launch note

อย่าเพิ่งทำ:

- login
- payment
- dashboard public
- AI automation ซับซ้อน
- multi-role permission
- file upload

ทำของเล็กให้จบก่อน แล้วค่อย iterate

กติกาความปลอดภัยก่อนเริ่ม

จำ 7 ข้อนี้ไว้ตลอดเล่ม:

1. อย่า paste secret/API key จริงลง prompt ถ้าไม่จำเป็น
2. อย่า commit `.env` หรือ `.env.local`
3. อย่าใส่ service role / secret key ใน frontend
4. อย่าปิด RLS เพื่อแก้ปัญหา production แบบถาวร
5. อย่า deploy dashboard ที่มีข้อมูล user โดยไม่มี auth/permission ที่เข้าใจ
6. ถ้า production พังและมี user จริง ให้คิดเรื่อง rollback ก่อน debug ยาว ๆ
7. ถ้ามี payment, sensitive data, หรือระบบสำคัญ ให้ developer/security reviewer ช่วยตรวจ

วิธีใช้ Claude ระหว่างอ่าน

คุณสามารถเปิด Claude Code แล้วใช้ prompt นี้เป็นตัวช่วยอ่านหนังสือได้:

```
ฉันกำลังอ่านหนังสือ Vibe Coding สำหรับคนไม่ใช่โปรแกรมเมอร์
ช่วยเป็น coach ให้ฉันทำตามทีละขั้น

บริบท:
- ฉันเป็น non-coder
- project ที่อยากทำคือ: [อธิบาย]
- ตอนนี้อ่านถึงบท/appendix: [ชื่อบท]

กติกา:
- ถามฉันทีละขั้น
- อย่าให้ command ยาว ๆ โดยไม่อธิบาย
- อย่าให้ฉัน paste secret/API key
- ถ้างานใหญ่ ให้ช่วยแตก scope
- ถ้าเสี่ยงเรื่อง data/security/deploy ให้เตือนก่อน
```

สิ่งที่หนังสือนี้ไม่ได้รับประกัน

หนังสือเล่มนี้ไม่ได้รับประกันว่า:

- app ที่ AI สร้างจะปลอดภัย 100%
- prototype จะ production-ready ทันที
- ทุก command จะใช้ได้เหมือนเดิมตลอดไป
- pricing, quota, plan หรือ feature availability จะไม่เปลี่ยน
- คุณจะไม่ต้องให้ developer ช่วยเลยในทุกกรณี

เครื่องมือ AI และ cloud platform เปลี่ยนเร็ว

ก่อน final deploy หรือใช้กับข้อมูลจริง ให้ตรวจ official docs ล่าสุดเสมอ โดยเฉพาะเรื่องราคา, security, environment variables, auth, database policy และ feature availability

---

ถ้าจะจำแค่ประโยคเดียว

จำประโยคนี้:

> ให้ AI เขียน code ได้ แต่อย่าให้ AI ตัดสินใจเรื่อง scope, data, security และ launch แทนคุณ

นี่คือหัวใจของหนังสือเล่มนี้

Glossary / Cheat Sheet สำหรับคนไม่ใช่โปรแกรมเมอร์

ใช้หน้านี้อย่างไร

หน้านี้คือพจนานุกรมสั้น ๆ ของศัพท์ที่เจอบ่อยในเล่ม

ไม่ต้องจำทั้งหมดก่อนเริ่มอ่าน

ให้เปิดกลับมาดูเวลามีคำที่ไม่คุ้น

หลักง่าย ๆ:

> ถ้าไม่เข้าใจศัพท์ อย่าเพิ่งปล่อยผ่าน
> เพราะศัพท์บางคำเกี่ยวกับความปลอดภัย เช่น `secret`, `RLS`, `production`, `rollback`

ศัพท์พื้นฐานของ project

คำ	แปลแบบคนทั่วไป
Project	งานหรือ app ที่คุณกำลังสร้าง
App	เว็บ/เครื่องมือ/ระบบที่ user ใช้
Codebase	ไฟล์ทั้งหมดของ project
Folder	กล่องเก็บไฟล์ในเครื่อง
File	เอกสารหรือ code หนึ่งไฟล์
Root folder	folder หลักของ project
README	ไฟล์อธิบาย project ให้คนอื่นเข้าใจ
`CLAUDE.md`	ไฟล์บอก Claude ว่า project นี้คืออะไรและมีกติกาอะไร

จำง่าย ๆ:

```
README = อธิบาย project ให้คนอ่าน
CLAUDE.md = อธิบาย project ให้ Claude ทำงานไม่มั่ว
```

ศัพท์ของ Git / GitHub

คำ	แปลแบบคนทั่วไป
Git	ระบบ save point ของ project
Repository / repo	กล่องเก็บ code ทั้ง project
GitHub	ที่เก็บ repo บน cloud
Commit	save point ที่ย้อนกลับ/ตรวจได้
Branch	ห้องทดลองสำหรับแก้งานหนึ่งเรื่อง
Main	branch หลัก มักเป็นของที่พร้อม deploy
Push	ส่ง code จากเครื่องขึ้น GitHub
Pull	ดึง code ล่าสุดจาก GitHub ลงเครื่อง
Pull Request / PR	ใบส่งงานให้ตรวจ ก่อนเอาเข้า main
Merge	เอา branch เข้า main
Diff	รายการว่าไฟล์เปลี่ยนอะไรบ้าง

จำง่าย ๆ:

```
Commit = save
Branch = ทดลอง
PR = ตรวจงาน
Merge = เอาเข้า main
```

ศัพท์ของ Claude Code

คำ	แปลแบบคนทั่วไป
Claude Code	AI coding agent ที่ทำงานกับไฟล์ project ได้
Agent	AI ที่ไม่ใช่แค่ตอบ แต่ลงมือทำงานเป็นขั้น ๆ ได้
Prompt	คำสั่งที่คุณพิมพ์ให้ AI
Context	ข้อมูลพื้นหลังที่ AI ใช้เข้าใจงาน
Plan mode	ให้ AI วางแผนก่อนแก้ไฟล์
Permission	การขออนุญาตก่อนทำสิ่งที่กระทบ project
Tool	ความสามารถที่ Claude ใช้ เช่น อ่านไฟล์ แก้ไฟล์ รัน command
Session	บทสนทนา/รอบการทำงานหนึ่งชุดกับ Claude
Checkpoint	จุดย้อนกลับของงานบางช่วง
Diff review	การดูว่า AI เปลี่ยนอะไร ก่อนยอมรับ/commit

จำง่าย ๆ:

> `อย่าให้ Claude แก้ก่อนเข้าใจ plan`
> `อย่า commit ก่อนดู diff`

ศัพท์ของ Terminal / Command

คำ	แปลแบบคนทั่วไป
Terminal	หน้าต่างพิมพ์คำสั่ง
Shell	โปรแกรมที่รับคำสั่ง เช่น zsh, bash, PowerShell
Command	คำสั่งหนึ่งบรรทัด

คำ	แปลแบบคนทั่วไป
Output	ผลลัพธ์ที่ terminal แสดง
Error log	ข้อความ error ที่ช่วย debug
`npm install`	ติดตั้ง package ของ project
`npm run dev`	รัน project ในเครื่อง
`npm run build`	build project เพื่อเช็กว่าพร้อม deploy ไหม
`git status`	ดูว่าไฟล์ไหนเปลี่ยนบ้าง

จำง่าย ๆ:

> `Terminal` ไม่ใช่ที่เดา
> ถ้าไม่เข้าใจ `command` ให้ถาม `Claude` ให้อธิบายก่อนรัน

ศัพท์ของ Web App

คำ	แปลแบบคนทั่วไป
Frontend	ส่วนที่ user เห็นใน browser
Backend	ส่วนหลังบ้านที่จัดการ logic/data
Client	ฝั่ง browser/user
Server	ฝั่งที่รัน logic หลังบ้าน
API	ช่องทางให้ app คุยกับ service หรือ backend
Route	path/หน้า/endpoint เช่น `/dashboard`
Form	ช่องให้ user กรอกข้อมูล
Validation	การตรวจว่าข้อมูลที่กรอกถูกต้องไหม
State	สถานะของ UI เช่น loading, success, error

คำ	แปลแบบคนทั่วไป
Responsive	ใช้ได้ทั้ง desktop และ mobile

จำง่าย ๆ:

```
Frontend = user เห็น
Backend = หลังบ้าน
API = ช่องทางคุยกัน
```

ศัพท์ของ Deploy / Vercel

คำ	แปลแบบคนทั่วไป
Deploy	เอา app ขึ้นออนไลน์
Build	แปลง project ให้พร้อม deploy
Deployment	version หนึ่งที่ถูก deploy แล้ว
Production	เว็บจริงที่ user ใช้
Preview	เว็บทดสอบก่อนขึ้น production
Local	เครื่องของคุณเอง
Environment	สภาพแวดล้อม เช่น local/preview/production
Environment variable / env var	ค่า config ที่แยกจาก code
Custom domain	domain ของคุณเอง เช่น `example.com`
Rollback	ย้อน production กลับ version ก่อนหน้า
Build log	log ตอน build/deploy
Runtime log	log ตอน app ทำงานจริงหลัง deploy

จำง่าย ๆ:

```
Local = เครื่องคุณ
Preview = ทดสอบ
Production = ของจริง
Rollback = ย้อนกลับเมื่อของจริงพัง
```

ศัพท์ของ Database / Supabase

คำ	แปลแบบคนทั่วไป
Database	ที่เก็บข้อมูลของ app
Table	ตารางข้อมูล คล้าย sheet
Row	แถวข้อมูลหนึ่งรายการ
Column / field	ช่องข้อมูล เช่น email, name
Schema	โครงสร้างข้อมูลของ table
Query	คำสั่งอ่าน/เขียนข้อมูล
Insert	เพิ่มข้อมูลใหม่
Select	อ่านข้อมูล
Update	แก้ข้อมูล
Delete	ลบข้อมูล
Supabase	platform ที่ให้ database/auth/storage และ API
RLS	Row Level Security: กฎว่าใครอ่าน/เขียน row ไหนได้
Policy	กฎย่อยของ RLS

จำง่าย ๆ:

```
Table = ตาราง
Row = แถว
Column = ช่อง
RLS = ประตูคุมสิทธิ์ของข้อมูล
```

ศัพท์ของ Key / Secret / Security

คำ	แปลแบบคนทั่วไป
API key	key ให้ app ใช้คุยกับ service
Publishable key	key ที่อยู่ใน public app ได้ แต่ยังต้องมี policy คุม data
Secret key	key ลับที่ห้ามโผล่ใน browser
Service role key	key สิทธิ์สูงมากของ Supabase ห้ามอยู่ frontend
`.env` / `.env.local`	ไฟล์เก็บค่าลับ/config ในเครื่อง
`.env.example`	ไฟล์ตัวอย่าง env ที่ไม่มีค่าจริง
Credential	ข้อมูลที่ใช้เข้าถึงระบบ เช่น password/token/key
Token	หลักฐานการยืนยันตัวตนหรือสิทธิ์
Secret scanning	การตรวจหา secret ที่หลุดใน repo
Rotate key	เปลี่ยน key เก่าเป็น key ใหม่เมื่อสงสัยว่าหลุด
Least privilege	ให้สิทธิ์น้อยที่สุดเท่าที่ต้องใช้

จำง่าย ๆ:

```
Publishable ไม่ได้แปลว่าปลอดภัยเอง
Secret ห้ามอยู่ใน browser/GitHub
Service role ห้ามใช้แก้ปัญหาแบบง่าย ๆ
```

ศัพท์ของ Auth / Permission

คำ	แปลแบบคนทั่วไป
Authentication / Auth	การพิสูจน์ว่า user คือใคร เช่น login
Authorization	การตัดสินว่า user มีสิทธิ์ทำอะไร
Role	บทบาท เช่น public, user, admin
Public user	คนทั่วไปที่ยังไม่ login
Authenticated user	user ที่ login แล้ว
Admin	คนที่มีสิทธิ์สูงกว่า user ทั่วไป
Session	สถานะ login ของ user
MFA / 2FA	การยืนยันตัวตนหลายชั้น

จำง่าย ๆ:

```
Auth = คุณคือใคร
Authorization = คุณทำอะไรได้
```

มี login แล้วไม่ได้แปลว่าปลอดภัย ถ้า permission ยังเปิดกว้าง

ศัพท์ของ Debug

คำ	แปลแบบคนทั่วไป
Bug	สิ่งที่ควรทำงานแต่ไม่ทำงาน
Debug	หาสาเหตุและแก้ bug
Reproduce	ทำให้ bug เกิดซ้ำได้
Expected result	สิ่งที่ควรเกิด

คำ	แปลแบบคนทั่วไป
Actual result	สิ่งที่เกิดจริง
Browser console	ที่ browser แสดง error ของหน้าเว็บ
Network tab	ที่ดู request/response ของเว็บ
Status code	เลขบอกผล request เช่น 404, 500
400	request ผิดรูปแบบ
401	ยังไม่ login/ไม่มี credential ที่ถูกต้อง
403	ไม่มี permission
404	หาไม่เจอ
409	ข้อมูลชนกัน เช่น duplicate
429	ส่งถี่เกิน limit
500	server error
Root cause	สาเหตุจริงของปัญหา
Hotfix	แก้ด่วนเฉพาะจุด

จำง่าย ๆ:

> `Debug ที่ดีเริ่มจาก evidence ไม่ใช่เดา`

ศัพท์ของ Product / Workflow

คำ	แปลแบบคนทั่วไป
Spec	เอกสารบอกว่าจะสร้างอะไร
Requirement	สิ่งที่ app ต้องทำได้

คำ	แปลแบบคนทั่วไป
Acceptance criteria	เงื่อนไขว่าถือว่างานเสร็จเมื่อไหร่
Scope	ขอบเขตงานรอบนี้
Out of scope	สิ่งที่ยังไม่ทำในรอบนี้
MVP	version เล็กสุดที่ทดสอบ idea ได้
Prototype	version ทดลอง
Beta	เปิดให้คนกลุ่มเล็กลองใช้
Feedback	ความเห็น/ปัญหาจาก user
Backlog	รายการงานที่รอทำ
Priority	ลำดับความสำคัญ
P0	ด่วนมาก/blocker
P1	สำคัญ ควรทำเร็ว
P2	ทำได้เมื่อมีเวลา
Later	เก็บไว้ก่อน
Handoff	ส่งต่อ project ให้ developer/team

จำง่าย ๆ:

```
Scope ชัด = AI เดาน้อยลง
Acceptance criteria ชัด = test ได้
```

คำที่ควรระวังเป็นพิเศษ

ถ้าเห็นคำเหล่านี้ ให้ช้าลงและอ่านให้เข้าใจ:

```
secret
service_role
RLS
production
deploy
rollback
database
auth
authorization
payment
env var
```

เพราะคำเหล่านี้เกี่ยวกับ data, security, หรือของจริงที่ user ใช้

ถ้าไม่แน่ใจ ให้ใช้ prompt นี้:

```
ช่วยอธิบายคำนี้แบบ non-coder:
[คำศัพท์]

บอกด้วยว่า:
1. มันคืออะไร
2. ใช้ทำอะไรใน project นี้
3. เสี่ยงตรงไหน
4. ฉันต้องตัดสินใจอะไรไหม
```

Cheat Sheet: 10 ประโยคที่ควรจำ

1. AI เขียน code ได้ แต่คุณต้องคุม scope
2. Spec ชัดกว่า prompt ยาว ๆ แบบไม่ตัดสินใจ
3. `CLAUDE.md` คือบริบทถาวรของ project
4. ดู diff ก่อน commit
5. อย่า commit `.env`
6. Secret ห้ามอยู่ใน frontend
7. RLS สำคัญถ้าใช้ Supabase จาก browser
8. Preview ก่อน production ถ้าเลี่ยงได้
9. Error คือหลักฐาน ไม่ใช่ความล้มเหลว

10. ถ้าเสี่ยงกับเงินจริง ข้อมูลจริง หรือ user จริง ให้คนช่วย review

Two-Hour Path — ถ้ามีเวลาแค่ 2 ชั่วโมง

หน้านี้มีไว้สำหรับคนที่อยากเริ่มให้เร็วที่สุด โดยไม่อ่านทั้งเล่มก่อน

เป้าหมายของ 2 ชั่วโมงนี้ไม่ใช่สร้าง app สมบูรณ์

เป้าหมายคือให้คุณเข้าใจ workflow ที่ต้องใช้จริง:

```
Idea → Spec → Claude วาง plan → Build เล็ก → Test → Diff → Deploy/Preview → Decide
```

ถ้าคุณทำ flow นี้ได้ 1 รอบ คุณจะอ่านส่วนที่เหลือของเล่มเข้าใจง่ายขึ้นมาก

สิ่งที่ต้องจำก่อนเริ่ม

```
คุณไม่ต้องเขียน code เป็น
แต่คุณต้องสั่งงาน ตรวจงาน และรู้ว่าเมื่อไหร่ควรหยุด
```

กฎ 5 ข้อ:

1. อย่าเริ่มจาก project สำคัญ
2. อย่าให้ AI แก้ไฟล์ก่อนอธิบาย plan
3. อย่า approve command ที่ไม่เข้าใจ
4. อย่าเอา secret/API key ลง GitHub หรือ chat
5. ถ้าเกี่ยวกับข้อมูลคนจริง ต้องคิดเรื่อง permission/RLS ก่อน

นาที 0–15: อ่าน mindset ให้ถูก

อ่าน:

- บท 1 — Vibe Coding คืออะไร และไม่ใช่อะไร
- บท 2 — Claude Code ทำงานยังไง แบบที่ non-coder ต้องรู้

คำถามที่ต้องตอบให้ได้:

> Claude Code แก้ไฟล์ได้จริงไหม?
> ฉันต้องตรวจอะไรเอง?
> งานแบบไหนไม่ควรให้ AI ทำแบบปล่อยอัตโนมัติ?

ถ้าตอบไม่ได้ ให้ยังไม่เริ่ม project จริง

นาที 15–30: เช็กว่าเครื่องพร้อมไหม

อ่าน:

- บท 3 — เตรียมเครื่องและเริ่มโปรเจกต์แรก
- Appendix H — Setup Playbook เฉพาะถ้ายัง setup ไม่เป็น

ผลลัพธ์ที่ควรได้:

> เปิด terminal ได้
> เปิด project folder ได้
> รัน claude ได้
> รู้ว่า git status ใช้ดูไฟล์ที่เปลี่ยน
> รู้ว่า project แรกต้องพังได้

ถ้ายังติดตั้ง Claude Code ไม่ได้ อย่าข้าม ให้แก้ setup ก่อน

นาที 30–50: ทำให้ AI ไม่ต้องเดา

อ่าน:

- บท 4 — Project Context / CLAUDE.md
- บท 5 — เขียน Spec ที่ AI เอาไปสร้างได้

สร้าง 2 อย่างนี้ก่อน build:

```
CLAUDE.md = กติกาและบริบทของ project
Spec = สิ่งที่จะสร้างใน version นี้
```

Prompt สั้น:

```
ฉันเป็น non-coder
ช่วยอ่าน project นี้ก่อน อย่าเพิ่งแก้ไฟล์
จากนั้นช่วยถามคำถามที่ต้องรู้ก่อนเขียน spec version แรก
```

นาที 50–95: สร้าง landing page เล็ก ๆ

อ่าน:

- บท 6 — Landing Page ที่ publish ได้
- Appendix A — Prompt Templates เฉพาะ template ที่ต้องใช้

Scope ที่แนะนำ:

```
หน้าเดียว
ไม่มี login
ไม่มี payment
ไม่มี database
มี headline, benefit, CTA, form mock, thank you state
```

Stop rule:

```
ถ้า Claude เสนอ database, auth, payment, dashboard หรือ integration ตั้งแต่รอบแรก
ให้ลด scope ทันที
```

นาที 95–115: Test และ debug แบบไม่มั่ว

อ่าน:

- บท 9 — Debug แบบคนไม่ใช่ programmer
- Appendix F — Debug Evidence Playbook ถ้ามี error

ก่อนบอกว่า “พัง” ให้เก็บ:

> เกิดอะไรขึ้น
> ควรเกิดอะไร
> ทำซ้ำยังไง
> `error/log` คืออะไร
> เพิ่งเปลี่ยนอะไร
> อยู่ `local` หรือ `production`

อย่าให้ Claude แก้จากคำว่า “มันพัง” อย่างเดียว

นาที 115–120: ตัดสินใจว่าจะไปทางไหนต่อ

ถ้าทำ landing page ได้แล้ว เลือกทางต่อ:

ทาง A — อยาก deploy

อ่าน:

- Appendix D — Vercel Deploy Playbook
- บท 11 — Launch, Iterate, and Hand Off

ทาง B — อยากเก็บ lead จริง

อ่าน:

- บท 7 — Waitlist App ที่เริ่มมี Database
- Appendix E — Supabase Waitlist Playbook
- บท 10 — Security Checklist

ทาง C — อยากทำ dashboard

อ่าน:

- บท 8 — Internal Dashboard

แต่ให้จำว่า dashboard เป็น optional/bonus และมี risk สูงกว่า landing page

ถ้ายังไม่มั่นใจเรื่อง auth/RLS/secret key ให้ข้ามก่อน

ไม่ต้องอ่านตอนนี้

ถ้ามีเวลาแค่ 2 ชั่วโมง ข้ามได้ก่อน:

> บท 8 — Dashboard ถ้ายังไม่มีข้อมูลจริง
> รายละเอียดลึกของ performance/accessibility
> การ integrate CRM/payment/email automation

กลับมาอ่านเมื่อ project เริ่มจริงจังขึ้น

เป้าหมายหลัง 2 ชั่วโมง

หลังจบ path นี้ คุณควรมีอย่างน้อยหนึ่งอย่าง:

- project folder ที่พร้อมใช้ Claude Code
- spec version แรก
- landing page draft
- checklist ว่าต้อง test อะไร
- ความเข้าใจว่าอะไรควรทำต่อ และอะไรควรหยุดก่อน

ถ้าคุณได้แค่นี้ ถือว่าสำเร็จแล้ว

ไม่ต้องรีบทำ app ใหญ่

ให้ทำ workflow เล็กให้จบก่อน

$ บทที่ 1

Vibe Coding คืออะไร และไม่ใช่อะไร

แก่นของบทนี้

Vibe Coding สำหรับคนไม่ใช่โปรแกรมเมอร์ไม่ใช่การ “ให้ AI ทำทุกอย่างแทนเรา”

มันคือการเปลี่ยนบทบาทของเราจากคนเขียนโค้ดเอง ไปเป็นคนที่ **สั่งงาน ตรวจงาน และตัดสินใจ** ให้ชัดพอที่ AI coding agent จะลงมือสร้าง software ได้

ในเล่มนี้ agent หลักของเราคือ **Claude Code**

Claude Code เป็นเครื่องมือเขียนโค้ดแบบ agentic ที่อ่าน codebase, แก้ไฟล์, รันคำสั่ง และทำงานร่วมกับเครื่องมือพัฒนา software ได้

แต่สิ่งสำคัญคือ:

> AI ช่วยลงมือได้เร็วขึ้น แต่ไม่ได้ช่วยรับผิดชอบแทนเรา

หน้าที่ของคุณคือ:

> คิดให้ชัด
> สั่งให้เป็น
> ตรวจให้ได้
> ลด scope ให้เล็กพอ
> รู้ว่าอะไรเสี่ยงเกินไป

“ไม่ต้องรู้โค้ด” ไม่ได้แปลว่า “ไม่ต้องรู้อะไรเลย”

คุณไม่จำเป็นต้องรู้ JavaScript, React, database หรือ API ตั้งแต่ต้น

แต่คุณต้องตอบคำถามเหล่านี้ได้:

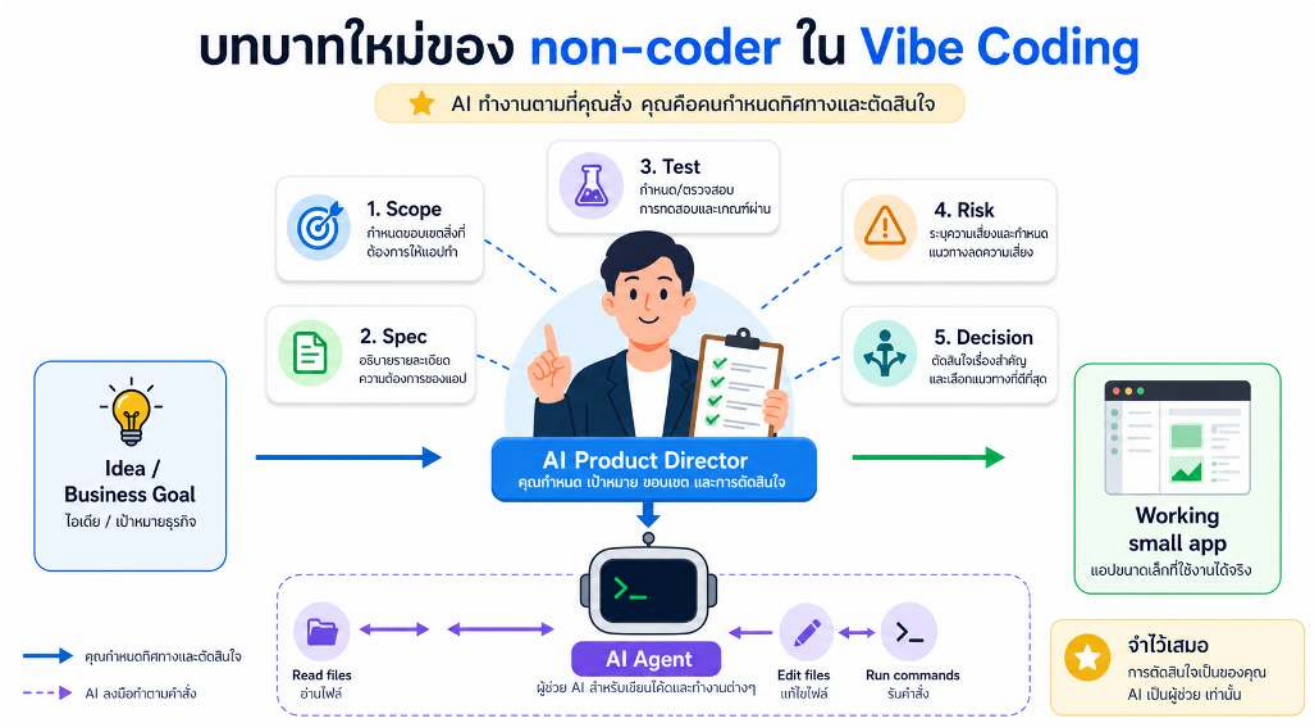
> แอปนี้สร้างให้ใครใช้
> เขาเข้ามาแล้วต้องทำอะไรได้
> `version` แรกต้องมีอะไรบ้าง
> อะไรยังไม่ควรทำตอนนี้
> ต้องเก็บข้อมูลอะไร
> ข้อมูลอะไรไม่ควรเก็บ
> เราจะรู้ได้ยังไงว่ามันใช้ได้จริง
> ถ้ามันพัง จะปิดหรือย้อนกลับยังไง

นี่คือทักษะของคนกำกับ product ไม่ใช่ทักษะของ programmer

ถ้าคุณตอบคำถามเหล่านี้ไม่ได้ AI จะเดาแทนคุณ

และเมื่อ AI เดา ผลลัพธ์อาจดูดี แต่ใช้งานจริงไม่ได้ หรือเสี่ยงโดยที่คุณไม่รู้ตัว

บทบาทใหม่: AI Product Director



บทบาทใหม่ของ non-coder ใน Vibe Coding

แผนภาพนี้สรุปบทบาทของคุณในเล่มนี้: ไม่ต้องเป็น programmer แต่ต้องกำกับ scope, spec, test, risk และ decision

เวลาสั่ง Claude Code ให้คิดเหมือนคุณกำลัง brief developer หนึ่งคน

อย่าสั่งแบบนี้:

> ทำเว็บให้หน่อย เอาสวย ๆ ใช้ง่าย ๆ

สั่งแบบนี้ดีกว่า:

```
ทำ landing page สำหรับบริการ AI Workflow Audit
กลุ่มเป้าหมายคือเจ้าของธุรกิจ SME
เป้าหมายคือให้คนกรอกฟอร์มนัด consult 30 นาที
version แรกยังไม่ต้องมี login หรือ payment
โทนภาษาไทยตรง กระชับ ไม่ hype
```

ความต่างไม่ได้อยู่ที่ความยาวของ prompt

แต่อยู่ที่ความชัดของ 4 เรื่อง:

1. **ใครใช้** — target user คือใคร
2. **ใช้ทำอะไร** — goal ของหน้านี้/app นี้คืออะไร
3. **version แรกมีแค่อะไร** — scope เล็กแค่ไหน
4. **อะไรห้ามพลาด** — data, security, mobile, form, deploy

ถ้าคุณชัด 4 เรื่องนี้ งานจะง่ายขึ้นมาก

งานแบบไหนเหมาะกับ Vibe Coding

เริ่มจากงานที่เล็ก ชัด และตรวจเองได้

ตัวอย่างที่เหมาะ:

- landing page
- waitlist / lead capture form
- calculator หรือ mini tool ง่าย ๆ
- questionnaire / assessment
- internal dashboard ขนาดเล็ก
- prototype หรือ MVP เพื่อทดสอบไอเดีย

งานเหล่านี้ดีเพราะคุณสามารถตรวจ flow เองได้ เช่น เปิดหน้าเว็บ กดปุ่ม กรอกฟอร์ม ดูผลลัพธ์ และแก้จาก feedback ได้เร็ว

หลักการคือ:

> ใช้ AI เพื่อลดระยะจากไอเดียไปสู่ของที่จับต้องได้ ไม่ใช่ใช้ AI เพื่อข้ามขั้นตอนคิด

งานแบบไหนไม่ควรทำเองแบบมั่นใจเกินไป

บางงานอาจใช้ Claude Code ช่วยร่างได้ แต่ไม่ควรปล่อยให้คนใช้จริงโดยไม่มี developer หรือ security reviewer ช่วยดู

โดยเฉพาะ:

- ระบบที่เกี่ยวกับเงินจำนวนมาก
- ระบบที่เก็บข้อมูลสุขภาพ การเงิน เอกสารส่วนตัว หรือข้อมูลลูกค้าที่ confidential
- ระบบ login/permission ซับซ้อน
- ระบบที่พังแล้วธุรกิจหยุด
- ระบบที่ต้อง comply กับกฎเฉพาะทาง

กฎง่าย ๆ คือ:

> ถ้าพังแล้วเสียเงิน เสียข้อมูล หรือเสียความเชื่อใจเยอะ อย่าใช้ AI ทำคนเดียวแล้ว deploy เลย

ความเสี่ยงหลัก: เห็นว่าใช้ได้ แล้วคิดว่าปลอดภัย

กับดักใหญ่ของ vibe coding คือของมัน “ดูเสร็จ” เร็วมาก

หน้าเว็บขึ้น
ปุ่มกดได้
สีสวย
form มี animation
deploy ได้ URL แล้ว

แต่ของที่ดูใช้ได้ ไม่ได้แปลว่าพร้อมใช้จริง

คุณต้องถามต่อ:

> ข้อมูลถูกส่งไปที่ไหน
> ใครอ่านข้อมูลนี้ได้บ้าง
> มี API key หรือ secret หลุดไหม
> database เปิดกว้างเกินไปไหม
> ถ้า user กรอกข้อมูลผิดจะเกิดอะไร
> ถ้า deploy แล้วพัง rollback ได้ไหม

OWASP เรียกความเสี่ยงแบบนี้ว่า **Blind Trust** — การเชื่อ output จากระบบอัตโนมัติหรือ AI โดยไม่ได้ตรวจสอบเอง

สำหรับเล่มนี้ ให้จำประโยคเดียว:

> AI ทำให้สร้างได้เร็วขึ้น แต่ไม่ได้ทำให้เราข้ามการตรวจได้

Workflow ของทั้งเล่ม

เราจะใช้ workflow เดียวซ้ำไปเรื่อย ๆ

```
Idea → Spec → Context → Build → Test → Fix → Deploy → Review
```

แปลเป็นภาษาคน:

1. **Idea** — อยากแก้ปัญหาอะไร
2. **Spec** — version แรกต้องทำอะไรได้บ้าง
3. **Context** — บอก Claude Code ว่า project นี้คืออะไร มีกติกาอะไร
4. **Build** — ให้ Claude Code วางแผนและแก้ไฟล์
5. **Test** — ทดลองใช้เหมือน user จริง
6. **Fix** — เอา error/log/feedback ให้ AI แก้ทีละจุด
7. **Deploy** — เอาขึ้นออนไลน์
8. **Review** — ตรวจอีกครั้งหลัง deploy

จุดสำคัญคือห้ามข้ามจาก Idea ไป Build ทันที

คนส่วนใหญ่พังตรงนี้

เขามีไอเดีย แล้วสั่ง AI ว่า “ทำเลย”

สิ่งที่ควรทำคือ:

> ช่วยถามคำถามก่อน
> ช่วยลด `scope` ก่อน
> ช่วยเขียน `spec` ก่อน
> ช่วยเสนอ `plan` ก่อนแก้ไฟล์

Version แรกควรเล็กมาก

จากไอเดียใหญ่สู่ version แรกที่เล็กพอ

ใช้ภาพนี้จำหลักสำคัญของบทนี้: ตัด feature เสี่ยงออกก่อน แล้วทำ version แรกที่เล็กพอจะ test เองได้

ข้อผิดพลาดของมือใหม่คืออยากให้ version แรกทำได้ทุกอย่าง

เช่น:

```
landing page + login + payment + dashboard + coupon + email automation +
analytics + admin panel
```

ใหญ่เกินไป

เริ่มแบบนี้ดีกว่า:

```
landing page + waitlist form + thank you message + deploy URL
```

เมื่อ version แรกใช้ได้ ค่อยเพิ่มทีละอย่าง

หลักการของเล่มนี้คือ:

> ทำให้เล็กพอจบ ดีกว่าทำให้ใหญ่พอดูเท่แต่ตรวจไม่ได้

Checklist ก่อนเริ่ม

ถ้าจะเริ่ม project ด้วย Claude Code ให้เช็ก 10 ข้อนี้ก่อน

- ☐ อธิบาย version แรกได้ใน 3 ประโยค
- ☐ feature หลักไม่เกิน 3 อย่าง
- ☐ รู้ว่าใครคือ user
- ☐ รู้ว่า user ต้องทำ flow อะไรให้สำเร็จ
- ☐ รู้ว่าต้องเก็บข้อมูลอะไร
- ☐ ไม่เก็บข้อมูลอ่อนไหวถ้าไม่จำเป็น
- ☐ ถ้า app พัง ธุรกิจไม่หยุดทันที
- ☐ ถ้า deploy ผิด สามารถปิดหรือ rollback ได้
- ☐ คุณทดสอบ flow หลักเองได้
- ☐ ถ้า project เสี่ยง คุณรู้ว่าจะให้ใครช่วย review

ถ้าติ๊กได้น้อยกว่า 7 ข้อ ให้ลด scope ก่อน

Prompt เริ่มต้น

ใช้ prompt นี้ก่อนให้ Claude Code เขียนโค้ด

```
ฉันอยากใช้ Claude Code ช่วยสร้าง software ขนาดเล็ก
ฉันไม่ใช่ programmer ดังนั้นอยากให้ช่วยลด scope ให้ชัดและปลอดภัยพอสำหรับ version แรก

ไอเดียของฉัน:
[อธิบายไอเดียแบบภาษาคน]

บริบท:
- กลุ่มผู้ใช้:
- เป้าหมายของ project:
- ข้อมูลที่อาจต้องเก็บ:
- คนที่จะใช้ version แรก:
- สิ่งที่ฉันกังวล:

กติกา:
- ห้ามเริ่มเขียนโค้ด
- ถ้าข้อมูลไม่พอ ให้ถามคำถามก่อน
- ช่วยลด scope ให้เล็กที่สุดแต่ยังมีประโยชน์
- ช่วยบอกสิ่งที่ไม่ควรทำใน version แรก
- ประเมิน risk เรื่อง data, security, payment, login และ deploy
- ใช้ภาษาที่ non-coder เข้าใจ

Output ที่ต้องการ:
1. สรุป project ใน 3 ประโยค
2. version แรกควรมีอะไรบ้าง
3. อะไรควรตัดออกก่อน
4. user flow หลัก
5. data fields ที่ต้องเก็บ
6. risk checklist
7. คำถามที่ต้องตอบก่อนเริ่ม build
```

ถ้า output ยังใหญ่ไป ให้ถามต่อ:

```
ช่วยลด scope ลงอีก 50% โดยยังให้ project มีประโยชน์อยู่
```

สรุป

Vibe coding ไม่ใช่เวทมนตร์

มันคือ workflow ที่ช่วยให้คนไม่ใช่ programmer สร้าง software ขนาดเล็กได้เร็วขึ้น โดยมี AI coding agent ช่วยลงมือ

แต่คุณยังต้องเป็นคนกำกับ:

- scope
- spec
- context
- test
- risk
- decision

บทต่อไป เราจะดูว่า Claude Code ทำงานยังไงแบบที่ non-coder ต้องรู้ เพื่อให้คุณสั่งงานมันได้ดีขึ้น และไม่กลัวเวลาเห็น terminal, file, command หรือ permission prompt

$ บทที่ 2

Claude Code ทำงานยังไง แบบที่ non-coder ต้องรู้

แก่นของบทนี้

คุณไม่จำเป็นต้องเข้าใจ Claude Code ลึกแบบ developer

แต่คุณต้องเข้าใจพอว่าเวลาเราสั่งงาน Claude Code มันไม่ได้แค่ “ตอบแชต”

มันสามารถ:

> อ่านไฟล์
> ค้นหาไฟล์
> แก้ไฟล์
> สร้างไฟล์ใหม่
> รันคำสั่ง
> ดูผลลัพธ์
> แก้ต่อจาก error

นี่คือเหตุผลที่ Claude Code ทรงพลังกว่า chatbot ธรรมดา

แต่ก็เป็นเหตุผลเดียวกันที่คุณต้องใช้มันอย่างระวัง

เพราะสิ่งที่มันทำไม่ใช่แค่ข้อความในหน้าจอ แต่มันอาจเปลี่ยนไฟล์จริงใน project ของคุณได้

Claude Code ไม่ใช่แค่ ChatGPT/Claude ที่ตอบคำถาม

ถ้าคุณใช้ chatbot ปกติ คุณถาม มันตอบ

เช่น:

> ช่วยเขียนโค้ดฟอร์มสมัครสมาชิกให้หน่อย

มันอาจตอบเป็น code block มาให้คุณ copy เอง

แต่ Claude Code ทำงานอีกแบบ

มันอยู่ใน project ของคุณ อ่านไฟล์จริง แก้ไฟล์จริง และรันคำสั่งจริงได้

ตัวอย่างเช่น คุณสั่งว่า:

> `ช่วยเพิ่มฟอร์ม waitlist ในหน้าแรก แล้วให้เก็บชื่อและอีเมล`

Claude Code อาจทำแบบนี้:

1. ดูว่า project ใช้ framework อะไร
2. หาไฟล์หน้าแรก
3. อ่าน component ที่เกี่ยวข้อง
4. แก้ไฟล์เพื่อเพิ่ม form
5. เพิ่ม validation เบื้องต้น
6. รัน build หรือ test
7. ถ้า error ก็อ่าน error แล้วแก้ต่อ
8. สรุปว่ามันแก้อะไรไปบ้าง

นี่คือสิ่งที่เรียกว่า agentic coding

แปลแบบง่าย ๆ คือ AI ไม่ได้แค่ตอบ แต่มัน “ลงมือทำเป็นขั้น ๆ” ได้

วงจรการทำงาน: อ่าน → ทำ → ตรวจ

Claude Code ทำงานเป็นวงจร ไม่ใช่แค่ตอบแชต

วงจรที่ควรใช้ซ้ำทุกครั้ง: อ่าน project → วาง plan → แก้ → test → อธิบาย diff → ให้คนตรวจ

Claude Code ทำงานเป็น loop

```
Gather context → Take action → Verify results
```

แปลเป็นภาษาคน:

```
อ่าน/ทำความเข้าใจ → แก้/สร้าง → ตรวจผล
```

ตัวอย่าง:

```
คุณ: ฟอร์ม submit แล้ว error ช่วยแก้ให้หน่อย

Claude Code:
1. อ่าน error
2. หาไฟล์ที่เกี่ยวข้อง
3. อ่านโค้ดส่วน form
4. แก้ logic
5. รันคำสั่งทดสอบหรือ build
6. ถ้ายังพัง กลับไปอ่าน error ใหม่
7. สรุปผลให้คุณ
```

จุดสำคัญคือคุณอยู่ใน loop นี้ด้วย

คุณสามารถหยุดมันได้
ให้ข้อมูลเพิ่มได้
บอกให้เปลี่ยนทางได้
หรือบอกให้วางแผนก่อนลงมือได้

อย่าคิดว่า Claude Code เป็นคนขับรถคนเดียว

ให้คิดว่ามันเป็นคนขับที่เก่ง แต่คุณยังเป็นคนนั่งข้าง ๆ ที่บอกจุดหมายและคอยดูว่าเลี้ยวผิดไหม

สิ่งที่ Claude Code มองเห็น

โดยทั่วไป เมื่อคุณเปิด Claude Code ใน project หนึ่ง มันอาจเข้าถึงสิ่งเหล่านี้ได้ตาม permission ที่คุณให้:

- ไฟล์ใน project
- โฟลเดอร์ย่อยใน project
- Git state เช่น branch และไฟล์ที่เปลี่ยน
- คำสั่งที่รันได้ใน terminal
- คำสั่ง build/test/dev server
- ไฟล์ instruction เช่น `CLAUDE.md`

แปลว่า Claude Code สามารถเข้าใจ project ได้กว้างกว่าการ copy โค้ดทีละไฟล์ไปถาม chatbot

แต่ก็แปลว่า:

อย่าเปิด project ที่มีข้อมูลลับปนอยู่โดยไม่รู้ตัว

ก่อนให้ Claude Code ทำงานจริง คุณควรรู้ว่า project นี้มีอะไรบ้าง

อย่างน้อยควรรู้ว่า:

```
ไฟล์สำคัญอยู่ตรงไหน
มีไฟล์ .env ไหม
มี API key หรือ secret ไหม
project นี้ commit ขึ้น GitHub หรือยัง
มีไฟล์ที่ไม่อยากให้ AI แตะไหม
```

บทเรื่อง security จะลงรายละเอียดอีกที

ตอนนี้จำแค่ว่า Claude Code อ่านและแก้ project จริงได้ ดังนั้น project ควรสะอาดพอให้ทำงานด้วย

Terminal คืออะไรในเล่มนี้

คำว่า terminal อาจทำให้ non-coder กลัว

แต่ในเล่มนี้ terminal คือแค่ “ช่องสั่งงานเครื่องด้วยข้อความ”

ตัวอย่างคำสั่งที่คุณอาจเห็น:

```bash
npm install
npm run dev
npm run build
git status
```

คุณไม่ต้องจำทุกคำสั่ง

แต่ควรรู้ความหมายคร่าว ๆ:

คำสั่ง	แปลแบบคนทั่วไป
`npm install`	ติดตั้งของที่ project ต้องใช้

คำสั่ง	แปลแบบคนทั่วไป
`npm run dev`	เปิดเว็บในเครื่องเพื่อทดลอง
`npm run build`	ตรวจว่า project build ได้ไหม
`git status`	ดูว่าไฟล์ไหนเปลี่ยน

Claude Code อาจรันคำสั่งพวกนี้ให้คุณ

แต่เวลามันขอ permission คุณไม่ควรกด approve แบบไม่อ่าน

ถ้าไม่เข้าใจ ให้ถามมันก่อน:

> คำสั่งนี้ทำอะไร อธิบายแบบ non-coder ก่อน ฉันควร approve ไหม มีความเสี่ยงอะไรไหม

Tools ที่ควรรู้จักแบบไม่ technical

Claude Code มี tools หลายตัว แต่ non-coder ไม่ต้องจำทั้งหมด

จำแค่กลุ่มนี้พอ

1. Read / Search

ใช้ดูไฟล์และค้นหาใน project

ความเสี่ยงต่ำ เพราะเป็นการอ่าน

เหมาะกับคำสั่งแบบ:

> อ่าน project นี้ก่อน แล้วสรุปว่า app ทำอะไร
> หาไฟล์ที่เกี่ยวกับหน้า landing page
> หา code ที่จัดการ form submit

2. Edit / Write

ใช้แก้ไฟล์หรือสร้างไฟล์ใหม่

เริ่มมีความเสี่ยง เพราะไฟล์จริงถูกเปลี่ยน

ก่อนให้แก้หลายไฟล์ ควรให้วางแผนก่อน

> อย่าเพิ่งแก้ไฟล์ อ่านก่อนแล้วเสนอ `plan` ว่าจะเปลี่ยนไฟล์ไหนบ้าง

3. Bash / command

ใช้รันคำสั่งในเครื่อง เช่น install, build, test, start server, git

บางคำสั่งปลอดภัย บางคำสั่งมีผลข้างเคียง

ถ้าเห็นคำสั่งที่เกี่ยวกับ delete, remove, push, deploy, database หรือ secret ให้ระวังเป็นพิเศษ

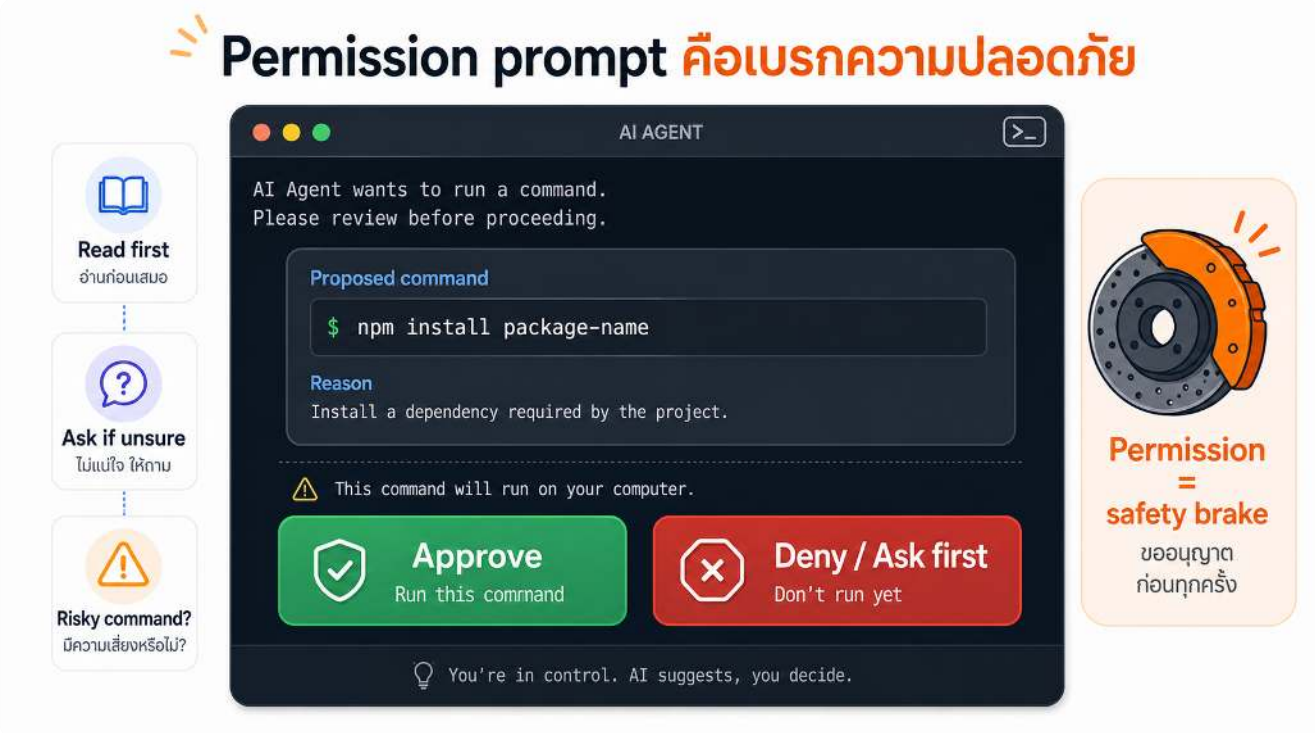
4. WebFetch / WebSearch

ใช้ค้นหรืออ่านข้อมูลจากเว็บ

ใช้ได้ดีเวลาต้องดู official docs หรือ error message

แต่สำหรับหนังสือเล่มนี้ ถ้าพูดเรื่อง fact เราจะยึด official sources เท่านั้น

Permission คือเบรก ไม่ใช่อุปสรรค



Permission prompt คือเบรกความปลอดภัย

permission prompt ไม่ใช่สิ่งกีดขวาง แต่เป็นจังหวะที่คุณควรหยุดอ่านก่อนอนุญาต

เวลา Claude Code จะทำบางอย่าง เช่น แก้ไฟล์หรือรันคำสั่ง มันอาจถามให้คุณ approve

มือใหม่บางคนมอง permission prompt เป็นสิ่งน่ารำคาญ

แต่ให้มองใหม่:

> permission prompt คือจังหวะที่คุณได้หยุดคิดก่อน AI เปลี่ยน project จริง

ถ้าเป็นงานแรก ๆ ให้ใช้ mode ที่ขออนุญาตบ่อยหน่อย

โดยเฉพาะถ้า project มีข้อมูลจริง หรือคุณยังไม่มั่นใจ

Mode ที่ควรรู้

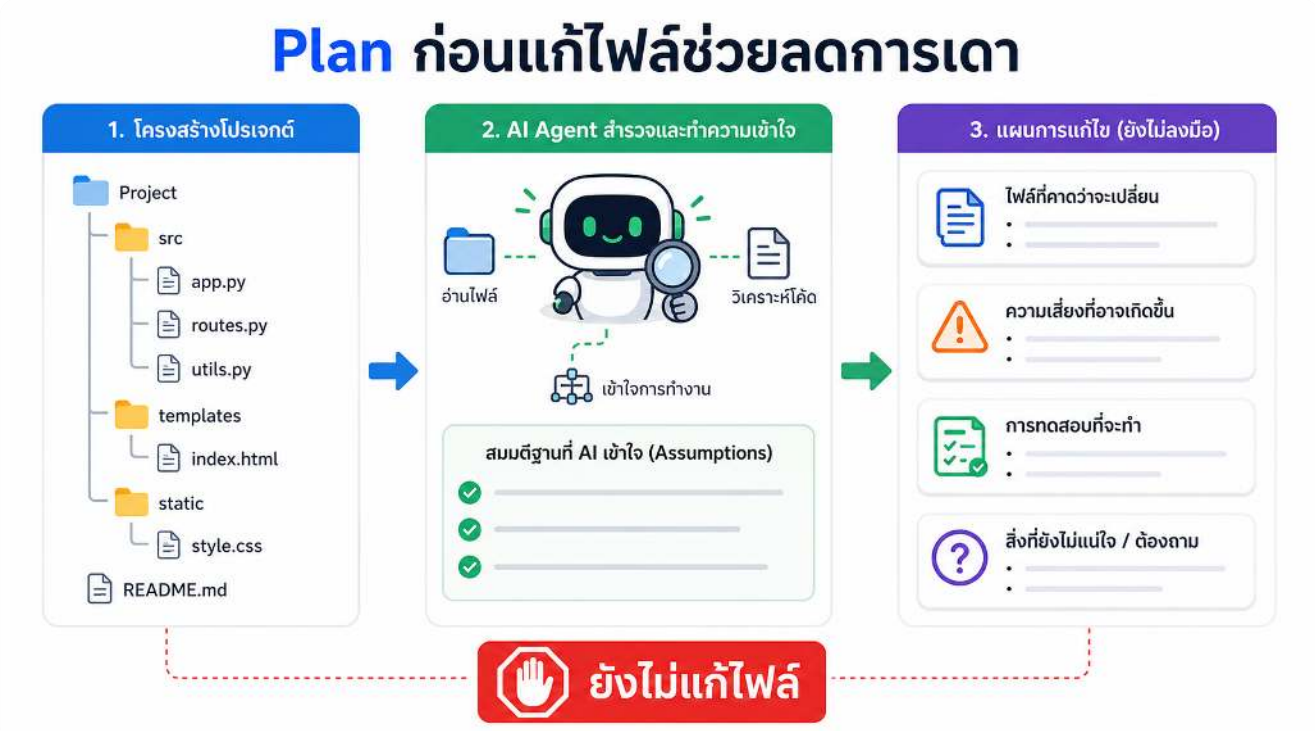
สำหรับ non-coder ให้รู้ 3 mode ก่อนพอ

Mode	ใช้เมื่อไหร่
Default	เริ่มต้น ใช้กับงานทั่วไปและงานที่ยังไม่มั่นใจ
Plan mode	ให้ Claude อ่านและวางแผนก่อนแก้ไฟล์
Accept edits	ใช้เมื่อคุณเริ่มไว้ใจ direction แล้ว และพร้อม review diff

ส่วน mode ที่ข้าม permission หรือปล่อยให้ทำเยอะมาก ๆ ไม่ควรใช้ใน project จริงตอนเริ่มต้น

ถ้าคุณไม่แน่ใจ ให้ใช้ default หรือ plan mode

Plan mode คือเพื่อนที่ดีที่สุดของ non-coder



Plan ก่อนแก้ไฟล์ช่วยลดการเดา

ก่อนให้ AI แก้ไฟล์ ให้มันสรุปไฟล์ที่จะเปลี่ยน ความเสี่ยง และวิธี test ก่อนเสมอ

ถ้าคุณไม่รู้ว่า AI จะไปแก้อะไร ให้ใช้ plan mode หรือสั่งให้มันวางแผนก่อน

ตัวอย่าง prompt:

> อย่าเพิ่งแก้ไฟล์
> ช่วยอ่าน `project` นี้ แล้วสรุปว่า `landing page` อยู่ไฟล์ไหน
> จากนั้นเสนอ `plan` ว่าถ้าจะเพิ่ม `waitlist form` ต้องแก้ไฟล์ไหนบ้าง
> ให้บอกความเสี่ยงและคำถามที่ต้องถามก่อนเริ่ม

ทำไมวิธีนี้ดี?

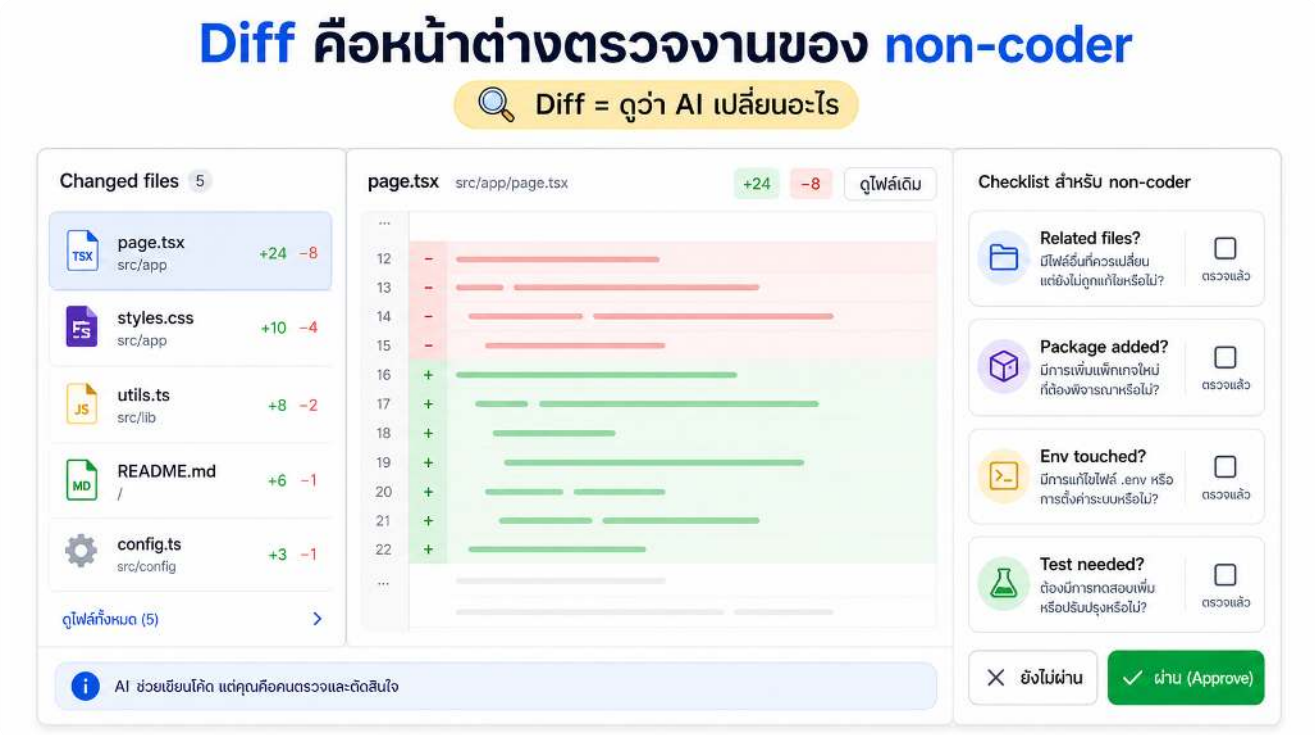
เพราะคุณได้เห็นก่อนว่า Claude Code เข้าใจ project ยังไง

ถ้ามันเข้าใจผิด คุณแก้ทิศทางได้ก่อนที่มันจะเปลี่ยนไฟล์

สำหรับ non-coder นี่สำคัญมาก

เพราะคุณอาจตรวจ code ลึก ๆ ไม่ได้ แต่คุณตรวจ plan ได้

Diff คืออะไร และทำไมต้องดู



Diff คือหน้าต่างตรวจงานของ non-coder

diff คือหลักฐานว่า AI เปลี่ยนอะไรไปบ้าง ก่อนคุณจะ commit หรือ deploy

Diff คือรายการว่าไฟล์เปลี่ยนอะไรบ้าง

คุณอาจอ่าน code ไม่ออกทั้งหมด แต่ยังดู diff แบบคร่าว ๆ ได้

ให้ถามตัวเอง:

> `Claude` แก้ไฟล์ที่เกี่ยวข้องจริงไหม
> มีไฟล์แปลก ๆ โผล่มาไหม
> มันลบอะไรเยอะเกินไปไหม
> มันไปแตะไฟล์ `config` หรือ `secret` ไหม
> มันเพิ่ม `package` ใหม่โดยไม่บอกไหม

ถ้าไม่เข้าใจ diff ให้ถาม Claude:

```
ช่วยอธิบาย diff นี้แบบ non-coder
แยกเป็น:
1. ไฟล์ไหนถูกแก้
2. แก้เพื่ออะไร
3. มีความเสี่ยงอะไร
4. ฉันควร test อะไรบ้างก่อน accept
```

นี่เป็นนิสัยสำคัญมาก

อย่า approve งานใหญ่โดยไม่ดูว่าเปลี่ยนอะไร

คำสั่งแรก ๆ ที่ควรใช้กับ Claude Code

เมื่อเปิด project ใหม่ อย่าเพิ่งสั่งให้ build ทันที

เริ่มจากให้มันทำความเข้าใจก่อน

Prompt 1: อ่าน project

```
ช่วยอ่าน project นี้แบบไม่แก้ไฟล์
สรุปให้ฉันเข้าใจแบบ non-coder:
1. project นี้น่าจะทำอะไร
2. ใช้ stack อะไร
3. ไฟล์สำคัญอยู่ตรงไหน
4. คำสั่งที่น่าจะใช้ run/test/build คืออะไร
5. มีอะไรที่ควรระวังก่อนแก้ไฟล์ไหม
```

Prompt 2: หา flow สำคัญ

```
ช่วยหา flow หลักของ app นี้แบบไม่แก้ไฟล์
เช่น หน้าแรก form submit database หรือ API ที่เกี่ยวข้อง
สรุปเป็นแผนภาพข้อความง่าย ๆ ให้ฉันเข้าใจ
```

Prompt 3: วางแผนก่อนแก้

```
ฉันอยากเพิ่ม/แก้สิ่งนี้:
[อธิบายสิ่งที่ต้องการ]

อย่าเพิ่งแก้ไฟล์
ช่วยเสนอ plan ก่อนว่า:
1. ต้องแก้ไฟล์ไหน
2. ต้องเพิ่มอะไร
3. มี risk อะไร
4. ต้อง test อะไร
5. มีคำถามอะไรที่ต้องถามฉันก่อนเริ่ม
```

ถ้าใช้ 3 prompt นี้เป็นนิสัย คุณจะลดโอกาสที่ AI จะเดาและแก้มั่วได้เยอะ

สรุป

Claude Code ไม่ใช่แค่ chatbot ที่เขียน code ให้ดู

มันเป็น AI coding agent ที่อ่าน project แก้ไฟล์ รันคำสั่ง และตรวจผลได้

สำหรับ non-coder สิ่งที่ต้องเข้าใจไม่ใช่ syntax แต่คือ workflow:

```
อ่านก่อน
วางแผนก่อน
ค่อยแก้
ดู diff
ทดสอบ behavior
ค่อย deploy
```

บทนี้อยากให้คุณจำ 5 ข้อ:

1. Claude Code ทำงานกับไฟล์จริง ไม่ใช่แค่ตอบข้อความ
2. Permission prompt คือเบรกที่ช่วยให้คุณไม่พลาดเร็วเกินไป
3. Plan mode เหมาะมากสำหรับ non-coder
4. Diff ต้องถูกอธิบายก่อน accept งานใหญ่
5. ถ้าไม่เข้าใจ command หรือ change ให้ถามก่อน approve

บทต่อไป เราจะเตรียมเครื่องและ project แรกให้พร้อมใช้ Claude Code โดยไม่จมกับ technical detail เกินจำเป็น

$ บทที่ 3

เตรียมเครื่องและเริ่มโปรเจกต์แรก

แก่นของบทนี้

ก่อนให้ Claude Code สร้างอะไร คุณต้องเตรียม “พื้นที่ทำงาน” ให้พร้อมก่อน

ไม่ใช่เพื่อให้คุณกลายเป็น programmer

แต่เพื่อให้คุณมีที่ปลอดภัยพอสำหรับให้ AI อ่านไฟล์ แก้ไฟล์ รันคำสั่ง และย้อนกลับได้ถ้าพัง

บทนี้จะทำให้คุณมี 4 อย่าง:

> 1. เครื่องพร้อมใช้ Claude Code
> 2. โฟลเดอร์ project ที่เป็นระเบียบ
> 3. Git สำหรับย้อนกลับ
> 4. prompt แรกสำหรับให้ Claude Code อ่าน project โดยไม่แก้ไฟล์

ถ้าทำ 4 อย่างนี้ได้ คุณพร้อมเริ่มบทถัดไป

ถ้าคุณต้องการทำตามแบบทีละขั้นจริง ๆ ไม่ต้องเดา command เอง ให้เปิด **Appendix H — Setup Playbook** คู่กับบทนี้ บทนี้อธิบายภาพรวม ส่วน Appendix H คือ checklist ลงมือทำ

สิ่งที่ต้องมีก่อนเริ่ม

เส้นทาง setup ขั้นต่ำก่อนเริ่ม project

ภาพนี้คือ setup path แบบย่อ: terminal, runtime, Claude Code, Git, project folder และ safety check

คุณไม่ต้อง setup ระบบซับซ้อน

แต่ควรมีสิ่งเหล่านี้:

- เครื่อง macOS, Windows หรือ Linux ที่ใช้งานประจำ
- internet connection
- terminal หรือ command prompt
- account ที่ใช้ Claude Code ได้
- Git/GitHub พื้นฐาน
- project folder หนึ่งอันสำหรับทดลอง

Claude Code official docs ระบุว่ามันใช้ได้หลาย interface เช่น terminal, web, desktop app, VS Code และ JetBrains

แต่ในเล่มนี้เราจะใช้แนวคิดหลักจาก terminal เพราะเข้าใจ workflow ได้ชัดที่สุด

ถ้าคุณใช้ desktop app หรือ VS Code หลายอย่างจะคล้ายกัน แค่หน้าตาเปลี่ยน

Terminal ไม่ได้น่ากลัวอย่างที่คิด

สำหรับเล่มนี้ terminal คือช่องสั่งงานเครื่องด้วยข้อความ

คุณจะเจอคำสั่งประมาณนี้:

```bash
cd my-project
claude
git status
npm run dev
```

แปลแบบคนทั่วไป:

คำสั่ง	ความหมาย
`cd my-project`	เข้าโฟลเดอร์ project
`claude`	เปิด Claude Code ในโฟลเดอร์นี้
`git status`	ดูว่าไฟล์ไหนเปลี่ยน
`npm run dev`	เปิดเว็บในเครื่องเพื่อทดลอง

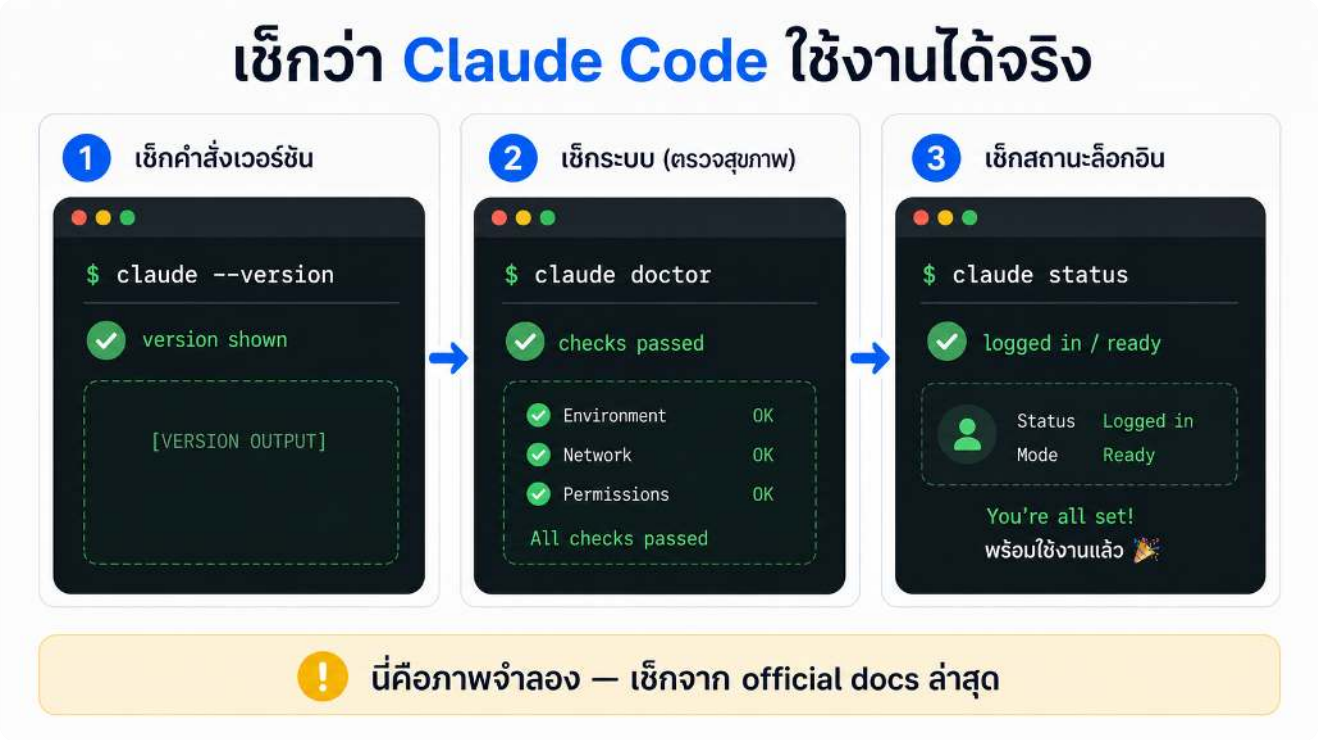
คุณไม่ต้องจำทุกคำสั่งตั้งแต่แรก

ถ้าไม่เข้าใจคำสั่ง ให้ถาม Claude Code ก่อน approve:

> คำสั่งนี้ทำอะไร อธิบายแบบ non-coder
> มีความเสี่ยงไหม
> ถ้าฉันไม่รันตอนนี้จะเกิดอะไรขึ้น

นิสัยนี้สำคัญกว่าการจำ command

ติดตั้ง Claude Code แบบไม่หลงรายละเอียด



เช็กว่า Claude Code ใช้งานได้จริง

หลังติดตั้ง อย่าดูแค่ว่า install ผ่าน ให้ verify command, diagnostic และ login ให้เรียบร้อย

Official docs ของ Claude Code มีหลายวิธีติดตั้งตามระบบปฏิบัติการ

ในภาพรวมคือ:

1. เปิด terminal หรือ command prompt
2. ใช้คำสั่งติดตั้งตามระบบของคุณ
3. เปิด project folder
4. พิมพ์ `claude`
5. login เมื่อระบบถาม

ตัวอย่างจาก official quickstart คือเปิด project แล้วรัน:

```bash
claude
```

ครั้งแรกระบบจะให้ login

หลังจาก login สำเร็จ คุณจะเริ่ม session กับ Claude Code ได้

ถ้าคุณติดตั้งแล้วพิมพ์ `claude` ไม่เจอ ให้มองหา error ก่อน ไม่ต้อง panic

error ที่เจอบ่อยคือเครื่องหา command ไม่เจอ เพราะ PATH ยังไม่ถูก หรือเปิด terminal เก่าที่ยังไม่ refresh

ให้ถาม Claude หรือดู troubleshooting ด้วยข้อความประมาณนี้:

> ฉันติดตั้ง Claude Code แล้ว แต่พิมพ์ claude แล้วขึ้น command not found
> ช่วยอธิบายสาเหตุแบบ non-coder และบอกขั้นตอนตรวจทีละข้อ

Project folder คือพื้นที่ทดลอง

Project แรกต้องเป็นพื้นที่ทดลองที่พังได้

ก่อนแตะ project จริง ให้แยก playground ออกจากงานธุรกิจและข้อมูลลูกค้าจริง

อย่าเริ่มจาก project สำคัญของธุรกิจทันที

ให้เริ่มจากโฟลเดอร์ทดลองก่อน

ตัวอย่าง:

```
vibe-coding-practice/
```

ข้างในอาจเป็น project ใหม่ หรือ project ตัวอย่างที่คุณตั้งใจใช้ฝึก

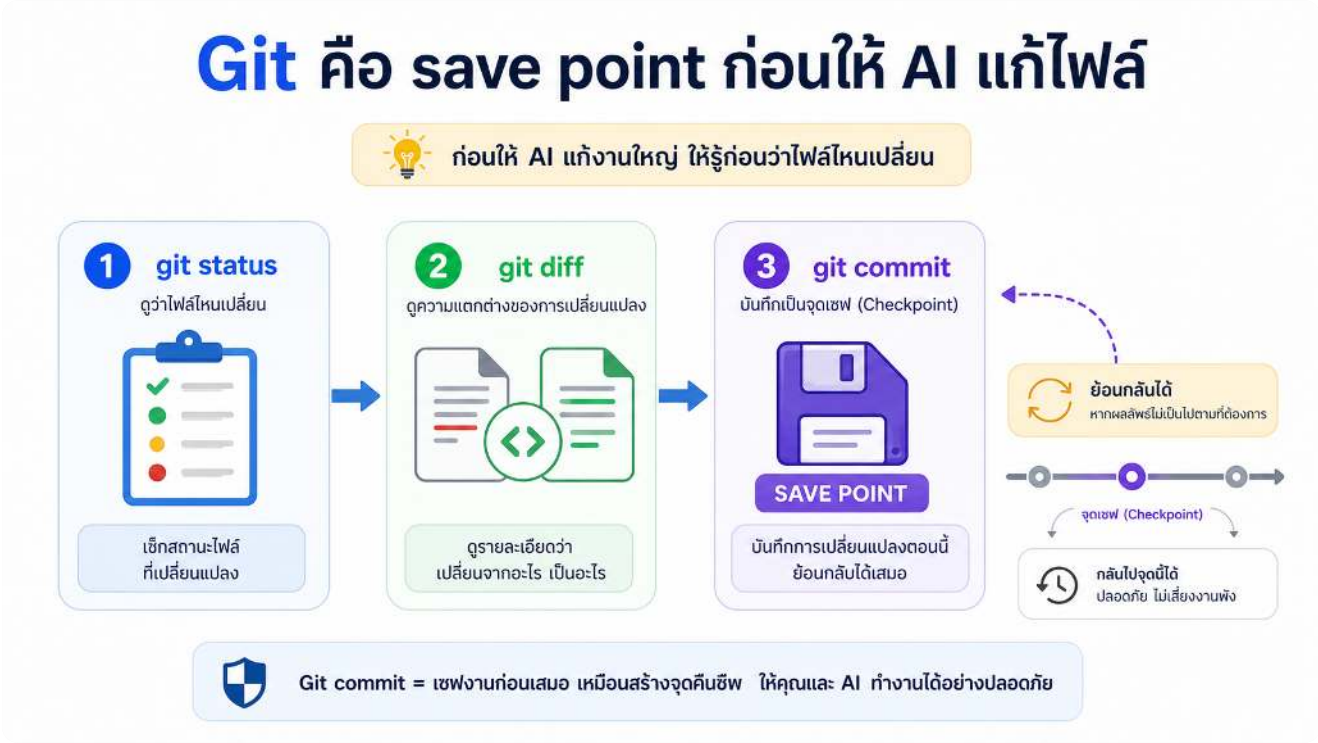
หลักการคือ:

> project แรกควรพังได้โดยไม่ทำให้ธุรกิจเสียหาย

ถ้าคุณมีเว็บบริษัทจริงอยู่แล้ว อย่าเพิ่งให้ AI แก้เว็บจริงตั้งแต่วันแรก

ให้ copy มาทดลอง หรือเริ่มจาก landing page ใหม่ที่ไม่กระทบระบบหลัก

Git คือปุ่ม save ที่ย้อนกลับได้



Git คือ save point ก่อนให้ AI แก้ไฟล์

Git ช่วยให้คุณเห็นไฟล์ที่เปลี่ยน ดู diff และบันทึก checkpoint ก่อนงานใหญ่

สำหรับ non-coder ให้คิดว่า Git คือระบบบันทึก version ของ project

ไม่ต้องเข้าใจทุกคำสั่งลึก ๆ ตั้งแต่แรก

ให้จำแค่นี้:

```
git status = ตอนนี้ไฟล์ไหนเปลี่ยน
git diff = เปลี่ยนอะไรบ้าง
git commit = บันทึก checkpoint
git log = ดูประวัติ checkpoint
```

ทำไม Git สำคัญ?

เพราะ Claude Code แก้ไฟล์ได้จริง

ถ้าไม่มี Git คุณอาจไม่รู้ว่า AI เปลี่ยนอะไรไปบ้าง

ถ้ามี Git คุณจะดูได้ว่า:

- ไฟล์ไหนถูกแก้
- แก้อะไร
- ก่อนแก้เป็นยังไง
- จะย้อนกลับได้ไหม

ก่อนให้ Claude Code ทำงานใหญ่ ให้เริ่มจากสถานะที่สะอาด

```bash
git status
```

ถ้ามีไฟล์เปลี่ยนอยู่แล้ว ให้รู้ก่อนว่าเป็นของใครและสำคัญไหม

อย่าให้ AI แก้ทับงานที่คุณยังไม่ได้บันทึกโดยไม่ตั้งใจ

GitHub จำเป็นไหม

Git อยู่ในเครื่องคุณ

GitHub คือที่เก็บ repo บน cloud เพื่อ backup, share, deploy หรือทำงานร่วมกับคนอื่น

สำหรับ project ฝึก คุณอาจยังไม่ต้องใช้ GitHub ทันที

แต่ถ้าจะ deploy กับ Vercel หรืออยาก backup project ไว้ GitHub จะมีประโยชน์มาก

กฎง่าย ๆ:

- project ฝึกส่วนตัว: ใช้ Git local ก่อนก็ได้

- project ที่จะ deploy: ใช้ GitHub จะสะดวก
- project ที่มีข้อมูลลับ: อย่า push ขึ้น public repo โดยไม่ตรวจ

ก่อน push ขึ้น GitHub ให้เช็กว่าไม่มีไฟล์ลับ เช่น:

```
.env
API key
token
password
service role key
```

บท security จะลงรายละเอียด แต่จำไว้ก่อนว่า:

> GitHub public repo ไม่ใช่ที่เก็บความลับ

เริ่ม session แรกอย่างปลอดภัย

เมื่อเข้า project folder แล้วเปิด Claude Code:

```bash
claude
```

อย่าเริ่มด้วยคำสั่งให้แก้ไฟล์ทันที

เริ่มด้วยการให้มันอ่านและอธิบาย project ก่อน

ใช้ prompt นี้:

```
ช่วยอ่าน project นี้แบบไม่แก้ไฟล์
สรุปให้ฉันเข้าใจแบบ non-coder:
1. project นี้น่าจะทำอะไร
2. ใช้ stack หรือเครื่องมืออะไร
3. ไฟล์สำคัญอยู่ตรงไหน
4. คำสั่งที่น่าจะใช้ run/test/build คืออะไร
5. มีอะไรที่ควรระวังก่อนแก้ไฟล์ไหม
```

คำว่า “แบบไม่แก้ไฟล์” สำคัญมาก

เพราะ session แรกควรเป็นการทำความเข้าใจ ไม่ใช่การลงมือทันที

ถ้า Claude Code เสนอจะรันคำสั่งหรือแก้ไฟล์ ให้คุณหยุดและถามก่อน:

```
ตอนนี้ฉันต้องการให้วิเคราะห์อย่างเดียว
อย่าเพิ่งแก้ไฟล์หรือรันคำสั่งที่เปลี่ยน project
```

คำสั่งตรวจความพร้อม

หลังจาก Claude Code อ่าน project แล้ว ให้ถามต่อ:

```
ช่วยทำ readiness checklist สำหรับ project นี้ก่อนเริ่มแก้ไฟล์
แยกเป็น:
1. สิ่งที่พร้อมแล้ว
2. สิ่งที่ยังไม่รู้
3. สิ่งที่ต้องระวัง
4. คำสั่งที่ควรใช้ run/test/build
5. ไฟล์ที่ไม่ควรแตะโดยไม่ถามก่อน
```

output นี้จะกลายเป็นแผนที่ของ project

คุณไม่ต้องเข้าใจทุกไฟล์ทันที

แต่คุณควรรู้ว่าอะไรคือพื้นที่ปลอดภัย และอะไรคือพื้นที่ที่ต้องถามก่อน

ถ้าไม่มี project เดิม ให้เริ่มจาก project ง่าย ๆ

ถ้าคุณยังไม่มี project ให้เริ่มจากสิ่งที่เล็กมาก

เช่น:

```
landing page 1 หน้า สำหรับบริการสมมติ
ไม่มี login
ไม่มี payment
ไม่มี database
มีแค่ headline, benefit, CTA และ contact form mockup
```

อย่าเริ่มจาก app ใหญ่

project ฝึกที่ดีควรมีคุณสมบัติ 4 อย่าง:

- ลบได้โดยไม่เสียดาย
- ไม่มีข้อมูลลูกค้าจริง
- ไม่มีเงินเกี่ยวข้อง
- ใช้ทดสอบ workflow ได้ครบ

เป้าหมายของ project แรกไม่ใช่สร้างธุรกิจทันที

เป้าหมายคือทำให้คุณคุ้นกับการสั่ง อ่าน plan ดู diff test และแก้จาก error

---

อย่าให้ AI install ทุกอย่างโดยไม่ถาม

เวลาเริ่ม project ใหม่ Claude Code อาจเสนอให้ติดตั้ง package หรือใช้ framework บางตัว

บางครั้งเหมาะสม

แต่สำหรับ non-coder อย่า approve การติดตั้งทุกอย่างแบบไม่อ่าน

ก่อนติดตั้ง package ใหม่ ให้ถาม:

```
ทำไมต้องติดตั้ง package นี้
ใช้ทำอะไร
มีทางเลือกที่ไม่ต้องติดตั้งเพิ่มไหม
package นี้จำเป็นกับ version แรกจริงไหม
ถ้าไม่ติดตั้ง ตอนนี้ยังทำงานต่อได้ไหม
```

version แรกควรใช้ของน้อยที่สุดเท่าที่พอทำงานได้

ยิ่ง dependency เยอะ คุณยิ่งต้องดูแลเยอะ

Folder ที่ควรระวัง

ในหลาย project จะมีไฟล์หรือโฟลเดอร์ที่ควรระวังเป็นพิเศษ

เช่น:

```
.env
.env.local
node_modules/
.git/
package-lock.json
supabase/
.vercel/
```

คุณไม่ต้องรู้ลึกทุกไฟล์

แต่ควรรู้ว่า:

- `.env` มักเกี่ยวกับ secret/config
- `.git` คือประวัติ version control
- `node_modules` คือ packages จำนวนมาก ไม่ควรแก้เอง
- lock file เกี่ยวกับ version ของ dependency

ถ้า Claude Code จะไปแตะไฟล์กลุ่มนี้ ให้ถามเหตุผลก่อน

```
ทำไมต้องแก้ไฟล์นี้
มีความเสี่ยงอะไร
ถ้าแก้ผิดจะกระทบอะไร
มีวิธีที่ปลอดภัยกว่านี้ไหม
```

Checklist: เครื่องและ project พร้อมหรือยัง

ก่อนจบบทนี้ ให้เช็ก 12 ข้อนี้

- ☐ เปิด terminal หรือ command prompt ได้
- ☐ ติดตั้ง Claude Code แล้ว
- ☐ เช็กแล้วว่า account/access ของคุณใช้ Claude Code ได้ตาม official docs ล่าสุด

- ☐ พิมพ์ `claude` แล้วเริ่ม session ได้
- ☐ login สำเร็จ
- ☐ มี project folder สำหรับฝึก
- ☐ project แรกไม่มีข้อมูลลูกค้าจริง
- ☐ project แรกไม่เกี่ยวกับเงิน
- ☐ รู้วิธีรัน `git status`
- ☐ รู้ว่าไฟล์ไหนเปลี่ยนก่อนให้ AI แก้ต่อ
- ☐ ให้ Claude Code อ่าน project แบบไม่แก้ไฟล์แล้ว
- ☐ ได้รายการไฟล์สำคัญและคำสั่ง run/test/build แล้ว
- ☐ รู้ว่าไฟล์ไหนควรระวัง เช่น `.env`

ถ้ายังไม่ครบ ไม่เป็นไร

แต่ให้แก้ให้ครบก่อนเริ่ม project จริง

Prompt รวมท้ายบท

ใช้ prompt นี้หลังเปิด Claude Code ใน project folder

```
ฉันเพิ่งเปิด Claude Code ใน project นี้
ฉันไม่ใช่ programmer และยังไม่ต้องการให้คุณแก้ไฟล์

ช่วยทำ project onboarding ให้ฉันแบบ non-coder:
1. project นี้น่าจะทำอะไร
2. โครงสร้าง folder หลัก ๆ คืออะไร
3. ไฟล์ไหนสำคัญที่สุด
4. command สำหรับ run/test/build น่าจะมีอะไรบ้าง
5. มีไฟล์ secret/config ที่ควรระวังไหม
6. ถ้าจะเริ่มแก้งานเล็ก ๆ ควรเริ่มตรงไหน
7. สิ่งที่ไม่ควรแตะโดยไม่ถามฉันก่อนคืออะไร

กติกา:
- ห้ามแก้ไฟล์
- ห้ามติดตั้ง package
- ห้ามรันคำสั่งที่เปลี่ยน project
- ถ้าต้องการรัน command เพื่ออ่านข้อมูล ให้บอกก่อนว่าคำสั่งนั้นทำอะไร
```

ถ้า Claude ตอบมาแล้วยัง technical เกินไป ให้ถามต่อ:

> ช่วยอธิบายใหม่แบบคนไม่เคยเขียนโค้ดมาก่อน และสรุปเป็น `checklist` ที่ฉันทำตามได้

สรุป

บทนี้ไม่ได้ต้องการให้คุณ setup ทุกอย่างแบบ developer มืออาชีพ

ต้องการแค่ให้คุณมีพื้นที่ทำงานที่ปลอดภัยพอสำหรับเริ่มใช้ Claude Code

จำ 5 ข้อนี้:

1. เริ่มจาก project ที่พังได้
2. ใช้ Git เป็น checkpoint
3. เปิด Claude Code ใน folder ที่ถูกต้อง
4. ให้ AI อ่านและสรุปก่อนแก้ไฟล์
5. อย่า approve command/install/change ที่ไม่เข้าใจ

บทต่อไป เราจะทำให้ project เข้าใจง่ายขึ้นสำหรับ Claude Code ด้วยไฟล์ context เช่น `CLAUDE.md` เพื่อให้ AI ไม่ต้องเดา requirement สำคัญทุกครั้งที่เริ่ม session ใหม่

$ บทที่ 4

Project Context: เขียน `CLAUDE.md` ให้ AI ไม่ต้องเดา

แก่นของบทนี้

Claude Code เก่งขึ้นมากเมื่อมันเข้าใจ context ของ project

แต่ถ้าคุณต้องอธิบายเรื่องเดิมทุก session คุณจะเสียเวลา และ AI ก็มีโอกาสลืมหรือเข้าใจไม่ตรงกัน

วิธีแก้คือเขียน context สำคัญไว้ในไฟล์ `CLAUDE.md`

ให้คิดว่า `CLAUDE.md` คือ briefing note สำหรับ Claude Code

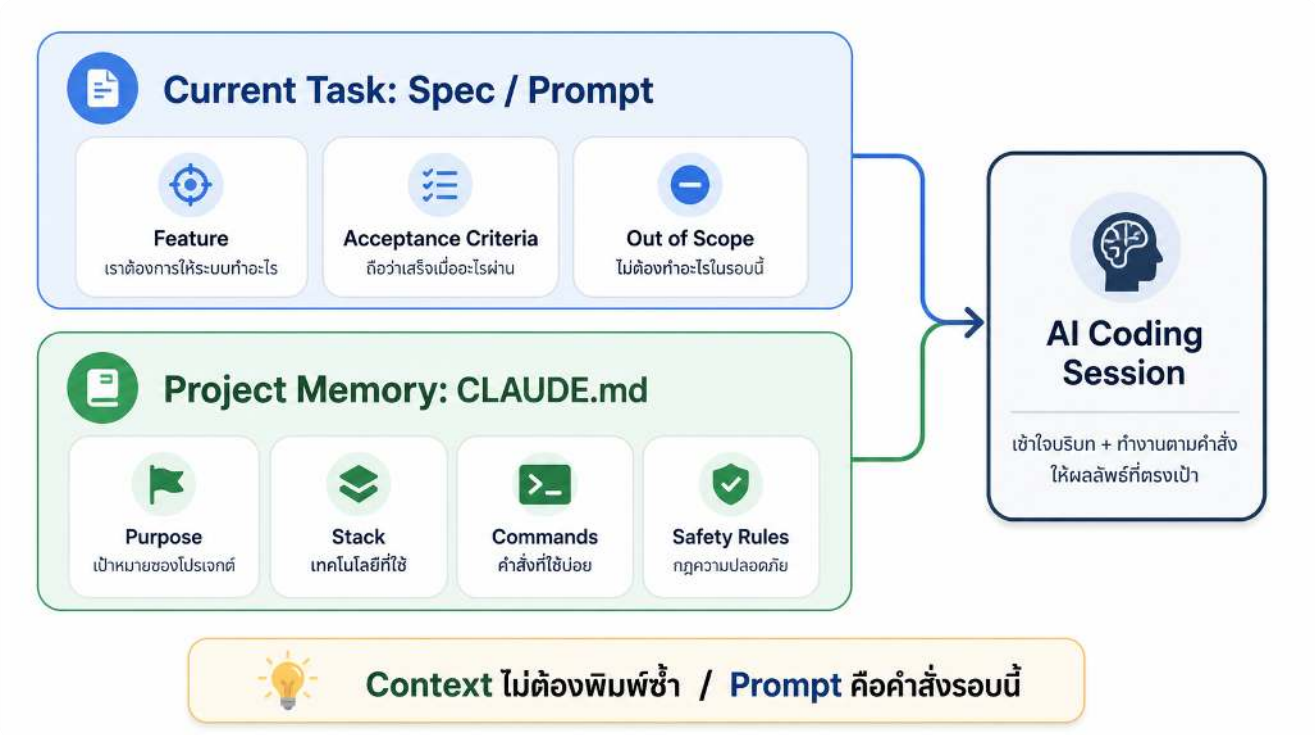
ไม่ใช่เอกสารยาว ๆ เพื่ออวดความละเอียด

แต่เป็นไฟล์สั้น ๆ ที่ตอบว่า:

> project นี้คืออะไร
> ใครใช้
> ใช้ stack อะไร
> รัน/test/build ยังไง
> กติกาการทำงานคืออะไร
> อะไรห้ามแตะโดยไม่ถาม
> เสร็จแล้วต้องตรวจอะไร

ถ้า prompt คือคำสั่งรายครั้ง
`CLAUDE.md` คือความจำระยะยาวของ project

ทำไม context สำคัญกว่า prompt ยาว ๆ ครั้งเดียว



Context ระยะยาวกับ prompt รายครั้งทำงานคู่กัน

`CLAUDE.md` *คือ memory ของ project ส่วน spec/prompt คือคำสั่งของงานรอบนี้*

มือใหม่มักพยายามเขียน prompt ยาวมากในทุก session

เช่น:

> นี่คือ project ของฉัน ใช้ Next.js มี Supabase กลุ่มลูกค้าเป็น SME ห้ามแตะ payment ตอนนี้ โทนภาษาไทยตรง ๆ ฟอร์มต้อง validate email...

ทำแบบนี้ได้ แต่เหนื่อย

และถ้าลืมบอกบางอย่าง AI ก็อาจเดา

`CLAUDE.md` ช่วยให้กติกาหลักถูกโหลดทุกครั้งที่เริ่ม session

ตัวอย่างสิ่งที่ควรอยู่ใน `CLAUDE.md`:

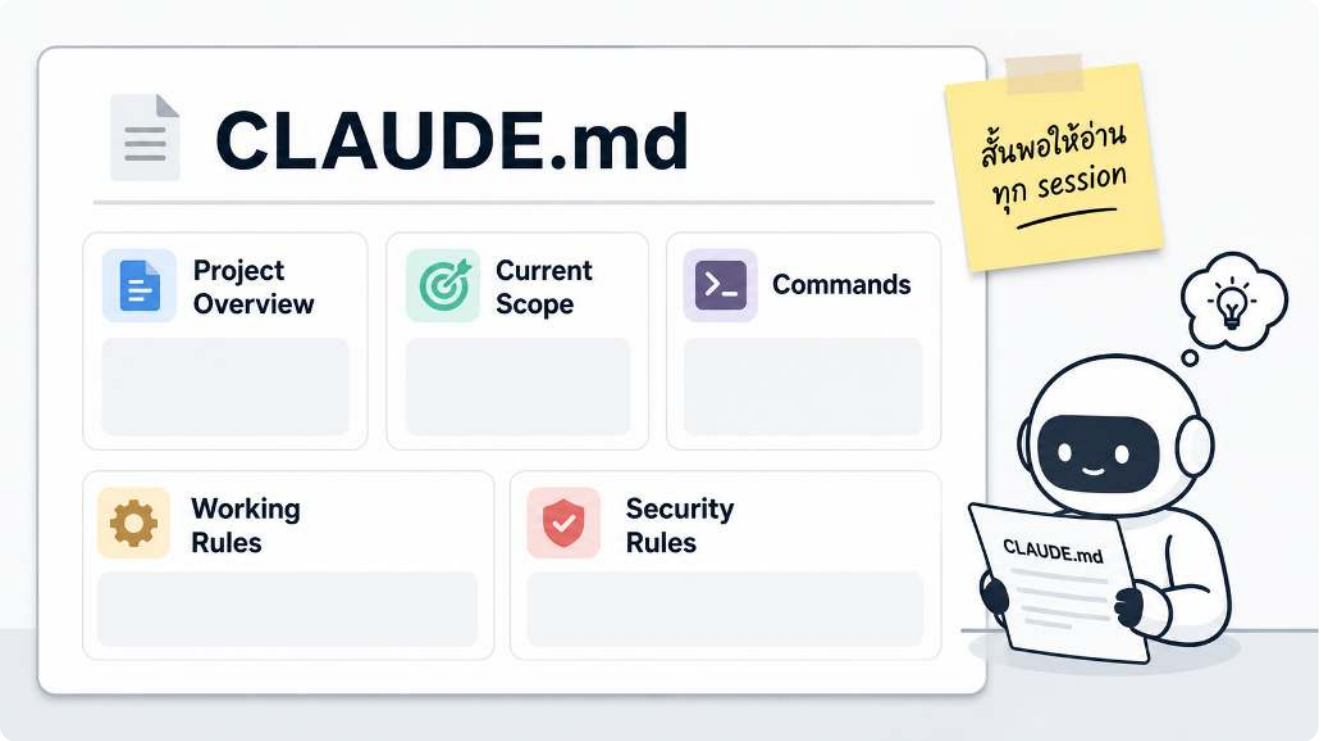
- project purpose
- target user
- tech stack
- run/test/build commands

- coding/style preference ที่จำเป็น
- security rules
- workflow rules
- definition of done

ไม่ต้องเขียนทุกอย่างในโลก

เขียนเฉพาะสิ่งที่ถ้าไม่บอกแล้ว Claude อาจทำผิด

`CLAUDE.md` ไม่ใช่เวทมนตร์



คือ briefing note ของ project

`CLAUDE.md` *ควรเป็น briefing note สั้น ๆ ที่ Claude อ่านได้ทุก session ไม่ใช่เอกสารยาวเพื่ออวดละเอียด*

ต้องเข้าใจข้อจำกัดก่อน

`CLAUDE.md` เป็น context ที่ Claude อ่าน ไม่ใช่กำแพงเหล็ก

ถ้าคุณเขียนว่า:

> ห้ามลบไฟล์สำคัญ

มันช่วยเตือน Claude แต่ไม่ได้ enforce แบบ permission rule

ถ้าต้องการบล็อกจริง ต้องใช้ permissions/settings ซึ่งจะเป็นเรื่องของบท security/advanced ต่อไป

สำหรับ non-coder ให้ใช้ `CLAUDE.md` เป็น 3 อย่างก่อน:

1. บอกภาพรวม project
2. ลดการเดา
3. ทำให้ Claude ตรวจงานตาม checklist เดิมทุกครั้ง

`CLAUDE.md` ควรสั้น

Official docs แนะนำให้เขียน instruction แบบเฉพาะเจาะจง กระชับ และมีโครงสร้างชัด

สำหรับเล่มนี้ ผมแนะนำกฎง่าย ๆ:

> `CLAUDE.md` เวอร์ชันแรกควรไม่เกิน 1–2 หน้า

ถ้ายาวกว่านั้นมาก แปลว่าคุณอาจใส่สิ่งที่ยังไม่จำเป็น

อย่าใส่:

- tutorial ยาว ๆ
- รายละเอียดทุกไฟล์
- ประวัติบริษัทเต็มหน้า
- requirement ที่เปลี่ยนทุกวัน
- secret/API key/password
- สิ่งที่ Claude อ่านจาก code เองได้

ให้ใส่:

- สิ่งที่ Claude เดาไม่ได้
- สิ่งที่ถ้าลืมแล้วเสี่ยง
- command ที่ต้องใช้จริง

- rule ที่ต้องทำซ้ำทุกงาน
- checklist ก่อนจบงาน

Template `CLAUDE.md` สำหรับ non-coder

ใช้ template นี้เป็นจุดเริ่มต้น

MARKDOWN

```
# Project Context

## What this project is
- This project is: [อธิบาย project ใน 1-3 ประโยค]
- Target users: [ใครใช้]
- Main goal: [เป้าหมายหลักของ version แรก]

## Current scope
- Version 1 includes:
  - [feature 1]
  - [feature 2]
  - [feature 3]
- Out of scope for now:
  - Login
  - Payment
  - Admin dashboard
  - Anything not explicitly requested

## Tech stack
- Framework: [เช่น Next.js / React / ยังไม่รู้]
- Database: [เช่น Supabase / none]
- Deployment: [เช่น Vercel / not decided]

## Common commands
- Run locally: `[command]`
- Build: `[command]`
- Test: `[command or not set up yet]`

## Working rules
- Before making multi-file changes, explain the plan first.
- Prefer small, reversible changes.
- Do not add new packages unless you explain why they are necessary.
- Do not change authentication, payment, database permissions, or environment files w
- After changes, explain what changed and what I should test.

## Security rules
- Never hard-code API keys, tokens, passwords, or secrets.
- Do not expose server-only secrets to the browser.
- If a change touches user data, database rules, login, payment, or deployment config

## Definition of done
A task is not done until:
- The app still runs or builds.
- The main user flow has been tested.
- Any changed files are summarized.
- Risks and manual test steps are listed.
```

คุณไม่ต้องกรอกให้สมบูรณ์ 100% ตั้งแต่วันแรก

เริ่มจากสิ่งที่รู้ แล้วค่อยให้ Claude ช่วยเติมจาก project จริง

วิธีให้ Claude ช่วยสร้าง `CLAUDE.md`

ถ้า project มีไฟล์อยู่แล้ว ให้ใช้ prompt นี้

```
ช่วยสร้างร่าง CLAUDE.md สำหรับ project นี้
ฉันไม่ใช่ programmer ดังนั้นอยากให้เขียนแบบอ่านง่ายและใช้ได้จริง

กติกา:
- อ่าน project ก่อน แต่อย่าเพิ่งแก้ไฟล์
- สรุปเฉพาะสิ่งที่ Claude ควรรู้ทุก session
- อย่าใส่ข้อมูลที่เดาไม่ได้ ให้ใส่ TODO แทน
- ห้ามใส่ secret หรือค่าใน .env
- ให้ร่างเป็น markdown ไม่เกิน 120 บรรทัด

หัวข้อที่ต้องมี:
1. What this project is
2. Current scope
3. Tech stack
4. Common commands
5. Working rules
6. Security rules
7. Definition of done
```

เมื่อได้ร่างแล้ว อย่าเพิ่ง commit ทันที

ให้อ่านด้วยสายตาคนธรรมดาก่อน

ถามตัวเอง:

```
อ่านแล้วเข้าใจไหมว่า project นี้คืออะไร
มีอะไรเกินจริงไหม
มี rule ที่เข้มเกินไปไหม
มี secret หลุดไหม
มีคำสั่งที่ยังไม่แน่ใจไหม
```

ถ้าไม่แน่ใจ ให้ใส่ `TODO` ดีกว่าแต่งข้อมูลขึ้นมา

ตัวอย่าง `CLAUDE.md` สำหรับ landing page

MARKDOWN

```
# Project Context

## What this project is
- This is a simple landing page for an AI Workflow Audit service.
- Target users are Thai SME owners who want to use AI to reduce repetitive work.
- The main goal is to get visitors to submit a consultation request.

## Current scope
- Version 1 includes:
  - Landing page
  - Contact/waitlist form UI
  - Thank you message after submit
- Out of scope for now:
  - Login
  - Payment
  - Admin dashboard
  - CRM integration

## Tech stack
- Framework: TODO
- Database: none for now
- Deployment: Vercel later

## Working rules
- Keep copy in Thai, direct and non-hype.
- Before making multi-file changes, explain the plan first.
- Prefer simple HTML/CSS/components over adding new packages.
- Do not add login, payment, or database unless explicitly requested.

## Security rules
- Do not hard-code secrets.
- Do not add tracking scripts or external services without asking.

## Definition of done
- Page works on desktop and mobile.
- CTA button is visible.
- Form has basic validation.
- Summarize changed files and manual test steps.
```

สังเกตว่าไฟล์นี้ไม่ได้ยาว

แต่ช่วยลดการเดาได้มาก

Claude จะรู้ว่าไม่ควรเพิ่ม login, payment หรือ CRM integration เองโดยไม่มีคำสั่ง

ตัวอย่าง `CLAUDE.md` สำหรับ waitlist app

MARKDOWN

```
# Project Context

## What this project is
- This is a waitlist app for a future SaaS product.
- Users submit name, email, company, and their main problem.
- The goal is to validate demand before building the full product.

## Current scope
- Version 1 includes:
  - Landing page
  - Waitlist form
  - Store submissions in Supabase
  - Success and error states
- Out of scope for now:
  - Login
  - Payment
  - Admin dashboard
  - Email automation

## Tech stack
- Frontend: TODO
- Database: Supabase
- Deployment: Vercel

## Working rules
- Ask before changing database schema.
- Ask before changing Supabase policies or environment variables.
- Do not add new packages unless necessary.
- Explain changes in non-coder language.

## Security rules
- Never expose Supabase service role keys to the browser.
- Do not commit `.env` or `.env.local`.
- If unsure whether a key is public or secret, stop and ask.

## Definition of done
- Form validates required fields.
- Successful submit shows a thank you message.
- Failed submit shows a useful error.
- Manual test steps are listed.
- Any security assumptions are listed.
```

แบบนี้พอสำหรับ version แรก

อย่าเพิ่งใส่รายละเอียด Supabase ทั้งหมดในโลกลงไป

เขียนเท่าที่ช่วยให้ Claude ไม่ทำพลาดในงานปัจจุบัน

ควรเก็บไว้ที่ไหน

สำหรับเล่มนี้ ให้เริ่มง่าย ๆ:

```
your-project/
  CLAUDE.md
```

หรือถ้าอยากเก็บให้ project root สะอาด:

```
your-project/
  .claude/
    CLAUDE.md
```

ทั้งสองแบบใช้ได้

สำหรับ non-coder ผมแนะนำเริ่มจาก `CLAUDE.md` ที่ root ก่อน เพราะเห็นง่ายและแก้ง่าย

ถ้าวันหนึ่ง project ใหญ่ขึ้น ค่อยแยก `.claude/rules/` หรือ settings เพิ่ม

ตอนนี้ยังไม่ต้องรีบทำให้ซับซ้อน

อะไรไม่ควรใส่ใน `CLAUDE.md`

ห้ามใส่ของพวกนี้:

```
API key
password
token
service role key
private customer data
ข้อมูลลูกค้าจริง
ข้อมูลการเงิน
```

ถ้าคุณอยากบอก Claude ว่ามี env var ชื่ออะไร ให้ใส่แค่ชื่อ placeholder เช่น:

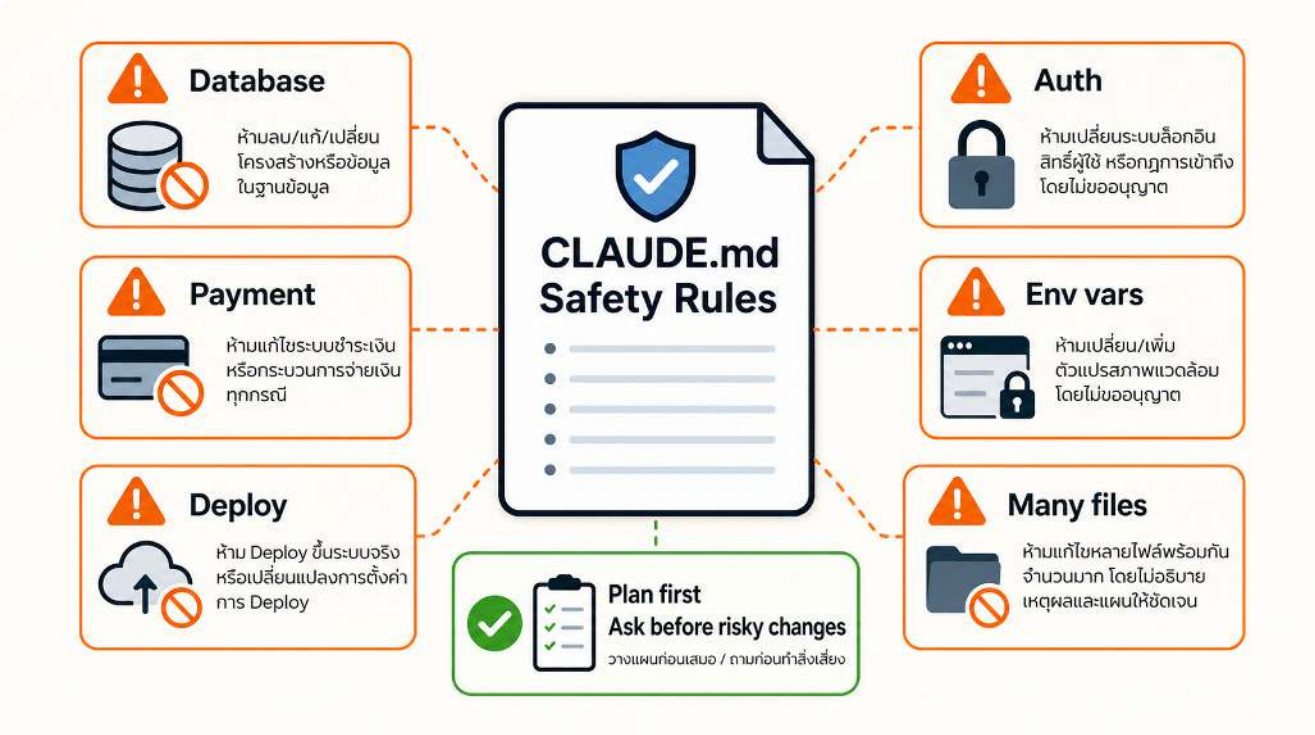
```
MARKDOWN

Environment variables used:
- NEXT_PUBLIC_SUPABASE_URL
- NEXT_PUBLIC_SUPABASE_ANON_KEY
- SUPABASE_SERVICE_ROLE_KEY (server-side only, never expose to browser)
```

อย่าใส่ค่าจริง

จำไว้ว่า:

> context file ไม่ใช่ตู้เซฟ

ใช้ `CLAUDE.md` เพื่อบังคับให้ AI ช้าลง



Safety rules ที่ควรอยู่ใน

ใส่ rule ให้ Claude ถามก่อนแตะส่วนเสี่ยง เช่น database, auth, payment, env, deploy หรือหลายไฟล์

สำหรับ non-coder rule ที่มีประโยชน์ที่สุดไม่ใช่ rule เรื่อง style

แต่คือ rule ที่บังคับให้ Claude อธิบายก่อนทำเรื่องเสี่ยง

ตัวอย่าง rule ที่ควรมี:

```
• MARKDOWN

## Approval rules
- For small copy or UI changes, you may edit directly.
- For changes touching multiple files, explain the plan first.
- For database, auth, payment, deployment, or environment variables, stop and ask bef
- If you are unsure, ask a clarifying question instead of guessing.
```

rule แบบนี้ช่วยให้คุณยังควบคุมงานได้ แม้ไม่อ่าน code ลึก

วิธี update `CLAUDE.md` ระหว่างทำงาน

`CLAUDE.md` ไม่ใช่ไฟล์ที่เขียนครั้งเดียวแล้วจบ

เมื่อเจอ pattern ซ้ำ ให้ update

ตัวอย่าง:

ถ้า Claude ชอบเพิ่ม package โดยไม่จำเป็น ให้เพิ่ม rule:

```
• MARKDOWN

- Prefer built-in browser/framework features before adding new packages.
```

ถ้า Claude ชอบเขียนภาษาอังกฤษ แต่คุณต้องการภาษาไทย ให้เพิ่ม rule:

```
• MARKDOWN

- User-facing copy should be Thai unless explicitly requested otherwise.
```

ถ้า Claude ชอบบอกว่าเสร็จทั้งที่ยังไม่ได้ test ให้เพิ่ม rule:

```
MARKDOWN
- Do not say a task is done until you list what was tested and what was not tested.
```

กฎง่าย ๆ:

> ถ้าคุณต้องแก้ Claude เรื่องเดิมเป็นครั้งที่สอง ให้เขียนลง `CLAUDE.md`

Checklist: `CLAUDE.md` ดีพอหรือยัง

ก่อนใช้จริง ให้เช็ก 10 ข้อนี้

- ☐ อธิบาย project ได้ใน 1–3 ประโยค
- ☐ บอก target user ชัด
- ☐ บอก scope ของ version แรก
- ☐ บอกสิ่งที่ out of scope
- ☐ มี command สำหรับ run/test/build เท่าที่รู้
- ☐ มี rule ให้ plan ก่อนแก้หลายไฟล์
- ☐ มี rule ให้ถามก่อนแตะ database/auth/payment/env/deploy
- ☐ ไม่มี secret หรือข้อมูลลูกค้าจริง
- ☐ มี definition of done
- ☐ ความยาวยังอ่านจบได้ในไม่กี่นาที

ถ้าติ๊กไม่ครบ ไม่เป็นไร

แต่ถ้าขาดเรื่อง scope, out of scope, หรือ security rules ให้เติมก่อนเริ่มงานจริง

Prompt รวมท้ายบท

ใช้ prompt นี้หลังคุณมี `CLAUDE.md` แล้ว

```
ช่วย review CLAUDE.md นี้แบบ non-coder

สิ่งที่ต้องการ:
1. มีอะไรคลุมเครือไหม
2. มีอะไรยาวหรือไม่จำเป็นไหม
3. มี secret หรือข้อมูลที่ไม่ควรอยู่ในไฟล์นี้ไหม
4. มี rule สำคัญที่ควรเพิ่มไหม
5. มี rule ไหนที่อาจทำให้ Claude ทำงานยากเกินไปไหม
6. ช่วยเสนอเวอร์ชันที่สั้นกว่าและชัดกว่า

กติกา:
- อย่าแก้ไฟล์ทันที
- อธิบายเหตุผลก่อน
- ใช้ภาษาที่ non-coder เข้าใจ
```

ถ้าได้เวอร์ชันใหม่แล้วค่อยให้ Claude แก้ไฟล์:

```
โอเค ใช้เวอร์ชันที่สั้นกว่า
ช่วยอัปเดต CLAUDE.md แล้วสรุป diff ให้ฉันตรวจ
```

สรุป

`CLAUDE.md` คือ briefing note ของ project

มันช่วยให้ Claude Code เข้าใจสิ่งที่ต้องรู้ทุกครั้งโดยไม่ต้องให้คุณพิมพ์ซ้ำ

สำหรับ non-coder ให้ใช้มันเพื่อควบคุม 5 เรื่อง:

1. project นี้คืออะไร
2. version แรกทำอะไรและไม่ทำอะไร
3. command สำคัญคืออะไร
4. อะไรเสี่ยง ต้องถามก่อน
5. งานแบบไหนถึงเรียกว่าเสร็จ

บทต่อไป เราจะใช้ context เหล่านี้ไปเขียน spec ให้ชัด เพื่อเปลี่ยนไอเดียกว้าง ๆ ให้กลายเป็นงานที่ Claude Code ลงมือสร้างได้โดยไม่ต้องเดา

$ บทที่ 5

เขียน Spec ที่ AI เอาไปสร้างได้

แก่นของบทนี้

ไอเดียที่ดีไม่พอสำหรับให้ AI สร้าง software

คุณต้องแปลงไอเดียให้เป็น **spec**

Spec คือเอกสารสั้น ๆ ที่บอกว่า:

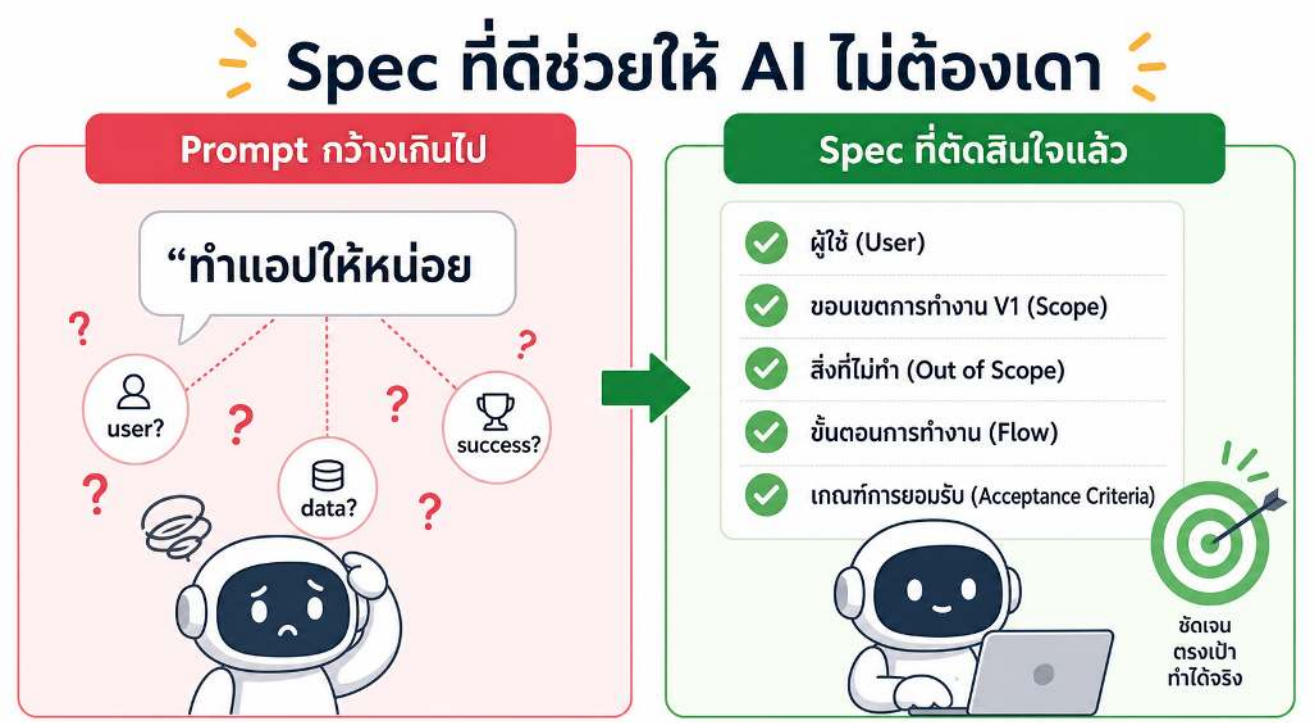
> เราจะสร้างอะไร
> สร้างให้ใคร
> `version` แรกทำอะไรได้บ้าง
> อะไรยังไม่ทำ
> `user` ต้องไหลจากจุดไหนไปจุดไหน
> เสร็จแล้วต้องตรวจยังไง

สำหรับ non-coder spec ไม่ต้องเป็นเอกสารหนา ๆ แบบบริษัทใหญ่

แต่ต้องชัดพอให้ Claude Code ไม่ต้องเดาเรื่องสำคัญ

ถ้า `CLAUDE.md` คือ context ระยะยาวของ project
Spec คือคำสั่งเฉพาะสำหรับงานรอบนี้

ทำไม “ช่วยทำแอปให้หน่อย” ถึงไม่พอ



Spec ที่ดีช่วยให้ AI ไม่ต้องเดา

เมื่อ spec ตัดสินใจเรื่องสำคัญแล้ว AI จะเดาน้อยลง และคุณตรวจงานง่ายขึ้น

คำสั่งแบบนี้กว้างเกินไป:

> ช่วยทำเว็บจองคิวให้หน่อย

Claude Code อาจต้องเดาหลายเรื่อง:

> จองคิวอะไร
> ใครเป็นคนจอง
> ต้อง `login` ไหม
> ต้องมี `admin` ไหม
> ต้องส่ง `email` ไหม
> ต้องกันเวลาซ้ำไหม
> ต้องรับเงินไหม
> ต้องเก็บข้อมูลอะไร
> ใช้ `database` อะไร
> เสร็จแปลว่าอะไร

ยิ่ง AI เดามาก คุณยิ่งเสีย control มาก

Spec ทำให้คำถามพวกนี้ถูกตอบก่อนเริ่ม build

ไม่ใช่ตอบทีหลังตอน project พังหรือบวมเกินไป

Spec ที่ดีต้องสั้นและตัดสินใจได้

Anatomy ของ spec ที่ AI เอาไป build ได้

spec ที่ดีไม่ต้องยาว แต่ควรมี goal, user, scope, out of scope, flow, data, acceptance และ risks

Spec สำหรับเล่มนี้ควรยาวประมาณ 1–3 หน้า

ไม่ต้องเขียนทุกอย่างในหัว

แต่ต้องมี decision สำคัญ

Spec ที่ดีควรตอบ 7 เรื่อง:

1. Goal — ทำไปเพื่ออะไร
2. User — ใครใช้
3. Scope — version แรกมีอะไร
4. Non-goals — อะไรยังไม่ทำ

5. User flow — user เดินทางยังไง
6. Data — ต้องเก็บ/แสดงข้อมูลอะไร
7. Acceptance criteria — แบบไหนเรียกว่าเสร็จ

ถ้าขาด acceptance criteria Claude อาจสร้างของที่ “ดูเสร็จ” แต่คุณตรวจไม่ได้

Claude Code best practices เน้นเรื่องการให้ verification หรือ success criteria เพราะถ้าไม่มี criteria ที่ชัด งานอาจดูถูกแต่ใช้งานจริงไม่ได้

แปลแบบง่าย ๆ:

> ถ้าคุณบอกไม่ได้ว่าเสร็จคืออะไร AI ก็เดาแทนคุณว่าเสร็จคืออะไร

Template Spec สำหรับ non-coder

ใช้ template นี้ได้เกือบทุก project

```markdown
# Spec: [ชื่อ feature/project]

## 1. Goal
เราต้องการสร้าง [สิ่งที่จะสร้าง] เพื่อ [ผลลัพธ์ที่ต้องการ]

## 2. Target user
ผู้ใช้หลักคือ [ใคร]
เขามีปัญหา/ความต้องการคือ [อะไร]

## 3. Version 1 scope
Version แรกต้องมี:
- [feature 1]
- [feature 2]
- [feature 3]

## 4. Out of scope
ยังไม่ทำใน version แรก:
- [สิ่งที่ยังไม่ทำ]
- [สิ่งที่ยังไม่ทำ]

## 5. User flow
1. User เข้าไปที่ [หน้า/จุดเริ่ม]
2. User เห็น [ข้อมูล/CTA]
3. User ทำ [action]
4. ระบบแสดง [ผลลัพธ์]

## 6. Data
ต้องเก็บ/ใช้ข้อมูล:
- [field 1]
- [field 2]

ข้อมูลที่ไม่ควรเก็บ:
- [field ที่เสี่ยง/ไม่จำเป็น]

## 7. Acceptance criteria
งานนี้ถือว่าเสร็จเมื่อ:
- [เงื่อนไขตรวจได้ 1]
- [เงื่อนไขตรวจได้ 2]
- [เงื่อนไขตรวจได้ 3]

## 8. Risks / questions
- [ความเสี่ยงหรือคำถามที่ยังไม่แน่ใจ]
```

ถ้า project ยังเล็กมาก ใช้แค่หัวข้อ 1–7 ก็พอ

อย่าใช้ spec เพื่อทำให้ project ดูใหญ่

ใช้ spec เพื่อทำให้ project เล็กและชัด

ตัวอย่าง: จากไอเดียกว้าง ๆ เป็น spec

ไอเดียกว้าง:

> อยากทำเว็บเก็บ lead สำหรับบริการ consult AI

Spec ที่ใช้ได้:

MARKDOWN

```
# Spec: AI Workflow Audit Landing Page

## 1. Goal
สร้าง landing page สำหรับบริการ AI Workflow Audit เพื่อให้เจ้าของธุรกิจกรอกฟอร์มนัด consult 30 นาที

## 2. Target user
เจ้าของธุรกิจ SME ที่อยากใช้ AI ลดงานซ้ำ แต่ไม่รู้จะเริ่มจากตรงไหน

## 3. Version 1 scope
Version แรกต้องมี:
- Hero section อธิบายบริการแบบสั้น
- Benefits 3 ข้อ
- Process 3 ขั้นตอน
- FAQ 4 ข้อ
- Contact/waitlist form
- Thank you message หลังส่งฟอร์ม

## 4. Out of scope
ยังไม่ทำ:
- Login
- Payment
- Calendar booking integration
- CRM integration
- Admin dashboard

## 5. User flow
1. User เปิด landing page
2. อ่านว่า service ช่วยอะไร
3. กด CTA หรือเลื่อนไปที่ form
4. กรอกชื่อ อีเมล บริษัท และปัญหาที่อยากแก้
5. กด submit
6. เห็น thank you message

## 6. Data
ต้องเก็บ:
- name
- email
- company
- main_problem

ไม่เก็บ:
- phone number
- payment information
- sensitive business documents

## 7. Acceptance criteria
งานนี้ถือว่าเสร็จเมื่อ:
- หน้าเว็บเปิดได้บน desktop และ mobile
- CTA พาไปที่ form ได้
```

```
- Form validate email เบื้องต้น
- Submit แล้วแสดง thank you message
- ไม่มี login/payment/database dashboard ใน version แรก
- Claude สรุปไฟล์ที่แก้และ manual test steps ให้ตรวจ
```

Spec นี้ไม่ได้ยาว

แต่ชัดพอให้ Claude Code ไม่สร้างเกินจำเป็น

Out of scope สำคัญมาก

มือใหม่มักเขียนแต่สิ่งที่อยากได้

แต่ไม่เขียนสิ่งที่ยังไม่ทำ

ผลคือ AI อาจ “ช่วยเกิน”

เช่น คุณขอ waitlist form แล้ว AI อาจเพิ่ม:

```
login
admin dashboard
email automation
analytics
payment
role permission
```

บางอย่างอาจดูดี แต่ทำให้ project บวม

ทุก spec จึงควรมีหัวข้อ:

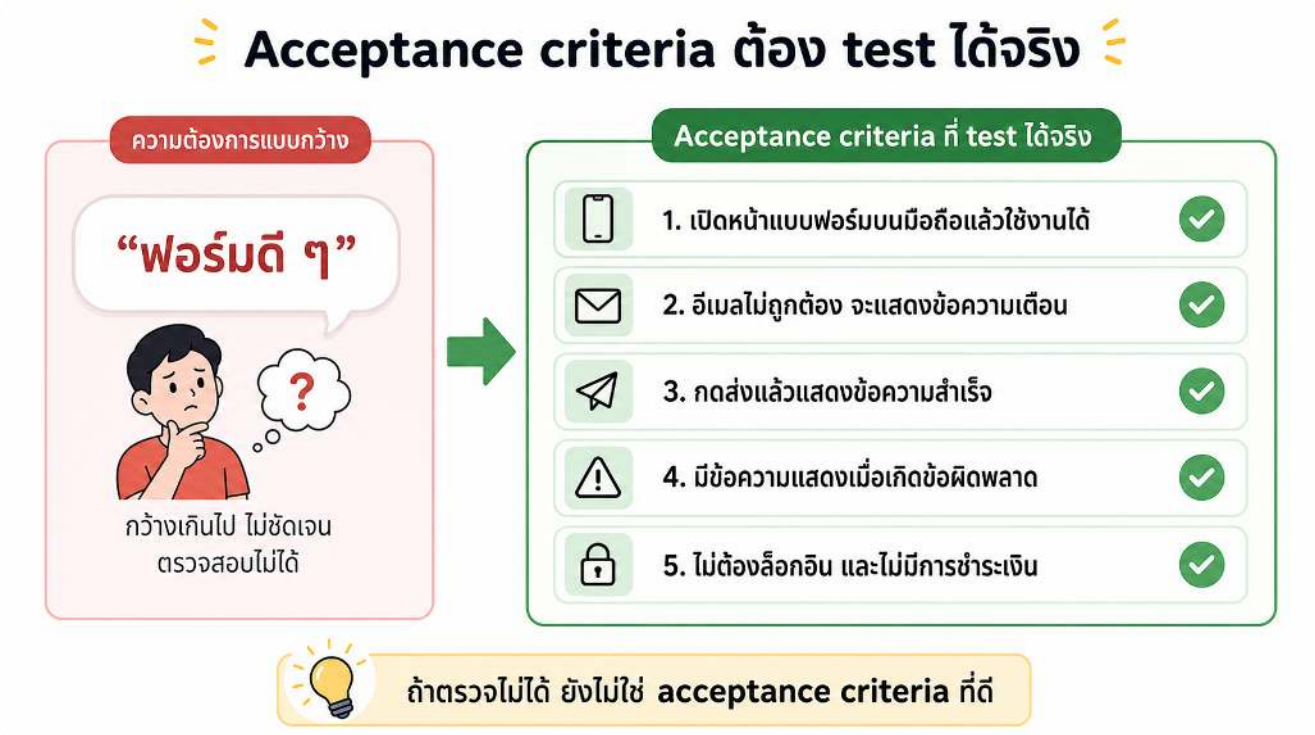
```
MARKDOWN
## Out of scope
```

ตัวอย่าง:

```
- No login in version 1
- No payment in version 1
- No admin dashboard in version 1
- No new third-party services unless approved
- No database schema changes without asking
```

การบอกว่า “ยังไม่ทำอะไร” สำคัญพอ ๆ กับการบอกว่า “จะทำอะไร”

Acceptance criteria ต้องตรวจได้



Acceptance criteria ต้อง test ได้จริง

acceptance criteria ที่ดีต้องเปลี่ยนเป็น checklist ที่คนอ่านกด test เองได้

Acceptance criteria คือเงื่อนไขว่า “งานนี้เสร็จแล้วจริง”

ไม่ดี:

```
เว็บต้องดูดี
ฟอร์มต้องใช้ง่าย
ระบบต้องเร็ว
```

เพราะตรวจยาก

ดีกว่า:

```
หน้าแรกโหลดได้โดยไม่มี error
CTA button scroll ไปที่ form
ถ้า email ไม่มี @ ให้แสดง error
ถ้าส่งฟอร์มสำเร็จ ให้แสดง thank you message
บนมือถือ form ไม่ล้นจอ
Claude ต้องรัน build หรือบอกเหตุผลว่าทำไมรันไม่ได้
```

Acceptance criteria ที่ดีมี 3 ลักษณะ:

1. ตรวจได้
2. ไม่คลุมเครือ
3. ผูกกับ user flow จริง

ถ้าคุณเขียน criteria ดี Claude Code จะช่วย test และ verify งานได้ดีขึ้น

Edge case คืออะไร

Edge case คือกรณีที่ไม่ใช่ flow ปกติ แต่เกิดขึ้นได้

เช่น form ปกติคือ:

```
กรอกข้อมูลครบ → กด submit → สำเร็จ
```

edge case คือ:

```
ไม่กรอก email
email ผิดรูปแบบ
กด submit ซ้ำ
เน็ตหลุด
database error
ข้อความยาวเกินไป
เปิดบนมือถือจอเล็ก
```

non-coder ไม่ต้องคิด edge case ครบทุกอย่าง

แต่ควรถาม Claude ให้ช่วยคิดก่อน build:

```
จาก spec นี้ ช่วย list edge cases ที่ควรตรวจสำหรับ version แรก
แยกเป็น required, optional, และยังไม่ต้องทำ
```

คำว่า “ยังไม่ต้องทำ” สำคัญ

เพราะ edge case มีไม่สิ้นสุด

เราแค่ต้องเลือกสิ่งที่เหมาะกับ version แรก

ให้ Claude interview คุณก่อนเขียน spec

ถ้าคุณยังอธิบาย project ไม่ชัด อย่าเริ่ม build

ให้ Claude ถามคำถามก่อน

Prompt:

```
ฉันอยากสร้าง [ไอเดีย]
แต่ยังอธิบายไม่ชัด
ช่วย interview ฉันทีละคำถามเพื่อทำ spec สำหรับ version แรก

กติกา:
- ถามทีละ 1–3 คำถาม
- ใช้ภาษาที่ non-coder เข้าใจ
- ช่วยลด scope ไม่ใช่เพิ่ม scope
- ถ้ามีคำตอบที่เสี่ยง ให้เตือน
- เมื่อข้อมูลพอแล้ว สรุปเป็น spec 1–3 หน้า
```

วิธีนี้ดีมากสำหรับ non-coder

เพราะคุณไม่ต้องรู้ตั้งแต่แรกว่าต้องเขียน spec ยังไง

ให้ Claude ช่วยถาม แต่คุณยังเป็นคนตัดสินใจ

Prompt: ให้ Claude แปลงไอเดียเป็น spec

ใช้ prompt นี้เมื่อคุณมีไอเดียแล้ว

```
ช่วยแปลงไอเดียนี้เป็น spec สำหรับ Claude Code

ไอเดีย:
[อธิบายไอเดีย]

บริบท:
- target user:
- goal:
- deadline:
- ข้อมูลที่อาจต้องเก็บ:
- สิ่งที่กังวล:

กติกา:
- ทำ version แรกให้เล็กที่สุดแต่ยังมีประโยชน์
- แยก scope กับ out of scope ให้ชัด
- ห้ามใส่ login/payment/admin dashboard ถ้าไม่จำเป็น
- ถ้ามีเรื่อง data/security/payment ให้ใส่ใน risk section
- ใช้ภาษาที่ non-coder เข้าใจ

Output:
1. Goal
2. Target user
3. Version 1 scope
4. Out of scope
5. User flow
6. Data fields
7. Acceptance criteria
8. Edge cases
9. Questions before build
```

หลังได้ spec แล้ว ให้ถามต่อ:

```
ช่วย critique spec นี้
มีอะไรยังคลุมเครือ เกิน scope หรือเสี่ยงเกินไปสำหรับ version แรกไหม
```

อย่าให้ AI เขียน spec แล้ว build ต่อทันที

ให้มีจังหวะ review ก่อน

Prompt: ให้ Claude วางแผนจาก spec โดยยังไม่แก้ไฟล์

เมื่อ spec พร้อมแล้ว ค่อยให้ Claude ดู project

```
นี่คือ spec สำหรับงานนี้:
[paste spec]

ช่วยอ่าน project นี้และเสนอ implementation plan

กติกา:
- อย่าเพิ่งแก้ไฟล์
- บอกว่าไฟล์ไหนน่าจะต้องแก้
- บอกว่ามีอะไรใน spec ที่ยังไม่ชัด
- บอก risk ที่ควรระวัง
- บอก manual test steps หลังทำเสร็จ
- ถ้า scope ใหญ่เกินไป ให้เสนอ version ที่เล็กกว่า
```

นี่คือจังหวะที่ใช้ Plan mode ได้ดีมาก

เพราะคุณให้ Claude เชื่อม spec กับ codebase จริงก่อนลงมือ

Checklist: Spec พร้อม build หรือยัง

ก่อนให้ Claude Code แก้ไฟล์ ให้เช็ก 12 ข้อนี้

- ☐ อธิบาย goal ได้ใน 1 ประโยค
- ☐ รู้ว่า user คือใคร
- ☐ version แรกมี feature หลักไม่เกิน 3–5 อย่าง
- ☐ มี out of scope ชัด
- ☐ user flow เขียนเป็นขั้นตอนได้
- ☐ รู้ว่าจะเก็บข้อมูลอะไร
- ☐ ตัดข้อมูลที่ไม่จำเป็นออกแล้ว
- ☐ acceptance criteria ตรวจได้จริง

- ☐ มี edge cases สำคัญอย่างน้อย 3 ข้อ
- ☐ มี risk section ถ้าเกี่ยวกับ data/security/payment/login
- ☐ มีคำถามที่ยังไม่ชัดแยกไว้
- ☐ Claude ยังไม่ได้เริ่มแก้ไฟล์ก่อนคุณ approve plan

ถ้าติ๊กไม่ครบ โดยเฉพาะ scope/out of scope/acceptance criteria ให้กลับไปแก้ spec ก่อน

สิ่งที่ควรเก็บเป็นไฟล์

สำหรับงานที่ใหญ่กว่า 1 prompt ให้เก็บ spec เป็นไฟล์

เช่น:

```
SPEC.md
```

หรือ:

```
docs/spec-waitlist-v1.md
```

ข้อดีคือ:

- กลับมาอ่านซ้ำได้
- ให้ Claude อ้างอิงได้
- review ก่อน build ได้
- ใช้เป็น checklist ตอน test ได้
- ส่งให้ developer ช่วยดูได้

ถ้า spec อยู่แค่ใน chat พอ session ยาวขึ้น context อาจรก

ไฟล์ spec ทำให้ project เป็นระบบกว่า

สรุป

Spec คือสะพานระหว่างไอเดียกับการลงมือ build

สำหรับ non-coder spec ไม่ต้องยาว แต่ต้องตัดสินใจแทน AI ในเรื่องสำคัญ

จำ 5 ข้อนี้:

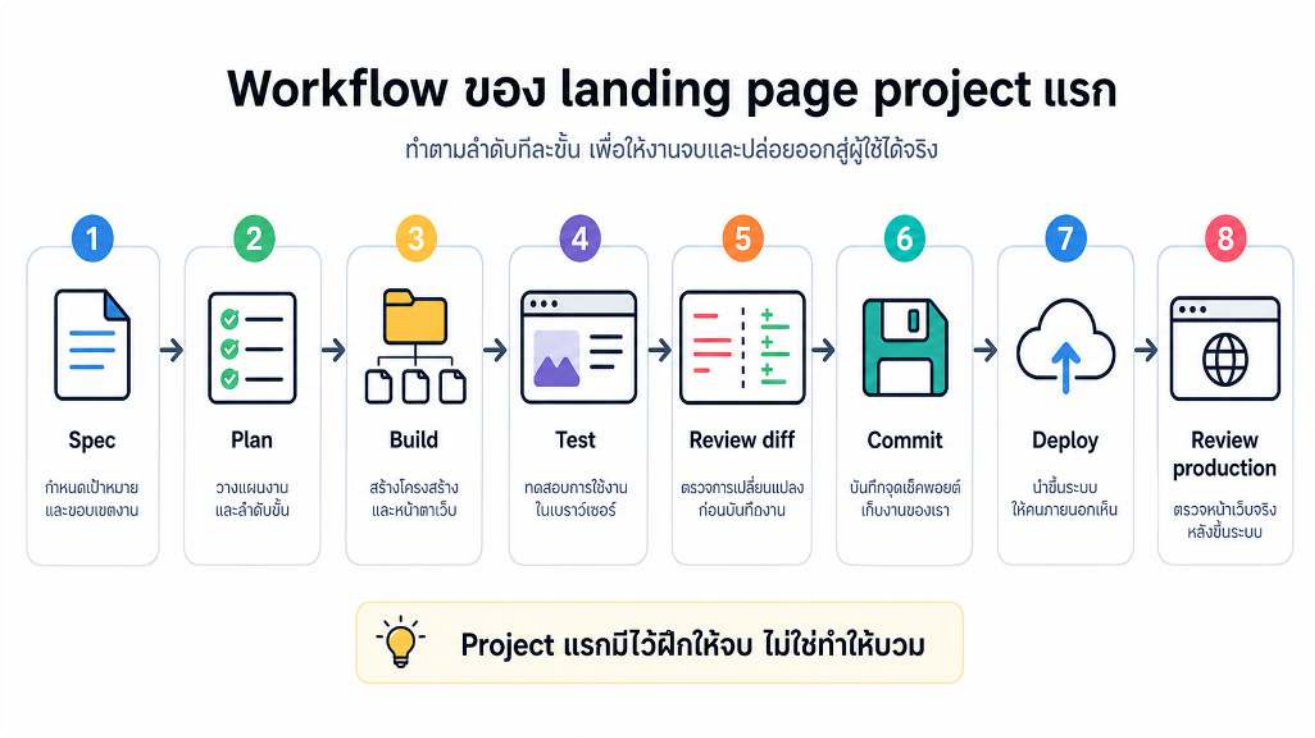
1. อย่าเริ่มจาก “ทำแอปให้หน่อย”
2. เขียน scope และ out of scope เสมอ
3. Acceptance criteria ต้องตรวจได้
4. ให้ Claude interview คุณได้ แต่คุณต้องตัดสินใจ
5. ให้ Claude วางแผนจาก spec ก่อนแก้ไฟล์

บทต่อไป เราจะใช้ spec แบบนี้ไปสร้าง project แรก: landing page ที่ publish ได้ โดยยังคุม scope ให้เล็กและตรวจเองได้

$ บทที่ 6

Project 1: Landing Page ที่ publish ได้

แก่นของบทนี้



Workflow ของ landing page project แรก

ภาพนี้คือ workflow หลักของ project แรก ตั้งแต่ spec จนถึง review หลัง deploy

Project แรกไม่ควรเป็นแอปใหญ่

ควรเป็นสิ่งที่เล็กพอให้คุณเห็น workflow ครบตั้งแต่ต้นจนจบ

ในบทนี้เราจะทำ **landing page 1 หน้า**

เป้าหมายไม่ใช่ทำเว็บที่สวยที่สุดในโลก

เป้าหมายคือให้คุณฝึก workflow นี้:

```
Spec → Plan → Build → Test → Fix → Deploy → Review
```

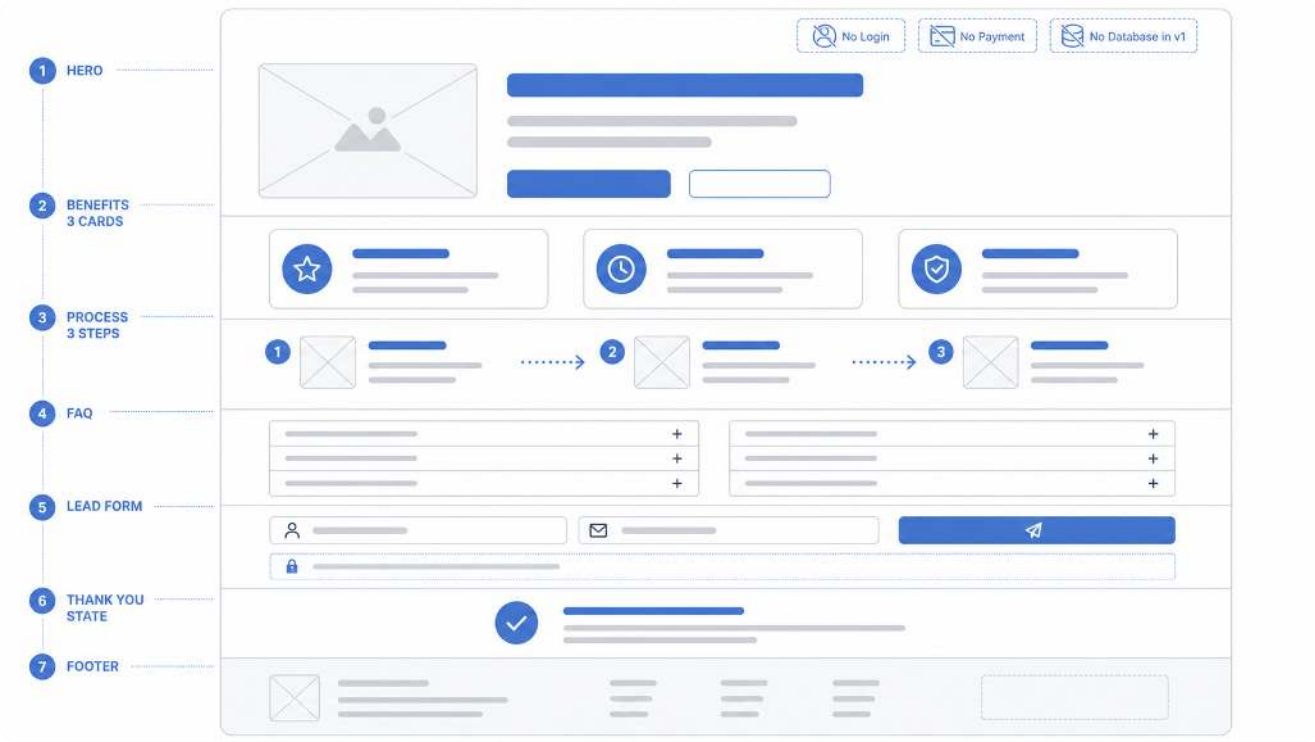
Landing page เหมาะเป็น project แรก เพราะ:

- scope เล็ก
- ตรวจเองได้ง่าย
- ยังไม่ต้องมี login
- ยังไม่ต้องมี payment
- ยังไม่ต้องมี database ซับซ้อน
- deploy แล้วเห็นผลจริงทันที

ถ้าคุณทำบทนี้จบ คุณจะเข้าใจแกนของ vibe coding มากกว่าการอ่านทฤษฎีอีกหลายบท

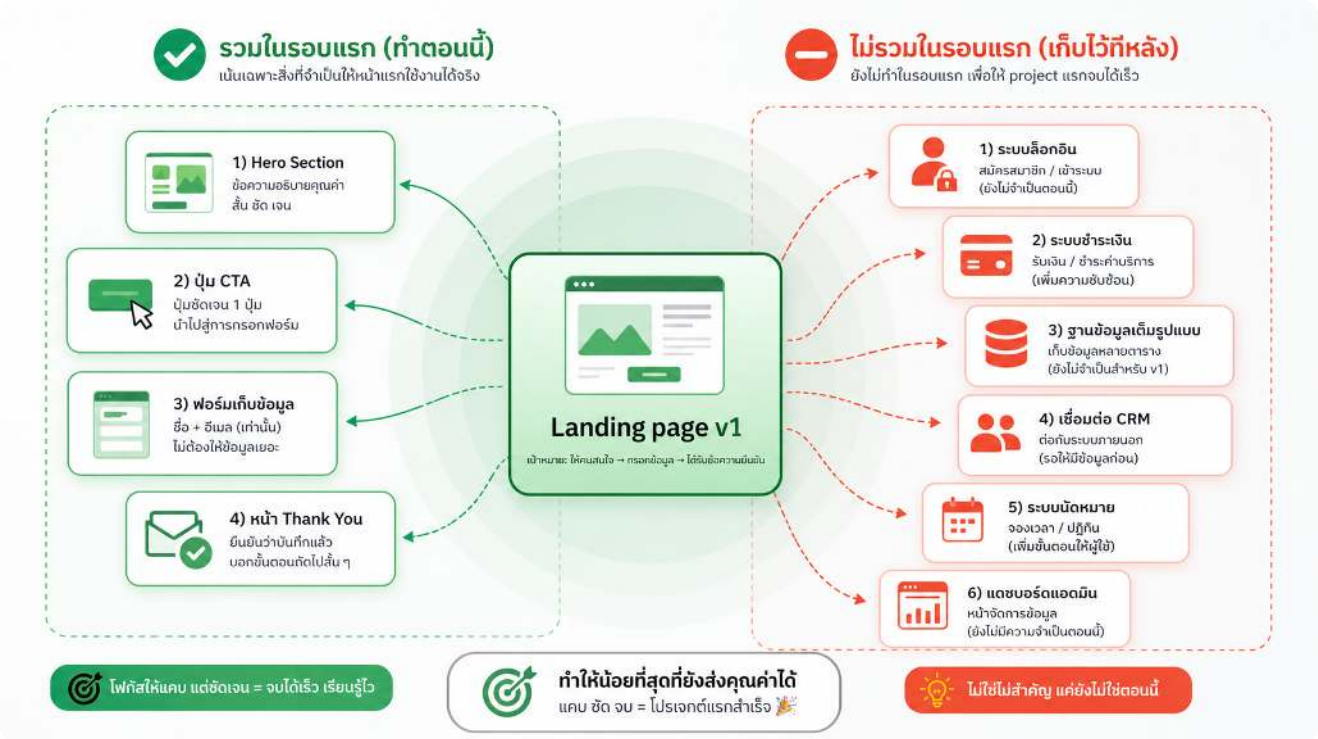
ถ้าจะ deploy จริง ให้ใช้บทนี้คู่กับ **Appendix D — Vercel Deploy Playbook** บทนี้บอก workflow หลัก ส่วน Appendix D บอกขั้นตอน deploy/check/rollback แบบละเอียดกว่า

Project ที่เราจะสร้าง



Wireframe ของ landing page v1 ที่ไม่บวม

หน้าแรกไม่ต้องใหญ่ แค่มี section สำคัญพอให้ user เข้าใจและกรอก form/mock form ได้



Out of scope ช่วยให้ project แรกจบ

project แรกจบได้เพราะตั้งใจตัด login, payment, database และ dashboard ออกก่อน

ตัวอย่างในบทนี้คือ landing page สำหรับบริการสมมติ:

```
AI Workflow Audit
```

บริการนี้ช่วยเจ้าของธุรกิจดูว่างานซ้ำ ๆ ตรงไหนควรใช้ AI ช่วยได้บ้าง

Version แรกมีแค่นี้:

- headline
- คำอธิบายสั้น
- benefits 3 ข้อ
- process 3 ขั้นตอน
- FAQ สั้น ๆ
- CTA ให้กรอกฟอร์ม
- form แบบ mock หรือแบบยังไม่ต่อ database
- thank you state หลัง submit

ยังไม่ทำ:

- login

- payment
- calendar booking
- CRM integration
- admin dashboard
- database จริง

จงสังเกตว่าเราตั้งใจตัดของออกเยอะมาก

เพราะ project แรกมีไว้ฝึกให้จบ ไม่ใช่ฝึกให้บวม

Step 1: เขียน spec สั้น ๆ

ก่อนให้ Claude Code แก้ไฟล์ ให้เริ่มจาก spec

MARKDOWN

```
# Spec: AI Workflow Audit Landing Page

## Goal
สร้าง landing page 1 หน้า เพื่อให้เจ้าของธุรกิจ SME สนใจกรอกฟอร์มนัด consult 30 นาที

## Target user
เจ้าของธุรกิจ SME ที่อยากใช้ AI ลดงานซ้ำ แต่ไม่รู้จะเริ่มจากตรงไหน

## Version 1 scope
- Hero section
- Benefits 3 ข้อ
- Process 3 ขั้นตอน
- FAQ 4 ข้อ
- Lead form UI
- Thank you message หลัง submit

## Out of scope
- No login
- No payment
- No database
- No calendar integration
- No admin dashboard

## User flow
1. User เปิดหน้า landing page
2. อ่านว่า service ช่วยอะไร
3. กด CTA หรือเลื่อนไปที่ form
4. กรอกชื่อ อีเมล บริษัท และปัญหาที่อยากแก้
5. กด submit
6. เห็น thank you message

## Data fields
- name
- email
- company
- main_problem

## Acceptance criteria
- หน้าเว็บเปิดได้บน desktop และ mobile
- CTA พาไปที่ form ได้
- Form มี validation เบื้องต้น
- Submit แล้วแสดง thank you message
- ไม่มี login/payment/database ใน version แรก
- Claude สรุปไฟล์ที่แก้และ manual test steps ให้ตรวจ
```

ถ้าคุณมี project จริง ให้เปลี่ยนชื่อบริการและ target user เป็นของคุณ

แต่ยังแนะนำให้รักษา scope ให้เล็กแบบนี้ก่อน

Step 2: ให้ Claude วางแผนก่อน

เปิด Claude Code ใน project folder แล้วใช้ prompt นี้

```
นี่คือ spec สำหรับ landing page แรก:
[paste spec]

ช่วยอ่าน project นี้และเสนอ implementation plan

กติกา:
- อย่าเพิ่งแก้ไฟล์
- บอกว่าไฟล์ไหนน่าจะแก้
- บอกว่าต้องเพิ่ม component หรือ state อะไรไหม
- บอกว่า scope มีอะไรที่เสี่ยงเกิน version แรกไหม
- บอก manual test steps หลังทำเสร็จ
- ใช้ภาษาที่ non-coder เข้าใจ
```

ถ้า Claude เสนอ plan ที่ใหญ่เกินไป เช่นเพิ่ม database หรือ auth ให้หยุดทันที

ตอบกลับว่า:

```
ลด scope กลับมาเฉพาะ version 1
ยังไม่ต้องมี database, login, payment หรือ integration ใด ๆ
ช่วยเสนอ plan ใหม่ที่เล็กกว่าและเปลี่ยนไฟล์น้อยที่สุด
```

บทนี้ไม่ต้องการความอลังการ

ต้องการความจบ

Step 3: ให้ Claude build แบบเปลี่ยนน้อยที่สุด

เมื่อ plan ดูโอเคแล้ว ค่อยให้ลงมือ

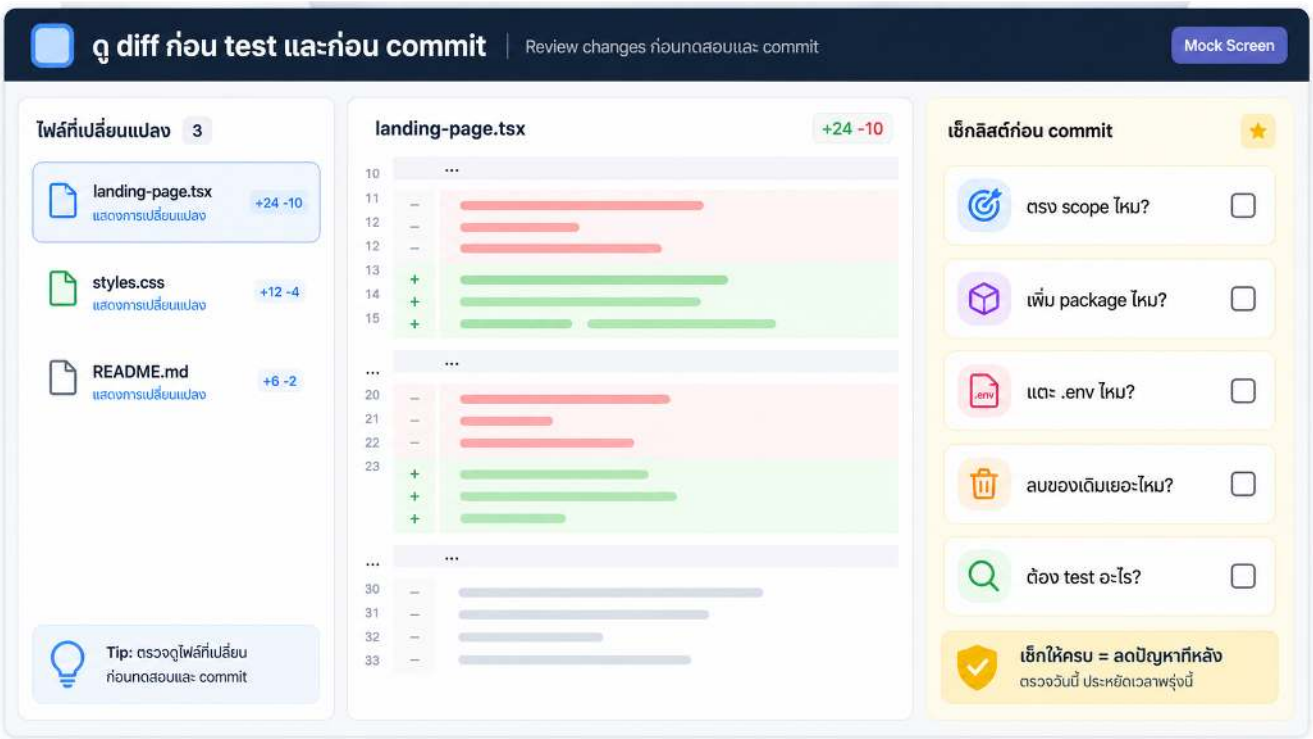
```
โอเค ทำตาม plan ได้
แต่ขอให้เปลี่ยนน้อยที่สุดเท่าที่จำเป็น
หลังทำเสร็จให้สรุป:
1. แก้ไฟล์ไหนบ้าง
2. แต่ละไฟล์แก้เพื่ออะไร
3. ฉันต้อง test อะไร
4. มีอะไรที่ยังไม่ได้ทำตาม scope บ้าง
```

คำว่า “เปลี่ยนน้อยที่สุด” สำคัญมาก

เพราะ AI มักมีแนวโน้มจัดใหญ่เกินจำเป็นถ้าคุณไม่กำกับ

สำหรับ project แรก ยิ่งเปลี่ยนน้อย ยิ่ง review ง่าย

Step 4: ดู diff ก่อน test



ดู diff ก่อน test และก่อน commit

หลัง Claude แก้ไฟล์ ให้ดู diff ก่อน เพื่อเช็กว่าแก้ตรง scope และไม่แตะไฟล์เสี่ยง

หลัง Claude แก้ไฟล์ ให้ดูว่าเปลี่ยนอะไร

ถ้าใช้ Git ให้ดูผ่าน:

```bash
git status
git diff
```

หรือถาม Claude:

> ช่วยอธิบาย diff นี้แบบ non-coder
> แยกเป็น:
> 1. ไฟล์ไหนถูกแก้
> 2. แก้เพื่ออะไร
> 3. มีไฟล์ไหนที่ไม่เกี่ยวกับ landing page ไหม
> 4. มีความเสี่ยงอะไรไหม

สิ่งที่ควรมองหา:

- มันแก้ไฟล์ที่เกี่ยวข้องจริงไหม
- มันเพิ่ม package ใหม่หรือเปล่า
- มันแตะไฟล์ `.env` หรือ config แปลก ๆ ไหม
- มันเพิ่ม login/payment/database ทั้งที่ out of scope ไหม
- มันลบของเดิมเยอะผิดปกติไหม

ถ้าไม่มั่นใจ อย่าเพิ่งไปต่อ

ให้ถามก่อน

Step 5: Test แบบ user จริง

การ test สำหรับ landing page ไม่ซับซ้อน

เปิดหน้าเว็บแล้วลองเหมือน user จริง

Checklist:

- ☐ หน้าเว็บเปิดได้
- ☐ headline อ่านรู้เรื่อง
- ☐ CTA มองเห็นชัด
- ☐ CTA กดแล้วไปที่ form หรือ section ที่ถูกต้อง

- ☐ benefits เข้าใจง่าย
- ☐ form กรอกได้
- ☐ ถ้า email ผิด ระบบเตือน
- ☐ submit แล้วเห็น thank you message
- ☐ เปิดบนมือถือแล้วไม่ล้นจอ
- ☐ ไม่มี error ชัดเจนในหน้าเว็บ

ถ้าคุณไม่รู้จะรันเว็บยังไง ให้ถาม Claude:

> ฉันจะเปิดเว็บนี้ในเครื่องเพื่อ test ได้ยังไง
> ช่วยบอก command ที่ต้องใช้ และอธิบายแต่ละ command แบบ non-coder
> อย่าเพิ่งรัน command จนกว่าฉันจะ approve

ถ้าเจอ error ให้ copy error ไปให้ Claude พร้อมขั้นตอนที่ทำ

ไม่ดี:

> มันพัง แก้หน่อย

ดีกว่า:

> ตอนฉันรัน npm run dev แล้วเปิดหน้าเว็บ เจอ error นี้:
> [paste error]
>
> ขั้นตอนที่ทำ:
> 1. run npm run dev
> 2. เปิด URL นี้
> 3. กดปุ่ม CTA
>
> expected behavior:
> ควร scroll ไปที่ form
>
> ช่วยอธิบายสาเหตุที่เป็นไปได้ก่อน แล้วเสนอวิธีแก้ที่เปลี่ยนน้อยที่สุด

Step 6: ตรวจ mobile และ accessibility เบื้องต้น

Landing page ที่ดีต้องไม่ใช่แค่สวยบนจอคุณ

ต้องอ่านได้บนมือถือ และคนใช้ควรเข้าถึงได้มากที่สุด

MDN อธิบาย accessibility ว่าคือการทำให้คนจำนวนมากที่สุดใช้เว็บได้ รวมถึงคนที่มีข้อจำกัดด้านความสามารถหรือวิธีเข้าถึงเว็บ

สำหรับ project แรก ไม่ต้องทำ WCAG audit เต็มรูปแบบ

แต่ควรเช็กขั้นต่ำ:

- ☐ text contrast อ่านได้
- ☐ font ไม่เล็กเกินไป
- ☐ ปุ่มมีข้อความชัด ไม่ใช่ “คลิกที่นี่” อย่างเดียว
- ☐ form field มี label
- ☐ ใช้ keyboard tab ไปที่ input/button ได้
- ☐ รูปภาพสำคัญมี alt text
- ☐ layout ไม่พังบนมือถือ

ใช้ prompt นี้:

```
ช่วย review landing page นี้ด้าน mobile และ accessibility เบื้องต้น
ฉันไม่ต้องการ audit เต็มรูปแบบ
ขอ checklist ที่ non-coder test เองได้:
1. mobile layout
2. readability
3. button/link clarity
4. form labels
5. keyboard navigation
6. alt text

ถ้ามีจุดที่ควรแก้ ให้เสนอเป็นรายการเล็ก ๆ และอย่าเพิ่งแก้ไฟล์
```

Step 7: ตรวจ performance แบบง่าย

สำหรับ landing page แรก อย่าเพิ่ง optimize ลึก

แต่ควรรู้ว่าหน้าเว็บไม่ควรหนักเกินไป

Google Web Vitals แบ่งคุณภาพประสบการณ์ผู้ใช้เป็นเรื่องอย่าง loading, interactivity และ visual stability

สำหรับ non-coder ให้แปลเป็นคำถามง่าย ๆ:

> หน้าโหลดช้ามากไหม
> กดปุ่มแล้วตอบสนองไหม
> layout กระโดดไปมาระหว่างโหลดไหม
> รูปภาพใหญ่เกินไปไหม

ใช้ prompt:

```
ช่วยตรวจ performance เบื้องต้นของ landing page นี้
โฟกัสเฉพาะสิ่งที่ non-coder ควรเข้าใจ:
- รูปภาพใหญ่เกินไปไหม
- มี package หรือ script ที่ไม่จำเป็นไหม
- layout มีโอกาสกระโดดตอนโหลดไหม
- มีอะไรควรแก้ก่อน deploy ไหม
อย่าเพิ่งแก้ไฟล์ ให้เสนอ checklist ก่อน
```

ไม่ต้องทำให้ perfect

แต่ต้องไม่แย่แบบชัดเจน

Step 8: Commit version แรก

ถ้า test ผ่านแล้ว ให้บันทึก checkpoint

```bash
git status
git add .
git commit -m "feat: add landing page"
```

ถ้าคุณไม่ถนัด Git ให้ให้ Claude ช่วยอธิบายก่อน:

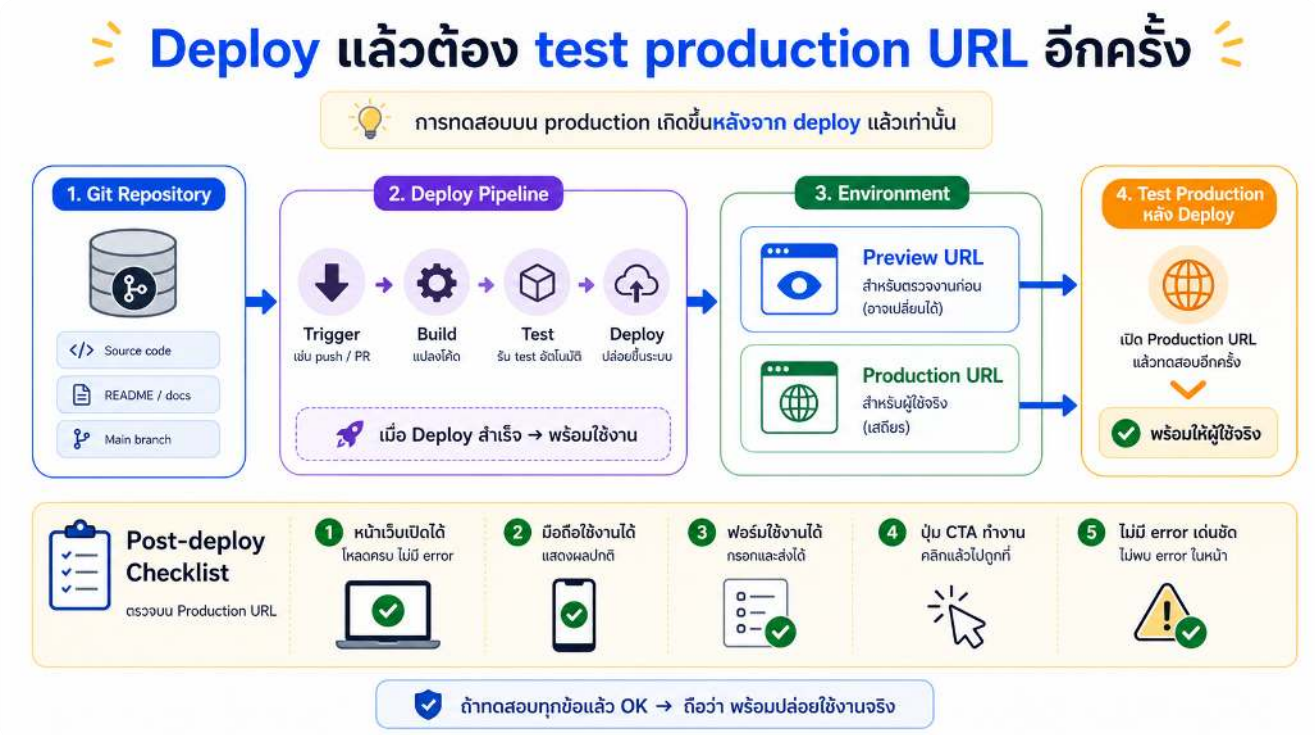
> ช่วยดู `git status` แล้วอธิบายว่าไฟล์ไหนเปลี่ยน
> ถ้าทุกอย่างโอเค ช่วยเสนอ `commit message` แบบสั้น
> อย่า `commit` จนกว่าฉันจะ `approve`

สำหรับ non-coder การ commit คือการบอกว่า:

> จุดนี้คือ version ที่ฉันยอมรับแล้ว

อย่า deploy สิ่งที่คุณยังไม่กล้า commit

Step 9: Deploy ด้วย Vercel



Deploy แล้วต้อง test production URL อีกครั้ง

deploy สำเร็จไม่ได้แปลว่าใช้งานได้ ต้องเปิด production URL แล้ว test flow จริงอีกครั้ง

ถ้าต้องทำตามหน้าจอจริง ให้เปิด **Appendix D — Vercel Deploy Playbook** ก่อน แล้วกลับมาใช้ checklist สั้นใน section นี้

เมื่อ landing page พร้อมแล้ว คุณสามารถ deploy ด้วย Vercel ผ่าน Git repository ได้

ตาม official docs ของ Vercel การเชื่อม Git repository ช่วยให้มี deployment จาก branch และมี preview deployments สำหรับการเปลี่ยนแปลงก่อน production

สำหรับ project แรก flow ง่าย ๆ คือ:

1. push project ไป GitHub
2. สร้าง project ใหม่ใน Vercel
3. เลือก repository
4. ตรวจ framework/build settings
5. กด deploy
6. เปิด URL ที่ Vercel ให้มา
7. test อีกครั้งบน URL จริง

หลัง deploy อย่าคิดว่าเสร็จทันที

ต้อง test บน production URL อีกครั้ง

Expected result ที่ควรเห็น:

```
กด production URL แล้วหน้าเปิดได้
CTA ทำงาน
form ทำงานตาม scope
ไม่มี error ชัดเจน
```

Stop rule:

```
ถ้า URL เปิดไม่ได้, build fail, env var หาย, หรือ form พังบน production
อย่าแชร์ link ต่อ ให้กลับไปเก็บ log ตาม Appendix F ก่อน
```

Checklist หลัง deploy:

- ☐ หน้าเว็บเปิดจาก URL จริงได้
- ☐ mobile ยังไม่พัง
- ☐ form ยังทำงานตามที่ตั้งใจ
- ☐ ไม่มีข้อมูล test แปลก ๆ โผล่
- ☐ title/meta พอใช้ได้
- ☐ CTA ยังชัด

ถ้า deploy แล้วพัง ให้กลับไปดู error log หรือ rollback ถ้าจำเป็น

Vercel มี Instant Rollback สำหรับย้อนกลับ production deployment ไปยัง deployment ก่อนหน้า แต่ก่อนใช้ rollback ต้องเข้าใจว่ามันย้อนกลับ deployment ไม่ได้แก้สาเหตุของ bug ให้เอง

Step 10: เขียน launch note สั้น ๆ

หลัง deploy project แรก ให้เขียน note สั้น ๆ ไว้

```markdown
# Launch Note

## URL
[production URL]

## What shipped
- Landing page for AI Workflow Audit
- Lead form UI
- Thank you state

## What is not included
- No login
- No payment
- No database
- No CRM integration

## Manual tests done
- Desktop open
- Mobile open
- CTA scroll
- Form validation
- Submit thank you state

## Known issues
- [ถ้ามี]

## Next possible improvements
- Connect form to database
- Add email notification
- Improve copy after feedback
```

Launch note ทำให้คุณไม่หลงว่า version นี้ทำอะไรได้จริง

และช่วยให้บทต่อไปต่อยอดง่ายขึ้น

Prompt รวมท้ายบท

ใช้ prompt นี้กับ Claude Code เมื่อจะทำ project landing page

```
ฉันต้องการสร้าง landing page version แรกจาก spec นี้:
[paste spec]

กติกา:
- ทำเฉพาะ version 1
- ห้ามเพิ่ม login, payment, database, calendar, CRM หรือ admin dashboard
- ถ้าต้องเพิ่ม package ใหม่ ให้ถามก่อนและอธิบายเหตุผล
- ก่อนแก้ไฟล์ ให้เสนอ implementation plan ก่อน
- หลังแก้ไฟล์ ให้สรุป diff และ manual test steps
- ใช้ภาษาที่ non-coder เข้าใจ

Acceptance criteria:
- หน้าเปิดได้บน desktop และ mobile
- CTA พาไปที่ form
- Form มี validation เบื้องต้น
- Submit แล้วแสดง thank you message
- ไม่มี feature นอก scope
```

หลัง build เสร็จ ใช้ prompt นี้ตรวจ:

```
ช่วย review landing page นี้ก่อน deploy
ตรวจเฉพาะระดับ version แรก:
1. ตรง spec ไหม
2. มีอะไรเกิน scope ไหม
3. mobile ใช้ได้ไหม
4. accessibility เบื้องต้นโอเคไหม
5. performance มีจุดเสี่ยงชัดเจนไหม
6. ต้อง test อะไรก่อน deploy
7. มีอะไรควรแก้ก่อน หรือ deploy ได้แล้ว

อย่าเพิ่งแก้ไฟล์ ให้รายงานก่อน
```

สรุป

Landing page เป็น project แรกที่ดี เพราะเล็กพอให้จบและตรวจเองได้

บทนี้ไม่ได้สอนให้ทำเว็บสวยที่สุด

แต่สอนให้คุณทำ workflow ที่ถูกต้อง:

```
spec ก่อน
plan ก่อน
build เล็ก ๆ
test behavior
ดู diff
commit
deploy
test production
```

ถ้าคุณทำ flow นี้ได้กับ landing page คุณจะพร้อมต่อยอดไป project ที่มี data จริงมากขึ้น

บทต่อไป เราจะเพิ่มความยากขึ้นหนึ่งระดับ: lead capture หรือ waitlist app ที่เริ่มมี database และต้องคิดเรื่อง key, permission และข้อมูลที่เก็บให้จริงจังกว่าเดิม

$ บทที่ 7

Project 2: Waitlist App ที่เริ่มมี Database

แก่นของบทนี้

บทที่แล้วเราสร้าง landing page ที่ยังไม่ต้องมี database จริง

บทนี้เราจะเพิ่มความจริงขึ้นอีกหนึ่งขั้น:

> `user กรอกฟอร์ม → ข้อมูลถูกเก็บใน Supabase`

นี่คือจุดที่ vibe coding เริ่มมีความเสี่ยงมากขึ้น

เพราะเราไม่ได้แค่ทำหน้าเว็บแล้ว

เราเริ่มเก็บข้อมูลคนจริง

ดังนั้นบทนี้จะเน้น 3 เรื่อง:

1. เก็บข้อมูลให้น้อยที่สุด
2. ใช้ key ให้ถูกประเภท
3. เปิด database access เท่าที่จำเป็น

เป้าหมายไม่ใช่ทำระบบหลังบ้านสมบูรณ์

เป้าหมายคือทำ waitlist app version แรกที่เล็กพอ ตรวจได้ และไม่ประมาทเรื่องข้อมูล

ถ้าจะลงมือทำจริง ให้เปิด **Appendix E — Supabase Waitlist Playbook** คู่กับบทนี้ เพราะบทนี้อธิบายหลักคิด ส่วน Appendix E ลงรายละเอียด checklist, RLS, key และ production test

---

Project ที่เราจะสร้าง

เราจะต่อยอดจาก landing page เดิม

เพิ่ม form ที่เก็บข้อมูลลง Supabase

Version แรกมีแค่นี้:

- form สำหรับ waitlist
- field พื้นฐาน
- submit ไป Supabase
- success state
- error state
- manual test checklist

ยังไม่ทำ:

- login
- admin dashboard
- email automation
- payment
- export CSV
- user account
- role permission ซับซ้อน

จำไว้ว่า:

> แค่เก็บ lead ได้อย่างปลอดภัยพอสำหรับ version แรก ก็ถือว่า project นี้สำเร็จแล้ว

ข้อมูลที่ควรเก็บให้น้อยที่สุด

ก่อนสร้าง table ให้ถามก่อนว่า “จำเป็นต้องเก็บอะไรจริง ๆ”

สำหรับ waitlist ส่วนใหญ่ เริ่มจาก field แบบนี้พอ:

```
name
email
company
main_problem
created_at
```

อย่าเพิ่งเก็บ:

> เบอร์โทร
> งบประมาณละเอียด
> เอกสารบริษัท
> ข้อมูลลูกค้าเขา
> รหัสผ่าน
> ข้อมูลบัตรเครดิต

ถ้ายังไม่จำเป็น อย่าเก็บ

ข้อมูลที่ไม่เก็บ ไม่มีวันหลุดจากระบบคุณ

นี่คือ security ที่ง่ายที่สุดสำหรับ non-coder

Spec สำหรับ Waitlist version แรก

ใช้ spec นี้เป็นฐาน

MARKDOWN

```
# Spec: Waitlist Form with Supabase

## Goal
ให้ผู้สนใจบริการ AI Workflow Audit กรอกข้อมูลเพื่อ join waitlist หรือขอนัด consult

## Target user
เจ้าของธุรกิจ SME ที่สนใจใช้ AI ลดงานซ้ำ

## Version 1 scope
- เพิ่ม waitlist form บน landing page
- เก็บข้อมูลลง Supabase table
- แสดง success message เมื่อส่งสำเร็จ
- แสดง error message เมื่อส่งไม่สำเร็จ
- Validate required fields เบื้องต้น

## Out of scope
- No login
- No payment
- No admin dashboard
- No email automation
- No file upload
- No sensitive documents

## Data fields
- name: required text
- email: required text, basic email validation
- company: optional text
- main_problem: optional text
- created_at: auto timestamp

## Acceptance criteria
- Form submit แล้ว row ใหม่ถูกสร้างใน Supabase
- ถ้าไม่กรอก name/email ต้องแสดง error
- ถ้า email format ผิด ต้องแสดง error
- ถ้า Supabase insert fail ต้องแสดง error ที่ user เข้าใจได้
- ไม่มี secret key อยู่ใน frontend code
- ไม่มี `.env` หรือ `.env.local` ถูก commit
- Claude สรุป manual test steps หลังทำเสร็จ
```

Spec นี้เล็กพอสำหรับเริ่ม

อย่าเพิ่ม dashboard จนกว่าคุณจะมั่นใจว่าเก็บข้อมูลได้ถูกและปลอดภัยพอ

เข้าใจ Supabase แบบ non-coder

สำหรับบทนี้ ให้เข้าใจ Supabase ง่าย ๆ ว่าเป็นบริการที่ช่วยให้เรามี:

```
database
API
auth
storage
```

แต่ project นี้ใช้แค่ database/API พื้นฐาน

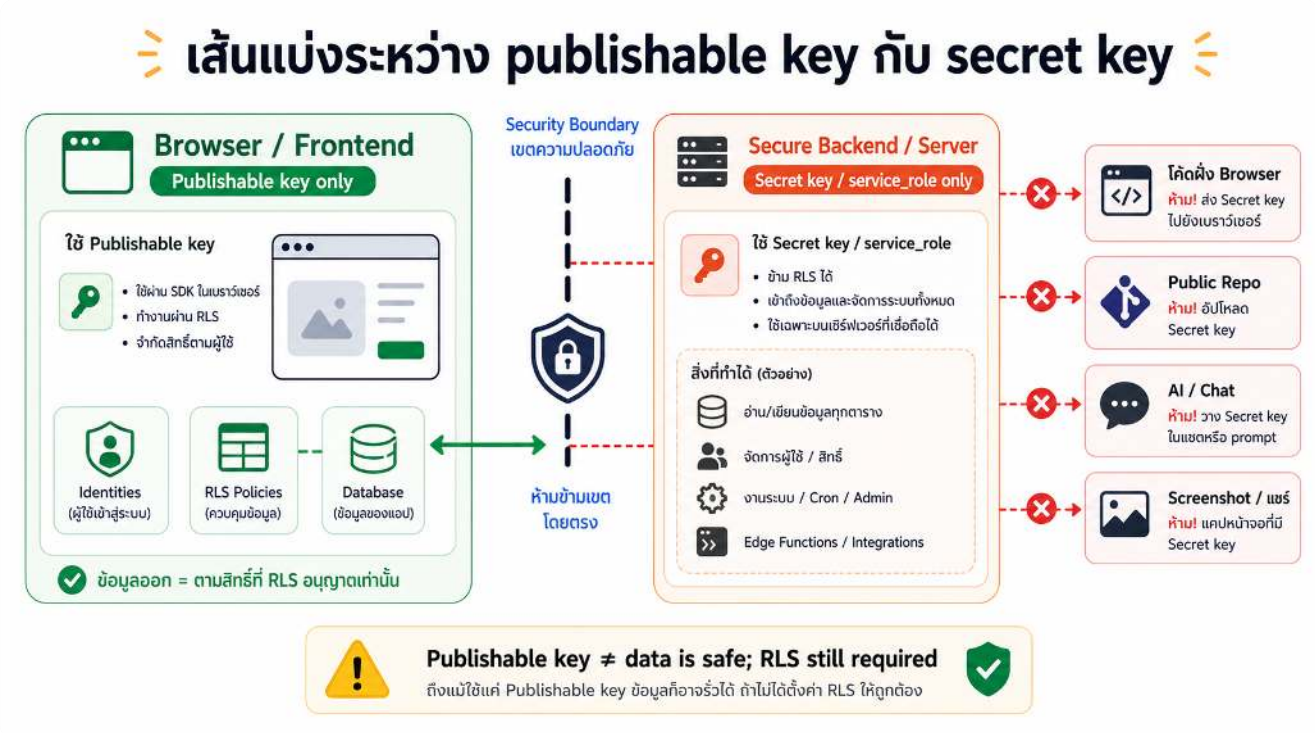
ยังไม่ใช้ auth

ยังไม่ใช้ storage

ยังไม่ใช้ function ซับซ้อน

ยิ่งใช้น้อย ยิ่งตรวจง่าย

API key มีหลายประเภท อย่าสับสน



เส้นแบ่งระหว่าง publishable key กับ secret key

publishable key ใช้ฝั่ง browser ได้ แต่ secret/service_role ต้องไม่อยู่ใน frontend หรือ public repo

Supabase official docs แยก key หลัก ๆ เป็น publishable key และ secret key

สำหรับ non-coder ให้จำแบบนี้:

Publishable key

ใช้กับส่วนที่อยู่ฝั่ง browser ได้

เช่นหน้าเว็บที่ user เปิด

มันไม่ได้เป็น “ความลับ” แบบ password

แต่การใช้ publishable key ต้องพึ่ง database policy/RLS ให้ถูก

Secret key / service role

ใช้เฉพาะฝั่ง server หรือ backend ที่ปลอดภัยกว่า

ห้ามเอาไปใส่ frontend

ห้าม commit ลง GitHub

ห้ามส่งใน chat แบบไม่ระวัง

ห้ามใช้ใน browser แม้จะเป็น localhost

ถ้าคุณไม่แน่ใจว่า key ไหนคืออะไร ให้หยุดและถามก่อน

> `Key นี้ควรอยู่ฝั่ง browser ได้ไหม หรือเป็น secret/server-only key?`
> `ถ้าเอาไปใส่ frontend จะเสี่ยงอะไร?`

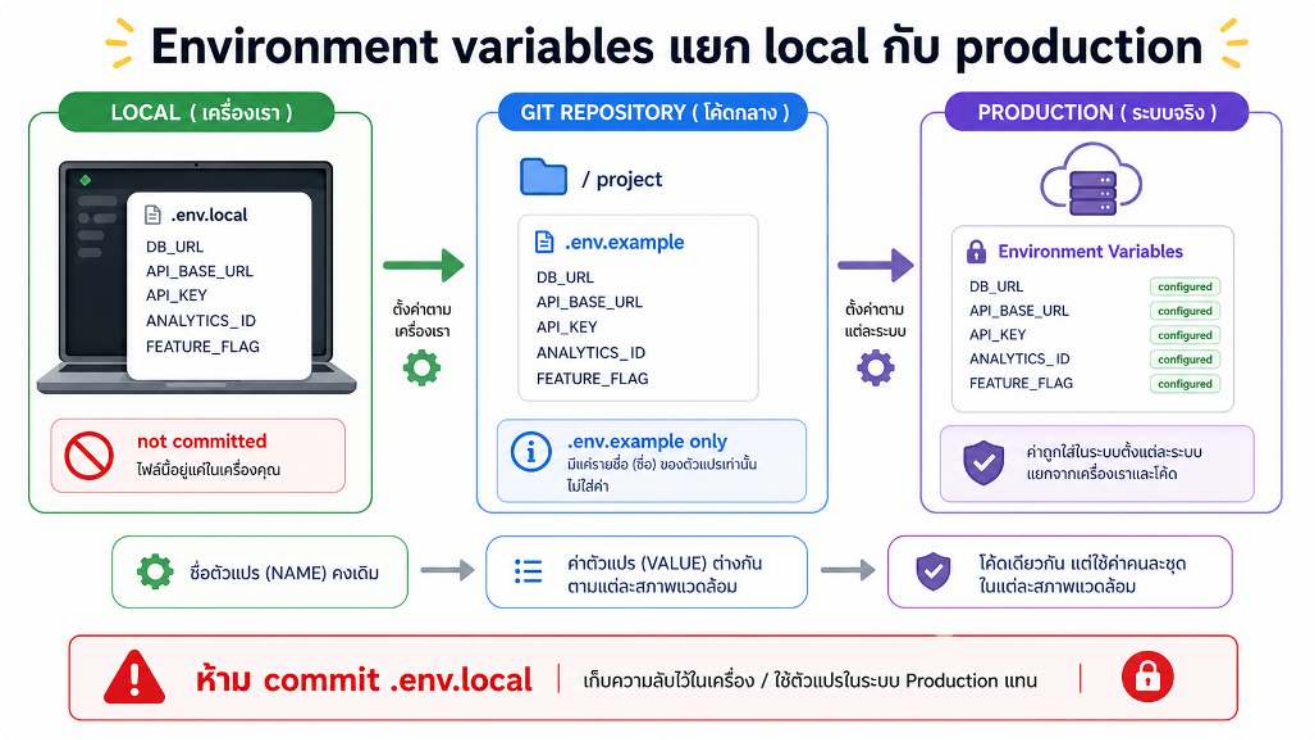
กฎง่าย ๆ:

> key ที่มีสิทธิ์สูง ห้ามอยู่ใน code ที่ user เปิดดูได้

Red line สำหรับ non-coder:

> `ถ้า Claude เสนอให้ใช้ secret key หรือ service_role เพื่อให้ form ทำงานง่ายขึ้น`
> `ให้หยุดทันที และถามหาวิธีที่ใช้ publishable key + RLS/policy ที่ถูกต้องแทน`

Environment variables คืออะไร



Environment variables แยก local กับ production

`.env.local` *อยู่ในเครื่อง ส่วน production ต้องตั้งค่าใน platform deploy และไม่ควร commit ค่า secret*

เวลา app ต้องใช้ค่าอย่าง Supabase URL หรือ key เรามักเก็บใน environment variables

ตัวอย่าง:

```
NEXT_PUBLIC_SUPABASE_URL=
NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY=
```

สำหรับ frontend บาง framework คำที่ขึ้นต้นด้วย `NEXT_PUBLIC_` หมายถึงค่าที่ถูกส่งไป browser ได้

ดังนั้นอย่าใส่ secret key ในตัวแปรที่ public

ไฟล์ที่มักใช้เก็บค่าเหล่านี้คือ:

```
.env.local
```

แต่ไฟล์นี้ไม่ควรถูก commit ขึ้น GitHub

ควรมีแค่ไฟล์ตัวอย่าง:

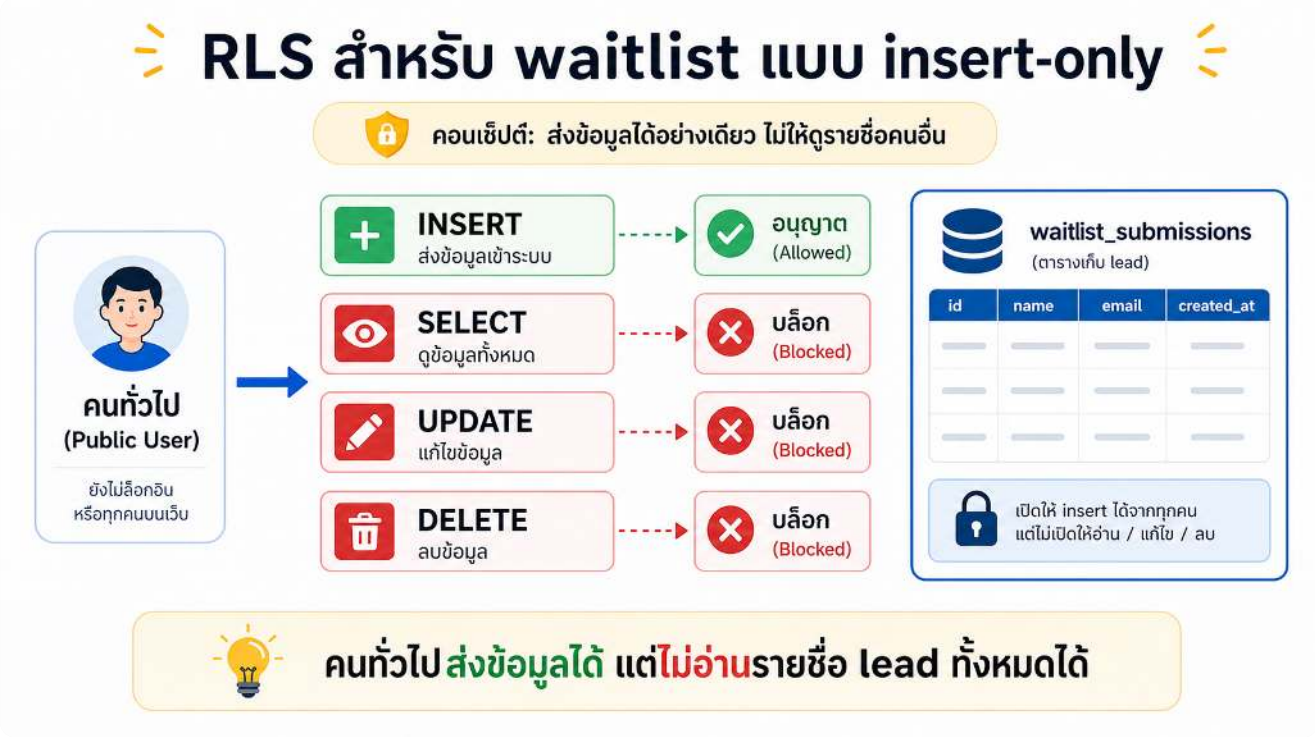
```
.env.example
```

ที่มีชื่อ variable แต่ไม่มีค่าจริง

ตัวอย่าง:

```
NEXT_PUBLIC_SUPABASE_URL=your-project-url
NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY=your-publishable-key
```

RLS คืออะไรแบบภาษาคน



RLS สำหรับ waitlist แบบ insert-only

สำหรับ waitlist แรก คนทั่วไปควร submit ได้ แต่ไม่ควรอ่าน แก้ หรือลบ lead ของคนอื่น

RLS ย่อมาจาก Row Level Security

ให้คิดว่าเป็นกติกาที่บอกว่า:

```
ใครอ่าน row ไหนได้
ใครเพิ่ม row ได้
ใครแก้ row ได้
ใครลบ row ได้
```

Supabase docs ระบุว่า RLS ควรเปิดบน table ที่อยู่ใน exposed schema เช่น `public`

สำหรับ waitlist form ที่ user ไม่ login เราอาจต้องการแค่:

```
anon insert ได้
anon select ไม่ได้
anon update ไม่ได้
anon delete ไม่ได้
```

แปลเป็นภาษาคน:

> คนทั่วไปส่งข้อมูลเข้ามาได้ แต่ไม่ควรอ่านรายชื่อ lead ทั้งหมดได้

นี่สำคัญมาก

ถ้าเผลอเปิด read public คุณอาจทำให้ใครก็ได้อ่านรายชื่อ lead ได้

ขอให้ Claude อธิบาย policy ก่อนสร้าง

อย่าให้ Claude สร้าง SQL policy แบบคุณไม่เข้าใจ

ใช้ prompt นี้:

```
ฉันต้องการสร้าง Supabase table สำหรับ waitlist
ผู้ใช้ทั่วไปควร submit form ได้ แต่ไม่ควรอ่าน แก้ หรือลบข้อมูลของคนอื่น

ช่วยเสนอ schema และ RLS policy ก่อน
กติกา:
- อย่าเพิ่งรัน SQL
- อธิบายเป็นภาษาคนว่า policy แต่ละอันอนุญาตอะไร
- แยกสิ่งที่จำเป็นกับ optional
- ถ้ามีความเสี่ยง ให้เตือน
```

ถ้าคุณอ่าน policy ไม่เข้าใจ ให้ถามต่อ:

> ช่วยอธิบาย `policy` นี้เหมือนอธิบายให้เจ้าของธุรกิจที่ไม่รู้ SQL
> ใครทำอะไรได้ และใครทำอะไรไม่ได้?

อย่า approve database change ที่คุณอธิบายเองไม่ได้เลย

Step 1: สร้าง table แบบเล็กที่สุด

Waitlist table schema ที่เล็กพอ

เริ่มจาก field เท่าที่จำเป็นก่อน ยิ่งเก็บน้อย ยิ่งตรวจและปกป้องง่าย

Table สำหรับ version แรกอาจมีแค่นี้:

```
waitlist_submissions
- id
- name
- email
- company
- main_problem
- created_at
```

ให้ Claude ช่วยเสนอ SQL ได้ แต่ให้มันอธิบายก่อน

Prompt:

```
ช่วยเสนอ SQL สำหรับ Supabase table ชื่อ waitlist_submissions
fields:
- id
- name required
- email required
- company optional
- main_problem optional
- created_at timestamp

ต้องการให้ user จากหน้าเว็บ insert ได้ แต่ไม่สามารถ select/update/delete rows ได้

กติกา:
- อย่าเพิ่งรัน SQL
- เสนอ SQL พร้อมคำอธิบายทีละส่วน
- อธิบาย RLS policy เป็นภาษาคน
- ชี้จุดที่ฉันควรตรวจใน Supabase dashboard
```

ถ้า SQL ดูยาวหรือซับซ้อนเกินไป ให้ถามว่า:

```
มีเวอร์ชันที่ง่ายและปลอดภัยพอสำหรับ waitlist version แรกไหม?
```

Step 2: ต่อ form กับ Supabase

หลัง table พร้อมแล้ว ค่อยให้ Claude Code ต่อ frontend

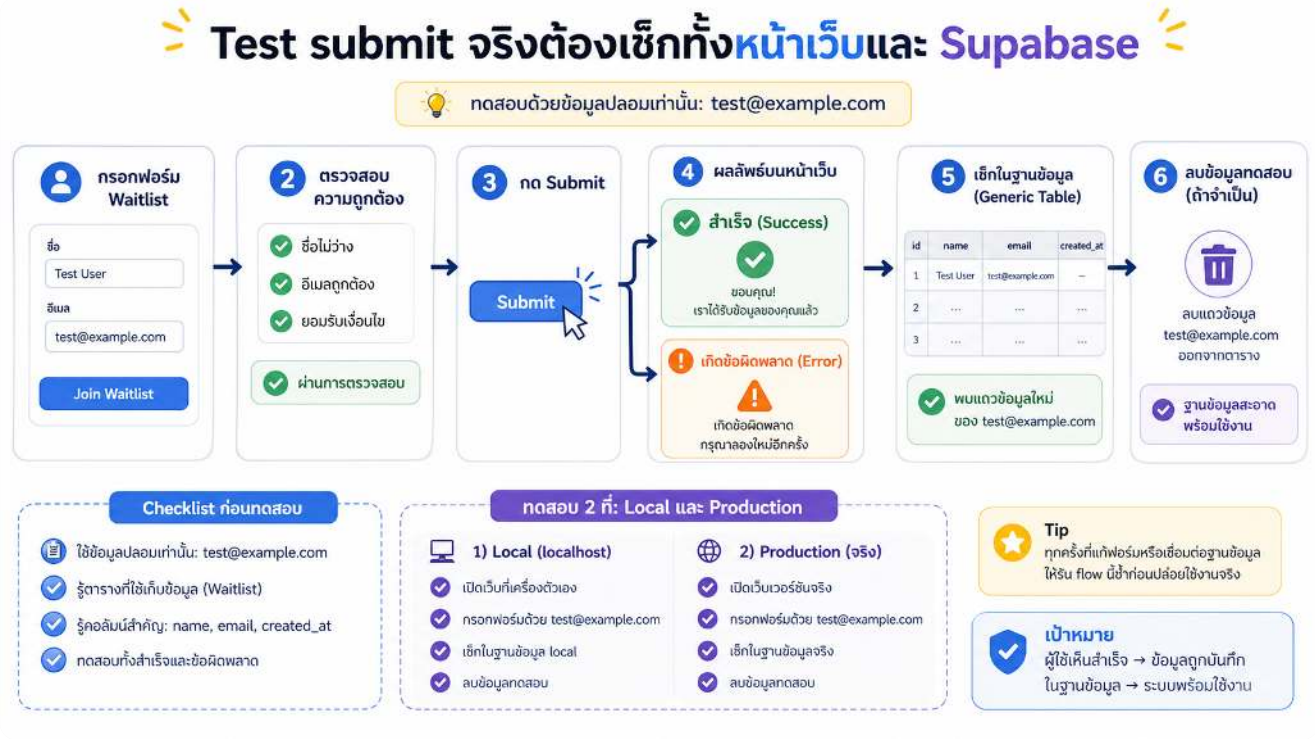
Prompt:

```
นี่คือ spec และ Supabase table ที่เตรียมไว้:
[paste spec/table summary]

ช่วยต่อ waitlist form ให้ submit ไป Supabase

กติกา:
- ใช้ publishable key เท่านั้นใน frontend
- ห้ามใช้ secret key หรือ service role key ใน browser
- ห้าม commit .env หรือ .env.local
- ถ้าต้องเพิ่ม package ใหม่ ให้ถามก่อน
- หลังทำเสร็จให้สรุปไฟล์ที่แก้และวิธี test
```

ถ้า Claude ขอ key ให้คุณอย่า paste secret ลง chat แบบไม่คิด

ให้ใส่ค่าใน `.env.local` เอง หรือให้ Claude บอกชื่อ variable ที่ต้องใช้

Step 3: Test submit จริง



Test submit จริงต้องเช็กทั้งหน้าเว็บและ Supabase

การ test waitlist ต้องเช็กทั้ง state บนหน้าเว็บและ row ที่เข้า database จริง

หลังต่อ form แล้ว ให้ test แบบนี้

Checklist:

- ☐ ไม่กรอก name แล้ว submit → เห็น error
- ☐ ไม่กรอก email แล้ว submit → เห็น error
- ☐ กรอก email ผิดรูปแบบ → เห็น error
- ☐ กรอกข้อมูลถูกต้อง → เห็น success message
- ☐ Supabase table มี row ใหม่
- ☐ refresh หน้าแล้ว form ยังใช้ได้
- ☐ error message ไม่โชว์ข้อมูล technical เกินไป
- ☐ browser ไม่เห็น secret key
- ☐ Git ไม่ track `.env.local`

ให้ Claude ช่วยสร้าง manual test plan:

```
ช่วยสร้าง manual test plan สำหรับ waitlist form นี้
แยกเป็น:
1. happy path
2. validation errors
3. Supabase/network error
4. security checks
5. สิ่งที่ต้องดูใน Supabase dashboard
```

Step 4: ตรวจว่า secret ไม่หลุด

ก่อน commit และ deploy ให้ตรวจเรื่อง secret

ถาม Claude:

```
ช่วยตรวจ project นี้ว่ามีโอกาส secret หลุดไหม
โฟกัสที่:
- .env หรือ .env.local ถูก track ไหม
- มี API key hard-code ใน source code ไหม
- มี Supabase secret/service_role key อยู่ใน frontend ไหม
- .env.example มีเฉพาะ placeholder ไหม

อย่าแสดงค่าของ secret ซ้ำในคำตอบ
ให้บอกแค่ว่าพบหรือไม่พบ และควรแก้อะไร
```

คำว่า “อย่าแสดงค่าของ secret ซ้ำ” สำคัญ

ถ้ามี secret โผล่ใน output คุณอาจทำให้มันกระจายไปอีกที่โดยไม่ตั้งใจ

GitHub มี secret scanning เพื่อช่วยตรวจ secret leaks แต่คุณไม่ควรพึ่งระบบ scan อย่างเดียว

นิสัยที่ดีที่สุดคืออย่า commit secret ตั้งแต่แรก

Step 5: Deploy แล้ว test production

ก่อน deploy จริง ให้เปิด **Appendix E — Supabase Waitlist Playbook** และ **Appendix D — Vercel Deploy Playbook** เพื่อเช็ก env var, RLS และ production test แบบไม่ข้ามขั้น

เมื่อทุกอย่างผ่านในเครื่องแล้ว ค่อย deploy

ถ้าใช้ Vercel ต้องใส่ environment variables ใน Vercel ด้วย

ไม่ใช่แค่ในเครื่องคุณ

Checklist ก่อน deploy:

- [ ] build ผ่านในเครื่อง
- [ ] `.env.local` ไม่ถูก commit
- [ ] Vercel มี Supabase URL และ publishable key
- [ ] RLS เปิดแล้ว
- [ ] anon insert ได้เท่าที่จำเป็น
- [ ] anon select/update/delete ไม่ได้
- [ ] form แสดง error ที่ user เข้าใจได้

หลัง deploy:

- [ ] เปิด production URL
- [ ] submit test lead
- [ ] ดู row ใน Supabase
- [ ] ลบ test data ถ้าไม่ต้องการเก็บ
- [ ] test mobile
- [ ] test validation

อย่าคิดว่าใช้ได้ในเครื่อง = ใช้ได้บน production

environment บน Vercel อาจต่างจากเครื่องคุณ

Step 6: Launch note สำหรับ waitlist

หลัง deploy ให้เขียน note สั้น ๆ

MARKDOWN

```
# Launch Note: Waitlist v1

## URL
[production URL]

## What shipped
- Waitlist form
- Supabase insert
- Success and error states

## Data collected
- name
- email
- company
- main_problem
- created_at

## Not included
- No login
- No payment
- No admin dashboard
- No email automation

## Security assumptions
- Uses publishable key in frontend
- Secret/service role key is not used in browser
- RLS enabled
- Public users can insert only

## Manual tests done
- required fields
- invalid email
- successful submit
- production submit
- mobile layout

## Known issues
- [ถ้ามี]
```

Launch note นี้ทำให้คุณรู้ว่า version นี้เก็บข้อมูลอะไรและยังไม่ทำอะไร

สำคัญมากเมื่อ project เริ่มมี data จริง

Prompt รวมท้ายบท

ใช้ prompt นี้เมื่อต้องการให้ Claude Code ทำ waitlist app

```
ฉันต้องการเพิ่ม waitlist form ที่เก็บข้อมูลลง Supabase

Spec:
[paste spec]

กติกาสำคัญ:
- ทำเฉพาะ version 1
- ไม่มี login, payment, admin dashboard, email automation หรือ file upload
- เก็บข้อมูลเท่าที่จำเป็นเท่านั้น
- ใช้ publishable key ใน frontend เท่านั้น
- ห้ามใช้ secret key หรือ service_role key ใน browser
- ห้าม commit .env หรือ .env.local
- ถ้าต้องสร้าง/แก้ RLS policy ให้เสนอและอธิบายก่อน อย่ารันเองทันที
- ก่อนแก้ไฟล์ ให้เสนอ implementation plan
- หลังแก้ไฟล์ ให้สรุป diff และ manual test plan

Acceptance criteria:
- Submit สำเร็จแล้ว row เข้า Supabase
- Required fields มี validation
- Error state เข้าใจง่าย
- RLS เปิดและ public user อ่านข้อมูล lead ทั้งหมดไม่ได้
- ไม่มี secret หลุดใน source code
```

หลังทำเสร็จ ใช้ prompt review:

```
ช่วย review waitlist implementation นี้ก่อน deploy
ตรวจเฉพาะสิ่งสำคัญ:
1. ตรง spec ไหม
2. มี feature เกิน scope ไหม
3. เก็บข้อมูลเกินจำเป็นไหม
4. ใช้ key ถูกประเภทไหม
5. มี secret หลุดไหม
6. RLS/policy อนุญาตอะไรบ้าง อธิบายเป็นภาษาคน
7. manual test ก่อน deploy ต้องทำอะไร
8. มี risk อะไรที่ควรรู้ก่อนเปิดให้คนจริงใช้

อย่าแก้ไฟล์ก่อน รายงานก่อน
```

สรุป

Waitlist app คือก้าวแรกจากหน้าเว็บธรรมดาไปสู่ app ที่มี data จริง

นั่นทำให้มันมีประโยชน์ขึ้น แต่ก็เสี่ยงขึ้น

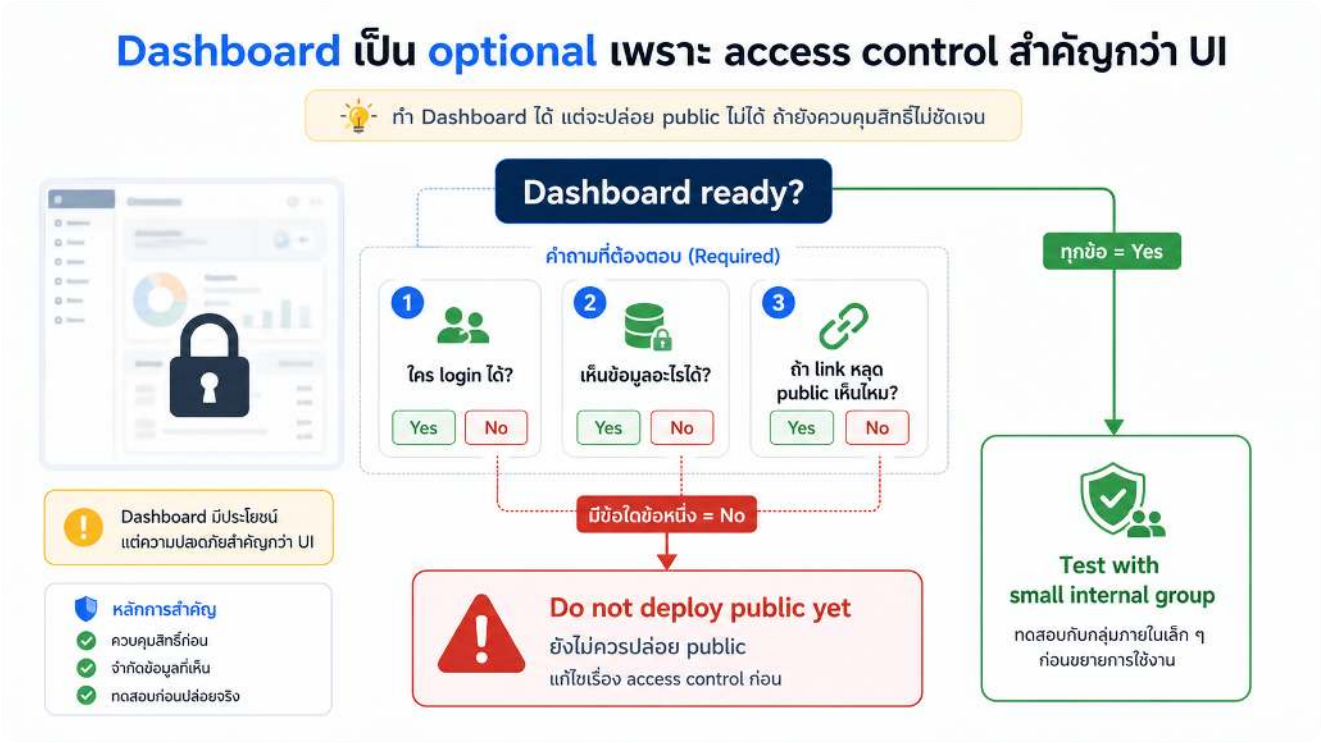
จำ 6 ข้อนี้:

1. เก็บข้อมูลให้น้อยที่สุด
2. แยก publishable key กับ secret key ให้ชัด
3. อย่า commit `.env` หรือ `.env.local`
4. เปิด RLS และให้สิทธิ์เท่าที่จำเป็น
5. Test ทั้ง local และ production
6. เขียน launch note ว่าเก็บอะไรและไม่ทำอะไร

บทต่อไปเป็น optional/bonus: เราจะต่อยอดจากข้อมูลที่เก็บได้ไปเป็น internal dashboard ขนาดเล็ก ถ้าคุณยังไม่มั่นใจเรื่อง auth, permission หรือ RLS ให้ข้ามบทนั้นชั่วคราวได้

Project 3: Internal Dashboard ขนาดเล็ก

แก่นของบทนี้



Dashboard เป็น optional เพราะ access control สำคัญกว่า UI

ถ้าตอบไม่ได้ว่าใคร login ได้และเห็นข้อมูลอะไรได้ ให้ยังไม่ deploy dashboard public

หลังจากมี waitlist form แล้ว คำถามต่อไปคือ:

> เราจะดู `lead` ที่เข้ามาได้ยังไง

คำตอบที่น่าทำคือ internal dashboard

แต่คำตอบที่อันตรายคือทำ dashboard ใหญ่เกินไปเร็วเกินไป

บทนี้เราจะสร้าง dashboard ขนาดเล็กที่เน้น **ดูข้อมูลและจัดการเบื้องต้น** เท่านั้น

ไม่ใช่ระบบ CRM เต็มรูปแบบ

ไม่ใช่ admin panel ครอบจักรวาล

เป้าหมายคือ:

```
ดู lead
ค้นหา/กรองเบื้องต้น
อัปเดตสถานะง่าย ๆ
ไม่เปิดข้อมูลให้คนที่ไม่ควรเห็น
```

นี่เป็นจุดที่ต้องคิดเรื่อง access และ permission จริงจังขึ้น

เพราะ dashboard คือหน้าที่แสดงข้อมูลที่ user กรอกเข้ามา

บทนี้เป็น **optional / bonus project** ไม่ใช่ทางบังคับของหนังสือ ถ้าคุณยังไม่มั่นใจเรื่อง login, permission, RLS หรือ secret key ให้ข้ามบทนี้ชั่วคราว แล้วใช้ Supabase dashboard ดูข้อมูลไปก่อน

กฎตัดสินใจง่าย ๆ:

```
Landing page public ได้ง่ายกว่า
Waitlist form public ได้ถ้า RLS ถูก
Internal dashboard ห้าม public ถ้ายังไม่มั่นใจเรื่อง access control
```

ทำไม internal dashboard ถึงน่าทำ

Internal tool เป็น use case ที่ดีมากสำหรับ non-coder

เพราะมันแก้ปัญหาใกล้ตัว เช่น:

- ดู lead ที่เข้ามา
- จัดลำดับคนที่ควร follow-up
- track status ว่าติดต่อแล้วหรือยัง
- ดูปัญหาที่ลูกค้าสนใจซ้ำ ๆ
- export insight ไปวางแผน content หรือ sales

แต่ข้อควรระวังคือ:

> internal ไม่ได้แปลว่าปลอดภัยโดยอัตโนมัติ

ถ้า URL หลุด หรือ permission เปิดผิด คนอื่นอาจเห็นข้อมูล lead ได้

ดังนั้น dashboard แรกต้องเล็กและถูกจำกัดการเข้าถึง

Project ที่เราจะสร้าง

เราจะต่อยอดจาก table `waitlist_submissions`

Dashboard version แรกมี:

- หน้า `/admin` หรือ `/dashboard`
- list รายการ lead
- field ที่แสดงเท่าที่จำเป็น
- search/filter ง่าย ๆ
- status เช่น `new`, `contacted`, `qualified`, `not_fit`
- manual refresh หรือ reload ได้

ยังไม่ทำ:

- role หลายระดับ
- team management
- audit log เต็มระบบ
- CRM sync
- bulk action
- email automation
- delete lead ผ่าน dashboard
- public sharing

ถ้า version แรกยังไม่มี login ที่แข็งแรงพอ ให้ dashboard นี้ยังไม่ควร deploy public

หรือควรจำกัดด้วยวิธีอื่นที่คุณเข้าใจและตรวจได้

Spec สำหรับ Dashboard version แรก

MARKDOWN

```
# Spec: Waitlist Internal Dashboard v1

## Goal
สร้าง internal dashboard สำหรับดู waitlist submissions และติดตามสถานะ follow-up เบื้องต้น

## Target user
เจ้าของ project หรือทีมเล็ก ๆ ที่ต้อง follow-up lead

## Version 1 scope
- แสดงรายการ waitlist submissions
- แสดง name, email, company, main_problem, status, created_at
- ค้นหาด้วย email/company/main_problem ได้แบบง่าย
- เปลี่ยน status ได้
- แสดง empty state และ error state

## Out of scope
- No public access
- No multi-role permission
- No bulk delete
- No CRM integration
- No email automation
- No payment
- No file upload

## Data fields
Existing:
- name
- email
- company
- main_problem
- created_at

New:
- status: new/contacted/qualified/not_fit
- notes: optional internal note

## Acceptance criteria
- Dashboard ไม่ควรเปิดให้ public user เห็นข้อมูล lead
- แสดงรายการ lead ได้ถูกต้อง
- Search/filter ใช้ได้เบื้องต้น
- เปลี่ยน status ได้
- ไม่แสดง secret หรือ key ใน browser
- มี manual test plan ก่อน deploy
- Claude อธิบาย permission/RLS ที่เกี่ยวข้องเป็นภาษาคน
```

สังเกตว่า acceptance criteria ข้อแรกคือเรื่อง access

เพราะ dashboard ที่ทำงานได้แต่เปิดให้คนทั้งโลกเห็นข้อมูล คือความล้มเหลว ไม่ใช่ความสำเร็จ

Authentication vs Authorization แบบง่าย

Supabase docs แยกสองคำนี้ชัด:

```
Authentication = คนนี้คือใคร
Authorization = คนนี้มีสิทธิ์ทำอะไร
```

ตัวอย่าง:

```
Authentication: คุณ login แล้ว เป็น krish@example.com
Authorization: คุณมีสิทธิ์ดู dashboard นี้ไหม
```

หลายคนทำผิดตรงนี้

เขาคิดว่า “มี login แล้ว” แปลว่าปลอดภัย

แต่จริง ๆ ต้องถามต่อ:

```
login แล้วเห็นข้อมูลอะไรได้บ้าง
ใคร update status ได้
ใคร delete ได้
ถ้าไม่ login จะเห็นอะไรไหม
```

สำหรับ dashboard แรก ให้ทำสิทธิ์ให้น้อยที่สุด

ถ้ายังไม่แน่ใจเรื่อง auth/permission ให้ทำ dashboard สำหรับ local-only หรือยังไม่ deploy public

---

อย่าใช้ service role key ใน browser

Dashboard มักทำให้คนเผลอใช้ key สิทธิ์สูง

เพราะอยากอ่านข้อมูลทั้งหมดง่าย ๆ

นี่อันตรายมาก

กฎเดิมยังใช้เหมือนบทที่แล้ว:

secret key / service role key ห้ามอยู่ใน browser

ถ้าคุณต้องอ่านข้อมูล lead ทั้งหมด ต้องมีวิธีที่ปลอดภัยกว่า เช่น server-side route, auth, หรือ policy ที่จำกัดเฉพาะ user ที่ควรเห็น

สำหรับ non-coder ถ้า Claude เสนอให้ใส่ service role key ใน frontend ให้หยุดทันที

Prompt ที่ควรถาม:

> วิธีนี้ใช้ secret หรือ service role key ใน browser ไหม
> ถ้าใช่ ห้ามทำ
> ช่วยเสนอวิธีที่ปลอดภัยกว่า และอธิบายเป็นภาษาคน

RLS สำหรับ dashboard ต้องคิดใหม่

Permission table ก่อนสร้าง dashboard

ก่อนเขียน policy ให้ทำตาราง permission ว่า public, authenticated และ admin ทำอะไรได้บ้าง

ในบท waitlist เราอยากให้ public user insert ได้ แต่ไม่อ่านข้อมูลทั้งหมดได้

สำหรับ dashboard เราต้องมีผู้ใช้บางคนอ่านข้อมูลได้

ดังนั้น policy ต้องชัดกว่าเดิม

คำถามที่ต้องตอบก่อน:

```
ใครคือ admin
admin login ยังไง
admin อ่าน rows ทั้งหมดได้ไหม
admin update status ได้ไหม
public user อ่านอะไรได้ไหม
public user update อะไรได้ไหม
```

ถ้าคุณตอบไม่ได้ อย่าเพิ่ง deploy dashboard public

ให้ Claude ช่วยแปล permission เป็นตารางก่อน

```
ช่วยทำ permission table สำหรับ dashboard นี้
แถวคือ role: public, authenticated user, admin
คอลัมน์คือ select, insert, update, delete
ใส่ allowed/denied และอธิบายเหตุผลแบบ non-coder
อย่าเพิ่งเขียน SQL
```

ตัวอย่างที่ควรได้:

Role	Select	Insert	Update	Delete
public/anon	deny	allow เฉพาะ form	deny	deny
admin	allow	optional	allow status/notes	deny ใน v1

ตารางแบบนี้ช่วยให้คุณเข้าใจก่อนเห็น SQL

Step 1: เพิ่ม field ให้น้อยที่สุด

Dashboard version แรกอาจต้องเพิ่มแค่ 2 field:

```
status
notes
```

อย่าเพิ่ม field เยอะตั้งแต่แรก

เช่น:

```
lead_score
owner
utm_source
last_contacted_at
deal_value
pipeline_stage
```

บางอย่างมีประโยชน์ แต่ยังไม่จำเป็น

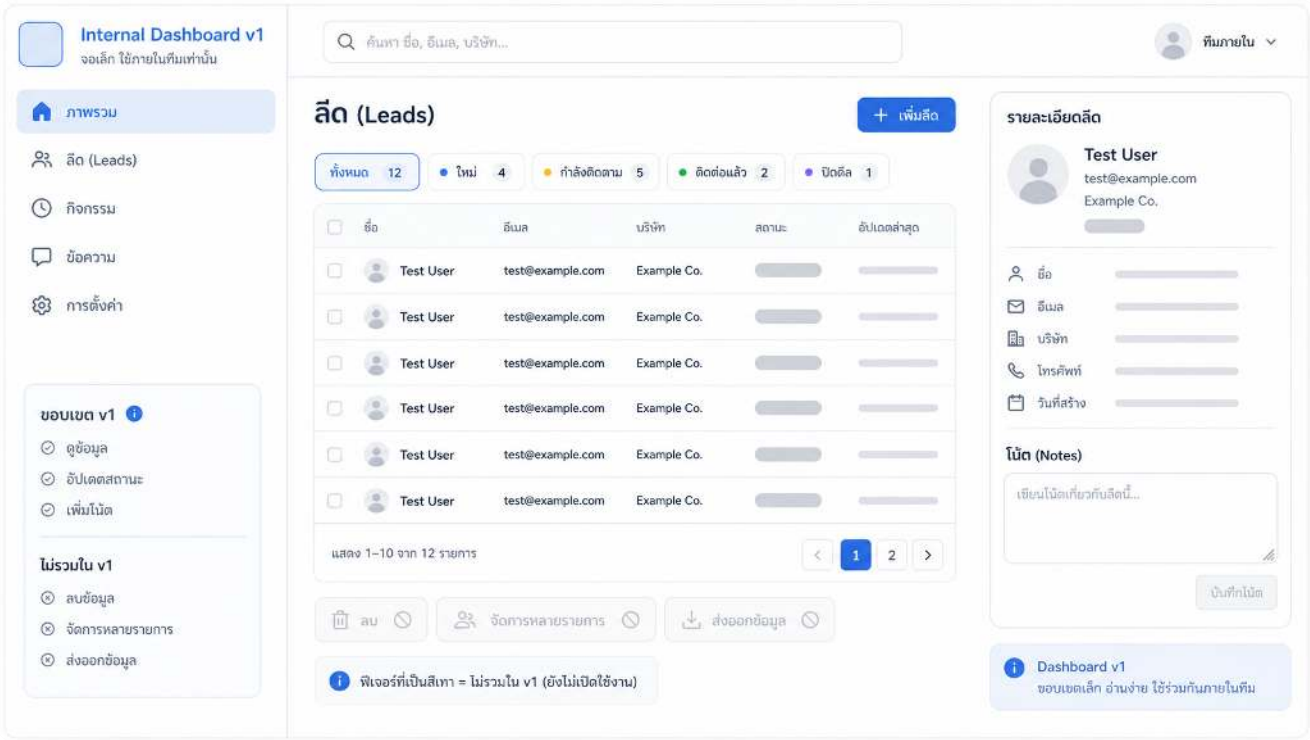
ให้เริ่มจาก:

```
status = ตอนนี้ lead อยู่สถานะไหน
notes = ข้อความสั้น ๆ สำหรับทีม
```

Prompt:

```
ช่วยเสนอ migration หรือ SQL สำหรับเพิ่ม field status และ notes ใน
waitlist_submissions
กติกา:
- อย่าเพิ่งรัน SQL
- status ควรมี default เป็น new
- notes เป็น optional
- อธิบายว่าการเปลี่ยนนี้กระทบข้อมูลเดิมไหม
- เสนอ manual test หลังเปลี่ยน schema
```

Step 2: ออกแบบ dashboard UI แบบเรียบง่าย



Dashboard v1 ควรเล็กและอ่านง่าย

dashboard แรกควรเน้นดู lead, search/filter และ update status เท่านั้น ยังไม่ต้องมี bulk/export/delete

Dashboard แรกไม่ต้องสวยมาก

ต้องใช้งานได้และไม่สับสน

ควรมี:

- title ชัด ๆ
- จำนวน lead ทั้งหมด
- filter/search
- table หรือ card list
- status badge
- ปุ่มเปลี่ยน status
- empty state
- error state

ไม่ควรมี:

- chart เยอะ ๆ
- animation หนัก ๆ
- export public link
- delete button ถ้ายังไม่จำเป็น
- bulk action ถ้ายังไม่เข้าใจ risk

Prompt:

```
ช่วยออกแบบ dashboard UI สำหรับ waitlist submissions v1
เน้นเรียบง่าย ใช้ได้จริง ไม่ต้องสวยอลังการ

ต้องมี:
- list/table ของ lead
- search/filter เบื้องต้น
- status badge
- เปลี่ยน status ได้
- empty state
- error state

ยังไม่ต้องมี:
- chart
- export
- delete
- bulk action
- CRM integration

อย่าเพิ่งแก้ไฟล์ ให้เสนอ layout และ user flow ก่อน
```

Step 3: ให้ Claude build แบบถามก่อนเรื่อง auth/permission

ก่อน build dashboard ต้องมี rule ชัด:

```
ถ้าเกี่ยวกับ auth, RLS, secret, service role, env ต้องถามก่อน
```

Prompt:

```
ฉันต้องการสร้าง internal dashboard v1 จาก spec นี้:
[paste spec]

กติกา:
- อย่าใช้ service role/secret key ใน browser
- ถ้าต้องแก้ RLS policy ให้เสนอและอธิบายก่อน อย่ารันเอง
- ถ้าต้องเพิ่ม auth ให้เสนอทางเลือกก่อน
- ห้ามเพิ่ม delete/bulk action/export ใน v1
- ก่อนแก้ไฟล์ ให้เสนอ implementation plan
- หลังแก้ไฟล์ ให้สรุป diff, permission assumptions, และ manual test plan
```

ถ้า Claude บอกว่า “ทำได้ง่าย ๆ ด้วย service role key ใน frontend” ให้หยุด

นั่นไม่ใช่ง่าย นั่นคือเสี่ยง

Step 4: Manual QA สำหรับ dashboard

Dashboard ต้อง test มากกว่า landing page

เพราะมีข้อมูลจริงและ permission

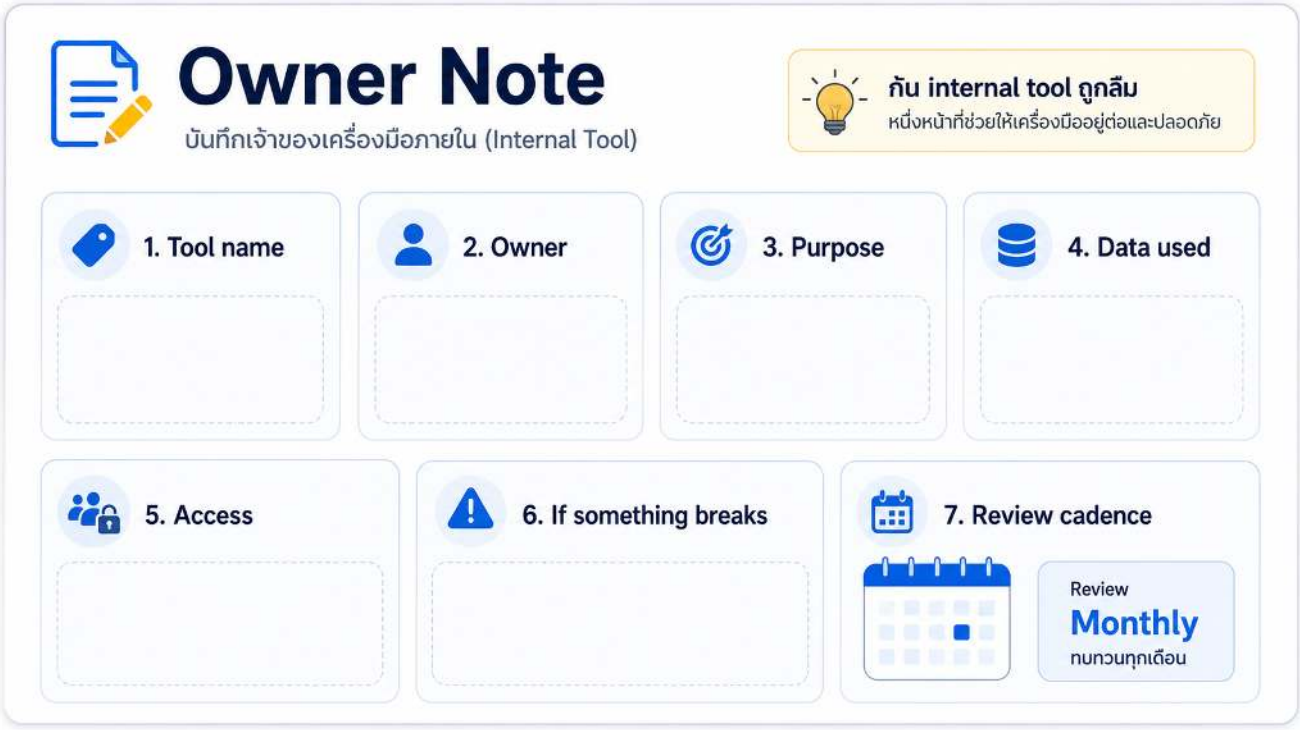
Checklist:

- ☐ หน้า dashboard ไม่ควรถูก public user เข้าถึง
- ☐ ถ้าไม่ login หรือไม่มีสิทธิ์ ควรเข้าไม่ได้
- ☐ admin เห็น list lead ได้
- ☐ search/filter ใช้ได้
- ☐ เปลี่ยน status ได้
- ☐ เปลี่ยน status แล้ว refresh ยังอยู่
- ☐ notes ไม่บังคับกรอก
- ☐ error state แสดงเมื่อโหลดข้อมูลไม่ได้
- ☐ empty state แสดงเมื่อไม่มีข้อมูล
- ☐ ไม่มี secret key อยู่ใน browser/source
- ☐ public user ไม่สามารถ select lead ทั้งหมดผ่าน API ได้
- ☐ delete ไม่มีใน v1 หรือถูกปิดไว้

ให้ Claude ช่วยสร้าง test plan:

```
ช่วยสร้าง manual QA checklist สำหรับ internal dashboard นี้
แยกเป็น:
1. access control
2. data display
3. search/filter
4. status update
5. error/empty states
6. security checks
7. production checks
```

Step 5: อย่าลืม ownership



Owner note กัน internal tool ถูกลืม

internal tool ต้องมี owner, purpose, access rule และ review cadence ไม่อย่างนั้นจะกลายเป็นของที่ไม่มีใครดูแล

OWASP พูดถึง asset management failure ใน citizen development: app ที่สร้างง่ายอาจถูกลืม ถูกทิ้ง หรือไม่มีเจ้าของดูแล

Internal dashboard เป็นตัวอย่างที่เจอได้ง่าย

คุณสร้างไว้ใช้เอง
ทีมเริ่มใช้
เวลาผ่านไปทุกคนพึ่งมัน
แต่ไม่มีใครเป็นเจ้าของ
ไม่มีใครดู dependency
ไม่มีใครรู้ว่าปิดได้ไหม

ดังนั้นทุก internal tool ควรมี owner note

```markdown
# Owner Note

## Tool name
Waitlist Internal Dashboard

## Owner
[ชื่อ/ทีม]

## Purpose
ดูและ follow-up waitlist submissions

## Data used
- name
- email
- company
- main_problem
- status
- notes

## Access
[ใครเข้าได้]

## If something breaks
[ใครรับผิดชอบ / rollback ยังไง]

## Review cadence
ตรวจทุกเดือนว่า tool นี้ยังใช้ไหม และ access ยังถูกไหม
```

ไม่ต้องยาว

แต่ต้องมีเจ้าของ

Step 6: Deploy หรือยังไม่ deploy?

สำหรับ dashboard คำถามนี้สำคัญกว่า landing page

ถ้าคุณตอบคำถาม 3 ข้อนี้ไม่ได้ ให้ยังไม่ deploy:

```
ใคร login ได้?
คนที่ login แล้วเห็นข้อมูลอะไรได้?
ถ้า link หลุด คนทั่วไปเห็นข้อมูล lead ได้ไหม?
```

ถ้า dashboard ยังไม่มี auth/access control ที่คุณมั่นใจ อย่า deploy public

ทางเลือก:

1. ใช้เฉพาะ local ก่อน
2. deploy แต่ป้องกันด้วย auth ที่เข้าใจและ test แล้ว
3. ให้ developer ช่วย review ก่อนเปิดใช้จริง
4. ยังไม่ทำ dashboard ใช้ Supabase dashboard ดูข้อมูลก่อนชั่วคราว

อย่า deploy เพราะ “AI ทำเสร็จแล้ว”

deploy เพราะคุณตรวจแล้วว่า risk อยู่ในระดับรับได้

Prompt รวมท้ายบท

ใช้ prompt นี้สำหรับ dashboard

```
ฉันต้องการสร้าง internal dashboard v1 สำหรับ waitlist submissions

Spec:
[paste spec]

กติกา:
- Dashboard ต้องไม่ public โดยไม่มี access control
- ห้ามใช้ service role/secret key ใน browser
- ห้ามเพิ่ม delete, bulk action, export, CRM integration หรือ email
automation ใน v1
- ถ้าต้องแก้ RLS/auth/env ให้เสนอและอธิบายก่อน อย่ารันเอง
- ก่อนแก้ไฟล์ ให้เสนอ implementation plan
- หลังแก้ไฟล์ ให้สรุป diff, permission assumptions, และ manual QA checklist

Output ก่อนเริ่ม build:
1. implementation plan
2. files likely to change
3. permission table: public / authenticated / admin
4. risks
5. questions before build
```

หลัง build ใช้ prompt review:

```
ช่วย review internal dashboard นี้ก่อน deploy
ตรวจเฉพาะสิ่งสำคัญ:
1. ตรง spec ไหม
2. มี feature เกิน scope ไหม
3. access control ทำงานยังไง อธิบายเป็นภาษาคน
4. public user อ่านข้อมูล lead ได้ไหม
5. มี secret/service role key ใน browser ไหม
6. RLS policy อนุญาตอะไรบ้าง
7. manual QA ที่ต้องทำก่อนเปิดใช้จริงคืออะไร
8. ถ้ายังไม่ควร deploy ให้บอกตรง ๆ ว่าทำไม

อย่าแก้ไฟล์ก่อน รายงานก่อน
```

สรุป

Internal dashboard มีประโยชน์มาก แต่เสี่ยงกว่า landing page และ waitlist form

เพราะมันแสดงข้อมูลที่ user ส่งมา

จำ 6 ข้อนี้:

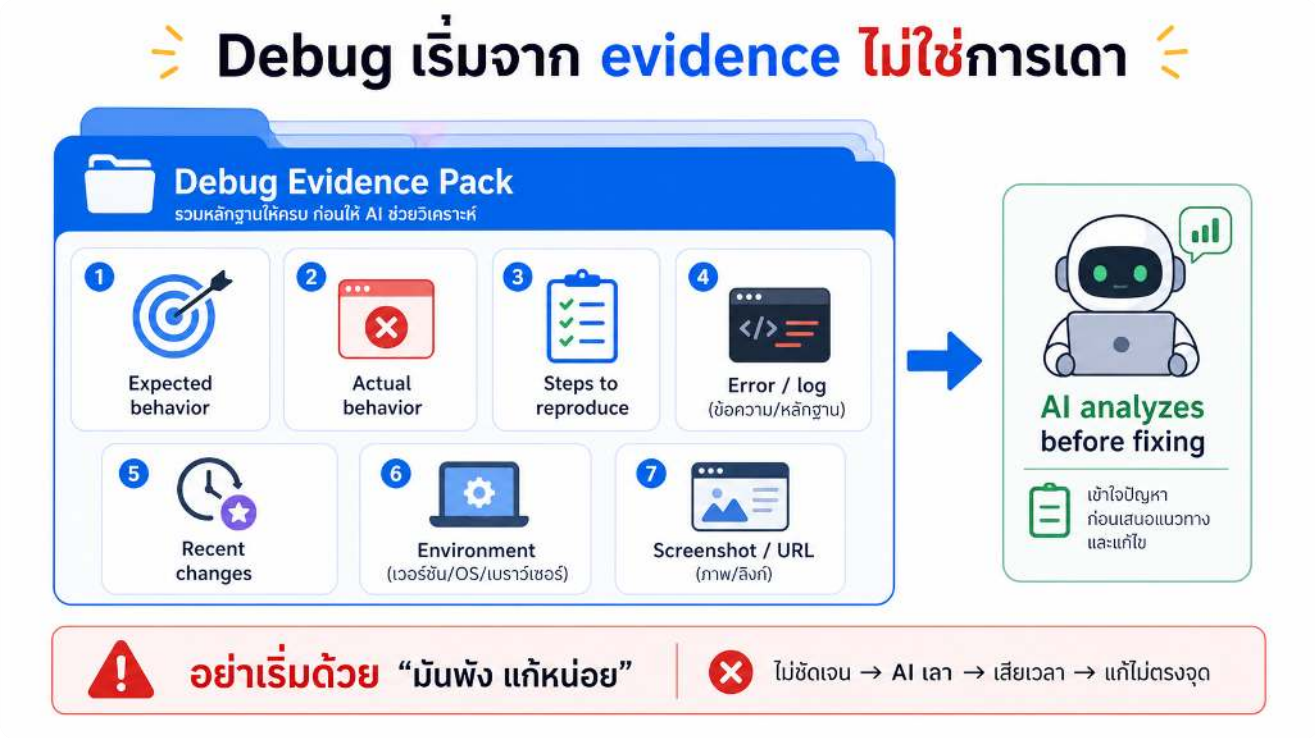
1. dashboard แรกควรเล็ก
2. อย่าเปิดข้อมูล lead ให้ public
3. อย่าใช้ service role/secret key ใน browser
4. ทำ permission table ก่อนเขียน policy
5. มี manual QA เรื่อง access control
6. ต้องมี owner note ไม่ให้ tool กลายเป็นของถูกลืม

บทต่อไป เราจะพูดเรื่อง debug แบบ non-coder: เวลา build พัง, deploy fail, form submit ไม่ได้ หรือ dashboard error ต้องเก็บหลักฐานยังไง แล้วให้ Claude Code แก้แบบไม่มั่ว

$ บทที่ 9

Debug แบบคนไม่ใช่ programmer

แก่นของบทนี้



Debug เริ่มจาก evidence ไม่ใช่การเดา

ก่อนให้ Claude แก้ bug ให้เก็บ expected, actual, steps, log, recent changes และ environment ก่อน

เวลาใช้ Claude Code สร้างเว็บหรือแอป คุณจะเจอ error แน่นอน

Error ไม่ได้แปลว่าคุณล้มเหลว

Error คือข้อมูล

ปัญหาของ non-coder ไม่ใช่ “อ่าน error ไม่ออก” อย่างเดียว

แต่คือไม่รู้ว่าจะเก็บข้อมูลอะไรให้ AI ช่วยแก้ได้แม่น

บทนี้จะสอน workflow ง่าย ๆ:

```
หยุด → เก็บหลักฐาน → อธิบาย expected behavior → ให้ Claude วิเคราะห์ก่อนแก้ → แก้ทีละจุด → test ซ้ำ
```

ถ้าคุณทำตามนี้ คุณไม่ต้องอ่าน code เก่ง ก็ debug ได้เป็นระบบขึ้นมาก

ถ้าเจอปัญหาจริงระหว่างทำ project ให้เปิด **Appendix F — Debug Evidence Playbook** คู่กับบทนี้ เพราะ appendix นั้นช่วยเก็บหลักฐานเป็นชุดก่อนส่งให้ Claude แก้

อย่าเริ่มด้วย “มันพัง แก้หน่อย”

คำสั่งนี้กว้างเกินไป:

```
มันพัง แก้ให้หน่อย
```

Claude อาจต้องเดาว่า:

```
พังตรงไหน
เกิดหลังทำอะไร
error message คืออะไร
expected behavior คืออะไร
เพิ่งแก้อะไรไป
รัน command อะไร
พังในเครื่องหรือ production
```

ยิ่งข้อมูลน้อย AI ยิ่งเดาเยอะ

และเมื่อ AI เดา มันอาจแก้ผิดจุด หรือเปลี่ยนหลายไฟล์โดยไม่จำเป็น

Debug ที่ดีเริ่มจากหลักฐาน ไม่ใช่อารมณ์

Error 5 แบบที่คุณจะเจอบ่อย

1. Install error

เกิดตอนติดตั้ง package หรือ dependency

ตัวอย่าง:

```
npm install แล้ว fail
package version conflict
permission denied
```

2. Build error

เกิดตอน project compile/build ไม่ผ่าน

ตัวอย่าง:

```
npm run build แล้ว error
TypeScript error
missing import
```

3. Runtime error

เว็บเปิดได้ แต่ใช้งานแล้วพัง

ตัวอย่าง:

```
กด submit แล้ว error
หน้า blank
console มี error
```

4. Data/API error

เกี่ยวกับ Supabase, API, env, permission

ตัวอย่าง:

```
insert ไม่เข้า database
RLS block
ใช้ env var ไม่เจอ
401/403/500
```

5. Deploy error

ใช้ได้ในเครื่อง แต่ deploy แล้วพัง

ตัวอย่าง:

```
Vercel build fail
production URL เปิดไม่ได้
environment variable ไม่ถูกตั้งค่า
```

แค่แยก error ได้ว่าอยู่กลุ่มไหน คุณก็ช่วย Claude ได้เยอะแล้ว

Debug report template

เวลาเจอปัญหา ให้ copy template นี้ไปใช้

```
ฉันเจอปัญหานี้ ช่วยวิเคราะห์ก่อนแก้

## What happened
[เกิดอะไรขึ้น]

## Expected behavior
[ควรเกิดอะไร]

## Steps to reproduce
1. [ขั้นตอนที่ 1]
2. [ขั้นตอนที่ 2]
3. [ขั้นตอนที่ 3]

## Error message / log
[paste error/log เต็ม ๆ]

## Recent changes
[เพิ่งให้ Claude แก้อะไร หรือเพิ่งเปลี่ยนอะไร]

## Environment
- Local or production:
- Command used:
- URL/page:

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อธิบายสาเหตุที่เป็นไปได้ 2-3 แบบก่อน
- บอกว่าจะตรวจอะไรเพื่อยืนยัน
- เสนอวิธีแก้ที่เปลี่ยนน้อยที่สุด
```

Template นี้สำคัญมาก

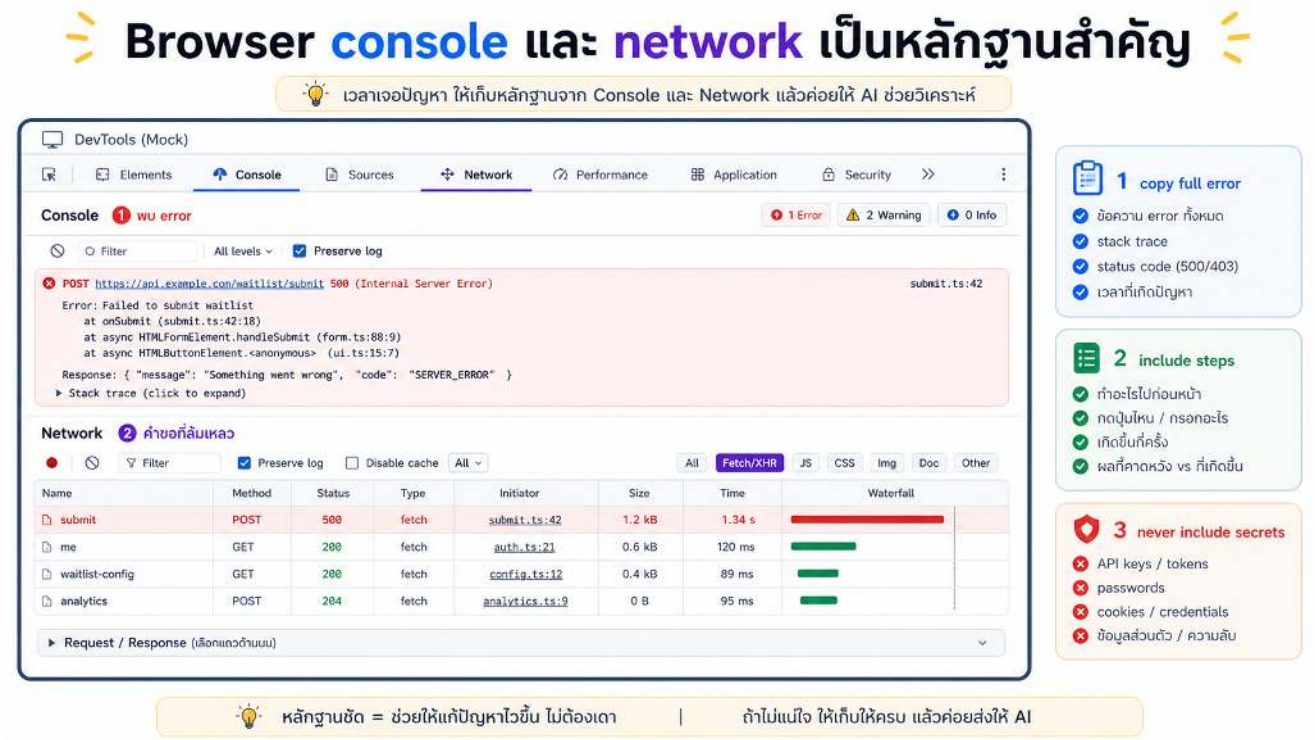
มันเปลี่ยนคุณจาก “คนบอกว่าแอปพัง” เป็น “คนให้ข้อมูล debug”

Evidence pack ขั้นต่ำก่อนให้ Claude แก้:

1. ทำอะไรอยู่
2. คาดหวังว่าจะเกิดอะไร
3. เกิดอะไรจริง
4. error/log เต็ม ๆ
5. เพิ่งเปลี่ยนอะไรล่าสุด
6. local หรือ production
7. screenshot/URL ถ้ามี

ถ้ามีไม่ครบ ให้เก็บให้ครบก่อน ค่อยขอให้ Claude แก้

Copy error ให้ครบ



Browser console และ network เป็นหลักฐานสำคัญ

เมื่อหน้าเว็บพัง ให้ดู console/network เพื่อเก็บ error, status code และ request ที่เกี่ยวข้องโดยไม่เผย secret

มือใหม่มัก copy error แค่บรรทัดสุดท้าย

แต่หลายครั้งข้อมูลสำคัญอยู่ก่อนหน้านั้น

ถ้า error ยาว ให้ copy:

- command ที่รัน
- error ตั้งแต่เริ่ม fail
- stack trace ถ้ามี
- file path ที่ error ชี้
- line number ถ้ามี
- environment เช่น local หรือ Vercel

ถ้า error ยาวมาก ให้ถาม Claude ว่าควรดูส่วนไหน

> `error` นี้ยาวมาก
> ช่วยบอกว่าควรโฟกัสบรรทัดไหน และข้อมูลไหนสำคัญ
> อย่าเพิ่งแก้ไฟล์

อย่าตัด error ตามความรู้สึกถ้าคุณยังไม่รู้ว่าอะไรสำคัญ

ให้ Claude อธิบายก่อนแก้

เวลาเจอ error อย่ารีบให้แก้ทันที

เริ่มด้วย:

> ช่วยอธิบาย `error` นี้แบบ `non-coder` ก่อน
> มันน่าจะเกิดจากอะไร
> มีสาเหตุที่เป็นไปได้กี่แบบ
> เราควรตรวจอะไรก่อนแก้

ทำไมต้องทำแบบนี้?

เพราะถ้า Claude เข้าใจผิด คุณจะเห็นก่อนมันแก้ไฟล์

ถ้าคำอธิบายฟังดูไม่เกี่ยวกับสิ่งที่คุณทำ ให้ถามต่อ:

> สาเหตุนี้เกี่ยวข้องกับการเปลี่ยนล่าสุดตรงไหน
> มีหลักฐานจาก `log` บรรทัดไหนสนับสนุนไหม

ให้ AI อ้างหลักฐานจาก error ไม่ใช่เดาแบบมั่นใจ

แก้ทีละจุด

เวลา error เกิดขึ้น Claude อาจเสนอแก้หลายอย่างพร้อมกัน

สำหรับ non-coder ให้ระวัง

Debug ที่ดีควรแก้ทีละจุด เพื่อรู้ว่าอะไรเป็นสาเหตุจริง

Prompt:

> ช่วยแก้ด้วยวิธีที่เปลี่ยนน้อยที่สุดก่อน
> อย่า `refactor` ใหญ่
> อย่าเพิ่ม `package` ใหม่ถ้าไม่จำเป็น
> หลังแก้ ให้บอกว่าเปลี่ยนอะไร และต้อง `test` อะไร

ถ้า Claude เสนอแก้หลายไฟล์ ให้ถาม:

> เราสามารถแก้ทีละขั้นได้ไหม
> ขั้นแรกที่เล็กที่สุดและปลอดภัยที่สุดคืออะไร

เป้าหมายของ debug ไม่ใช่ทำ code สวยที่สุด

เป้าหมายคือหา root cause และทำให้ระบบกลับมาใช้ได้อย่างปลอดภัย

ใช้ Git เป็น safety net

ก่อนแก้ bug ใหญ่ ให้เช็กสถานะก่อน

```
• BASH

git status
```

ถ้ามี changes ที่ยังไม่ commit อยู่ ให้รู้ก่อนว่าเป็นของอะไร

ถ้าคุณปล่อยให้ Claude แก้ซ้อนบนงานที่ยังไม่บันทึก คุณอาจแยกไม่ออกว่าอะไรทำให้พัง

ก่อนแก้ bug ให้ถาม Claude:

```
ช่วยดู git status และสรุปว่าตอนนี้มีไฟล์อะไรเปลี่ยนอยู่
ก่อนเริ่มแก้ bug เราควร commit, stash, หรือแยกงานอย่างไร
อย่าแก้ไฟล์ก่อน
```

หลังแก้ bug ให้ดู diff:

```
ช่วยอธิบาย diff จากการแก้ bug นี้แบบ non-coder
แยกว่าแก้ root cause ตรงไหน และมีผลข้างเคียงอะไรที่ต้อง test
```

เมื่อไหร่ควร revert หรือ rollback

บางครั้งการแก้ต่อไม่ใช่ทางที่ดีที่สุด

ถ้าแก้ไปหลายรอบแล้วยิ่งพัง ให้หยุด

สัญญาณว่าควร revert:

- Claude แก้หลายไฟล์แต่ error ยังเหมือนเดิม
- error เปลี่ยนไปเรื่อย ๆ แต่ไม่เข้าใกล้สาเหตุ
- feature เดิมที่เคยใช้ได้เริ่มพัง
- คุณอธิบายไม่ได้แล้วว่า project เปลี่ยนอะไรไปบ้าง
- production พังและมี user จริงเจอปัญหา

ถ้าอยู่ใน local ให้ใช้ Git กลับ checkpoint ก่อนหน้า

ถ้าอยู่ production และ deploy บน Vercel อาจใช้ Instant Rollback เพื่อย้อน production deployment ไปเวอร์ชันก่อนหน้า

แต่ต้องเข้าใจว่า rollback แค่พาผู้ใช้กลับไปใช้ version เก่า

ไม่ได้อธิบายหรือแก้ root cause ให้คุณ

หลัง rollback ยังต้องกลับมาสืบว่า bug เกิดจากอะไร

Local พัง vs Production พัง

Local พังกับ Production พังต้องคิดคนละแบบ

ถ้า production กระทบ user จริง ให้กู้บริการหรือ rollback ก่อน แล้วค่อย debug ลึก

สองกรณีนี้ต้องคิดต่างกัน

Local พัง

ถ้าพังในเครื่องคุณ:

- หยุด
- เก็บ error
- ให้ Claude วิเคราะห์
- แก้ทีละจุด
- test ซ้ำ

ยังไม่ต้อง panic เพราะ user จริงยังไม่กระทบ

Production พัง

ถ้า production พัง:

- ประเมินว่า user กระทบไหม

- ถ้ากระทบหนัก ให้ rollback ก่อน
- จดเวลาและ deployment ที่พัง
- เก็บ error/log
- แยก branch หรือ local เพื่อแก้
- test ก่อน deploy ใหม่

กฎง่าย ๆ:

> ถ้า user จริงกำลังเจอปัญหา ให้กู้บริการก่อน แล้วค่อย debug ลึก

Session ยาวเกินไปก็ทำให้มั่วได้

บางครั้งคุณคุยกับ Claude นานมาก แก้ไปแก้มาหลายรอบ จน context เริ่มรก

Claude Code มีคำสั่งอย่าง `/clear`, `/compact`, `/resume` เพื่อจัดการ session

สำหรับ non-coder ให้ใช้หลักง่าย ๆ:

- ถ้า task เดิมยังต่อเนื่อง ใช้ session เดิมได้
- ถ้าแก้ผิดทางหลายรอบ ให้สรุปสิ่งที่เรียนรู้ แล้วเริ่ม session ใหม่
- ถ้ากลับมาทำต่อ ใช้ resume ได้
- ถ้า context รก ให้ขอ summary ก่อน clear

Prompt ก่อนเริ่มใหม่:

```
ช่วยสรุปสถานะ debug ตอนนี้ให้ฉันเอาไปเริ่ม session ใหม่
รวม:
1. ปัญหาคืออะไร
2. error/log สำคัญ
3. สิ่งที่ลองแล้ว
4. สิ่งที่ไม่น่าใช่สาเหตุ
5. next step ที่แนะนำ
```

แล้วค่อยเริ่ม session ใหม่ด้วย summary ที่สะอาดกว่า

Debug form submit ไม่ได้: ตัวอย่าง workflow

สถานการณ์:

> Waitlist form submit แล้วไม่เข้า Supabase

อย่าสั่งว่า:

> แก้ Supabase ให้หน่อย

ให้ใช้แบบนี้:

```
Waitlist form submit แล้วไม่เห็น row ใหม่ใน Supabase

Expected behavior:
- กรอก name/email/company/main_problem
- กด submit
- เห็น thank you message
- Supabase table waitlist_submissions มี row ใหม่

Actual behavior:
- กด submit แล้วเห็น error message: [ข้อความ]
- Supabase ไม่มี row ใหม่

Steps:
1. เปิด localhost:3000
2. กรอก form ด้วย test@example.com
3. กด submit
4. ดู Supabase table แล้วไม่มี row

Error/log:
[paste browser console/network/server log]

Recent changes:
- เพิ่งต่อ form กับ Supabase

กติกา:
- อย่าเพิ่งแก้ไฟล์
- ช่วยแยกสาเหตุที่เป็นไปได้ เช่น env, key, RLS, table name, validation, network
- บอกว่าควรตรวจอะไรทีละข้อ
```

Claude จะช่วยแยกปัญหาได้ดีกว่ามาก

Debug deploy fail: ตัวอย่าง workflow

Vercel build log ต้องอ่านบรรทัดที่ fail จริง

เวลา deploy fail ให้ส่ง log ที่มี command, error แรก, file path และ line number ไม่ใช่แค่บอกว่า deploy fail

สถานการณ์:

> ใช้ได้ใน `local` แต่ `Vercel build fail`

Prompt:

```
Project นี้รันใน local ได้ แต่ Vercel deploy fail

Expected behavior:
- Vercel build ผ่าน
- production URL เปิดได้

Actual behavior:
- Vercel build fail

Vercel build log:
[paste log]

Local command ที่เคยรันผ่าน:
[paste command เช่น npm run build]

Recent changes:
[สรุปล่าสุด]

กติกา:
- อย่าเพิ่งแก้ไฟล์
- ช่วยชี้บรรทัด log ที่สำคัญ
- แยกสาเหตุที่เป็นไปได้ เช่น env var, dependency, build command, TypeScript error
- เสนอ fix ที่เปลี่ยนน้อยที่สุด
```

Deploy fail ส่วนใหญ่มักมีหลักฐานใน log

อย่าให้ AI เดาโดยไม่มี log

Checklist ก่อนบอกว่า bug fixed

หลัง Claude แก้ bug อย่าเพิ่งเชื่อทันที

เช็กก่อน:

- ☐ error เดิมหายจริง
- ☐ run command เดิมผ่าน
- ☐ user flow เดิมใช้ได้
- ☐ feature อื่นที่เกี่ยวข้องยังไม่พัง
- ☐ diff เปลี่ยนเฉพาะสิ่งที่เกี่ยวข้อง

- ☐ ไม่มี secret โผล่
- ☐ ไม่มี package ใหม่โดยไม่จำเป็น
- ☐ Claude อธิบาย root cause ได้
- ☐ มี manual test steps
- ☐ commit หลัง fix แล้ว

ถ้า fix แล้วแต่ไม่รู้ root cause อย่างน้อยให้เขียนไว้ว่าเป็น assumption

อย่าเขียนว่า “แก้แล้ว” แบบไม่มีหลักฐาน

Prompt รวมท้ายบท

ใช้ prompt นี้เวลามี error

```
ฉันต้องการ debug ปัญหานี้แบบเป็นระบบ

## What happened
[เกิดอะไรขึ้น]

## Expected behavior
[ควรเกิดอะไร]

## Steps to reproduce
1. ...
2. ...
3. ...

## Error/log
[paste error/log]

## Recent changes
[เปลี่ยนอะไรล่าสุด]

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อธิบาย error แบบ non-coder
- ชี้บรรทัด log ที่สำคัญ
- เสนอ root causes ที่เป็นไปได้ 2-3 ข้อ
- บอกวิธีตรวจแต่ละข้อ
- แนะนำ fix ที่เปลี่ยนน้อยที่สุด
- ถ้าต้องแก้หลายไฟล์ ให้เสนอ plan ก่อน
```

หลังแก้ ใช้ prompt นี้:

```
ช่วย review bug fix นี้ก่อน commit
ตรวจว่า:
1. แก้ root cause หรือแค่กลบอาการ
2. เปลี่ยนไฟล์อะไรบ้าง
3. มีผลข้างเคียงอะไรที่ต้อง test
4. มี secret/package/config แปลก ๆ เพิ่มไหม
5. ต้องรัน manual test อะไร
6. commit message ควรเป็นอะไร
```

สรุป

Debug สำหรับ non-coder ไม่ได้แปลว่าต้องอ่าน code เก่ง

แต่ต้องเก็บหลักฐานให้ดี และไม่ปล่อยให้ AI แก้แบบเดา

จำ 6 ข้อนี้:

1. Error คือข้อมูล ไม่ใช่ความล้มเหลว
2. Copy log ให้ครบ
3. บอก expected behavior เสมอ
4. ให้ Claude วิเคราะห์ก่อนแก้
5. แก้ทีละจุดและดู diff
6. ถ้าพังใน production ให้ rollback ก่อนถ้ากระทบ user จริง

บทต่อไป เราจะรวบทุกอย่างเป็น security checklist สำหรับ vibe coder ก่อนเปิดให้คนอื่นใช้จริง

$ บทที่ 10

Security Checklist สำหรับ Vibe Coder

แก่นของบทนี้

แอปที่ “ใช้ได้” ไม่ได้แปลว่า “ปลอดภัย”

Claude Code ช่วยให้คนที่ไม่ใช่ programmer สร้างเว็บ form database dashboard และ deploy ได้เร็วมาก แต่ความเร็วไม่ได้ลบความเสี่ยงด้าน security

บทนี้ไม่พยายามทำให้คุณเป็น security engineer

เป้าหมายคือให้คุณมี checklist สั้น ๆ ก่อนเปิดให้คนอื่นใช้จริง:

- อะไรห้ามทำเด็ดขาด
- อะไรต้องให้ Claude ตรวจทุกครั้ง
- red flag หน้าตาเป็นแบบไหน
- เมื่อไหร่ต้องให้คน technical ช่วย review

Security สำหรับ non-coder คือการไม่พลาดเรื่องพื้นฐานที่ทำให้ข้อมูลหลุด ระบบพัง หรือสิทธิ์เปิดกว้างเกินไป

กฎข้อแรก: อย่า blind trust

OWASP ระบุ Blind Trust เป็นความเสี่ยงสำคัญของ citizen development และ AI-assisted coding

แปลแบบง่าย ๆ:

> `AI สร้างให้ → app รันได้ → เราเชื่อว่าถูกและปลอดภัยแล้ว`

นี่คือกับดัก

Claude ช่วยเขียน ช่วยอธิบาย และช่วย review ได้ แต่คุณต้องเป็นคนถือ checklist ไม่ใช่กด accept ทุกอย่างเพราะ “AI น่าจะรู้”

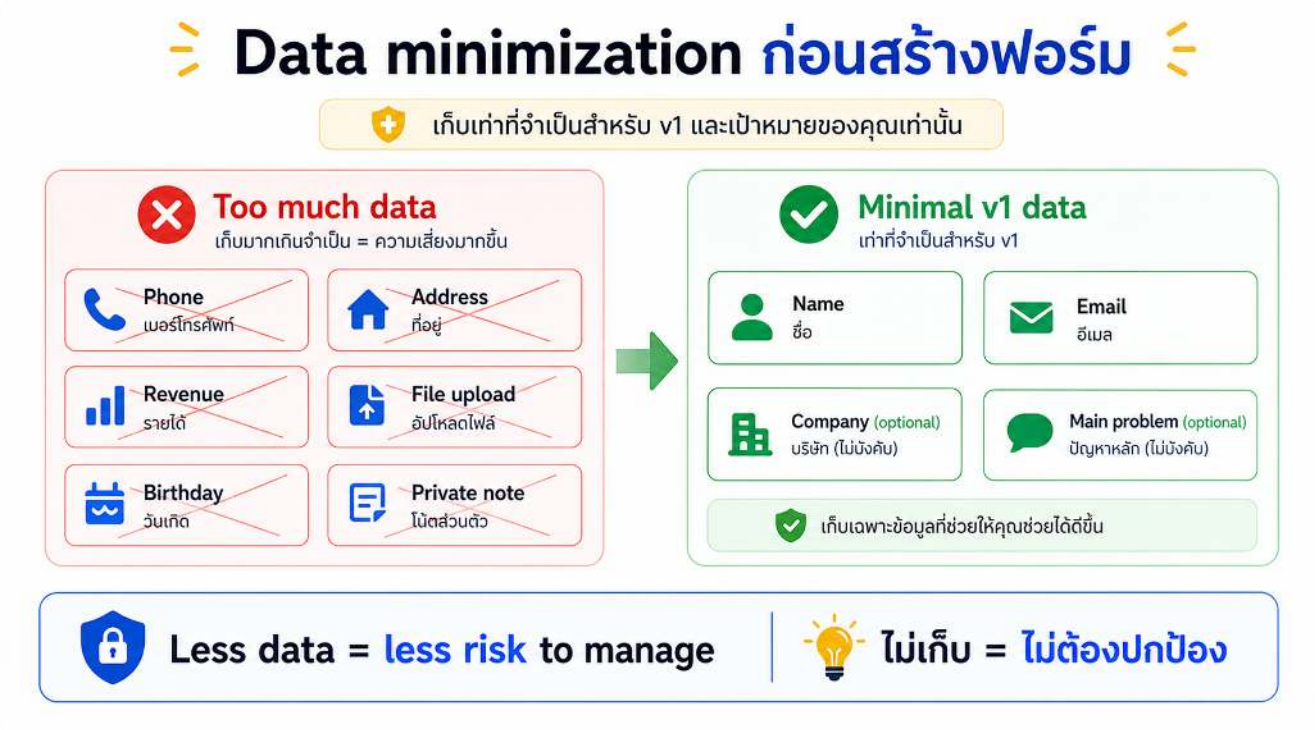
Checklist 10 ข้อก่อนเปิดให้คนอื่นใช้

Security launch gate: Green / Yellow / Red

Green / Yellow / Red: ใช้ launch gate เพื่อแยกว่าควร launch แบบจำกัด, beta/review ก่อน หรือห้าม launch

ใช้ checklist นี้ก่อน deploy จริง หรือก่อนส่ง link ให้คนอื่นกรอกข้อมูล

1. เก็บข้อมูลเท่าที่จำเป็น



Data minimization ก่อนสร้างฟอร์ม

ข้อมูลที่ไม่เก็บ คือข้อมูลที่ไม่ต้องปกป้อง ลด field ตั้งแต่แรกจึงลด risk ได้จริง

คำถามแรกคือ:

> จำเป็นต้องเก็บข้อมูลนี้จริงไหม

ตัวอย่าง waitlist อาจต้องมีแค่ email และ use case สั้น ๆ ก็พอ

อย่ารีบเก็บเบอร์โทร ที่อยู่ วันเกิด หรือข้อมูลส่วนตัวอื่น ถ้ายังไม่มีเหตุผลชัด

ข้อมูลที่ไม่เก็บ ย่อมไม่มีวันรั่วจากระบบของคุณ

Prompt:

```
ช่วย review data fields ของ app นี้
แยกเป็น 3 กลุ่ม:
1. จำเป็นต่อ feature หลัก
2. มีประโยชน์แต่ยังไม่จำเป็น
3. เสี่ยงหรือไม่ควรเก็บตอนนี้
อธิบายแบบ non-coder และอย่าเพิ่งแก้ code
```

2. ห้ามใส่ secret ใน code, GitHub หรือ frontend

Secret ไม่ควรไหลไปที่ไหนบ้าง

secret ไม่ควรไปอยู่ใน frontend, public repo, chat, screenshot, logs หรือ URL

Secret คือข้อมูลที่ถ้าคนอื่นเห็นแล้วเอาไปใช้แทนคุณได้ เช่น API secret key, database password, service role key, webhook secret, admin token

กฎง่าย ๆ:

```
secret ห้ามอยู่ใน GitHub
secret ห้ามอยู่ใน frontend/browser
secret ห้าม paste ลง prompt โดยไม่จำเป็น
```

ไฟล์ที่ต้องระวัง:

```
.env
.env.local
.env.production
config file
README หรือ screenshot ที่มี key จริง
```

ใน repo ควรมีได้แค่ไฟล์ตัวอย่าง เช่น `.env.example` ที่ใช้ placeholder ไม่ใช่ค่าจริง

Prompt:

```
ช่วยตรวจ project นี้ว่ามี secret หรือ credential หลุดใน code ไหม
กติกา:
- อย่า print ค่า secret จริงซ้ำออกมา
- ถ้าเจอ ให้บอกแค่ชื่อไฟล์และชนิดของ secret
- แนะนำวิธีย้ายไป environment variable
- ตรวจว่ามี .env ถูก track ใน git หรือไม่
```

---

3. เข้าใจ Supabase key แบบสั้นที่สุด

ถ้าใช้ Supabase ให้จำเรื่องนี้ก่อน deploy:

```
publishable key = ใช้ใน public app ได้ แต่ยังต้องมี RLS/policy คุม data
secret key / service_role = ใช้เฉพาะฝั่ง server/backend เท่านั้น
```

จาก official docs ของ Supabase: secret key ให้สิทธิ์ระดับสูงกับ project data และ service role เป็น elevated key ที่ bypass RLS ได้

ดังนั้น red flag คือ:

```
Claude เสนอให้ใส่ service_role key ใน frontend
Claude เสนอให้ใช้ secret key ใน client component
Claude เสนอให้ commit key จริงเพื่อให้ deploy ผ่าน
```

ถ้าเห็นแบบนี้ให้ถามทันที:

```
หยุดก่อน
key นี้เป็น publishable หรือ secret/service_role
มันจะอยู่ฝั่ง browser หรือ server
อธิบายความเสี่ยงแบบ non-coder
อย่าเพิ่งแก้ code
```

---

4. เปิด RLS และทำ policy ให้แคบ

RLS หรือ Row Level Security คือกฎว่าใครอ่าน/เขียน row ไหนได้บ้าง

หลักสำหรับ non-coder:

```
เปิด RLS ก่อน
แล้วค่อย allow เฉพาะ action ที่ feature ต้องใช้
```

ตัวอย่าง waitlist:

คนทั่วไปควรทำได้:

```
insert email ตัวเอง
```

คนทั่วไปไม่ควรทำได้:

```
select email ทุกคน
update/delete record
อ่าน dashboard leads
```

Prompt:

```
ช่วย review RLS และ database permissions ของ project นี้
ทำเป็นตาราง:
- table name
- user type: public / authenticated / admin
- action: select / insert / update / delete
- ควร allow หรือ deny
- เหตุผลแบบ non-coder
กติกา: อย่าเสนอปิด RLS เพื่อแก้ error
```

ถ้า Claude บอกว่า “ปิด RLS ก่อนเพื่อให้ใช้ได้” ให้ถือเป็น red flag สำหรับ app ที่จะเปิดจริง

---

5. Validate input ทั้งหน้าเว็บและฝั่ง server

Input คือสิ่งที่ user กรอก ส่ง หรือ upload เข้ามา เช่น email, name, message, URL, file, search box

อย่าคิดว่า user จะกรอกดี ๆ และอย่าคิดว่า form UI ตรวจแล้วพอ

Checklist พื้นฐาน:

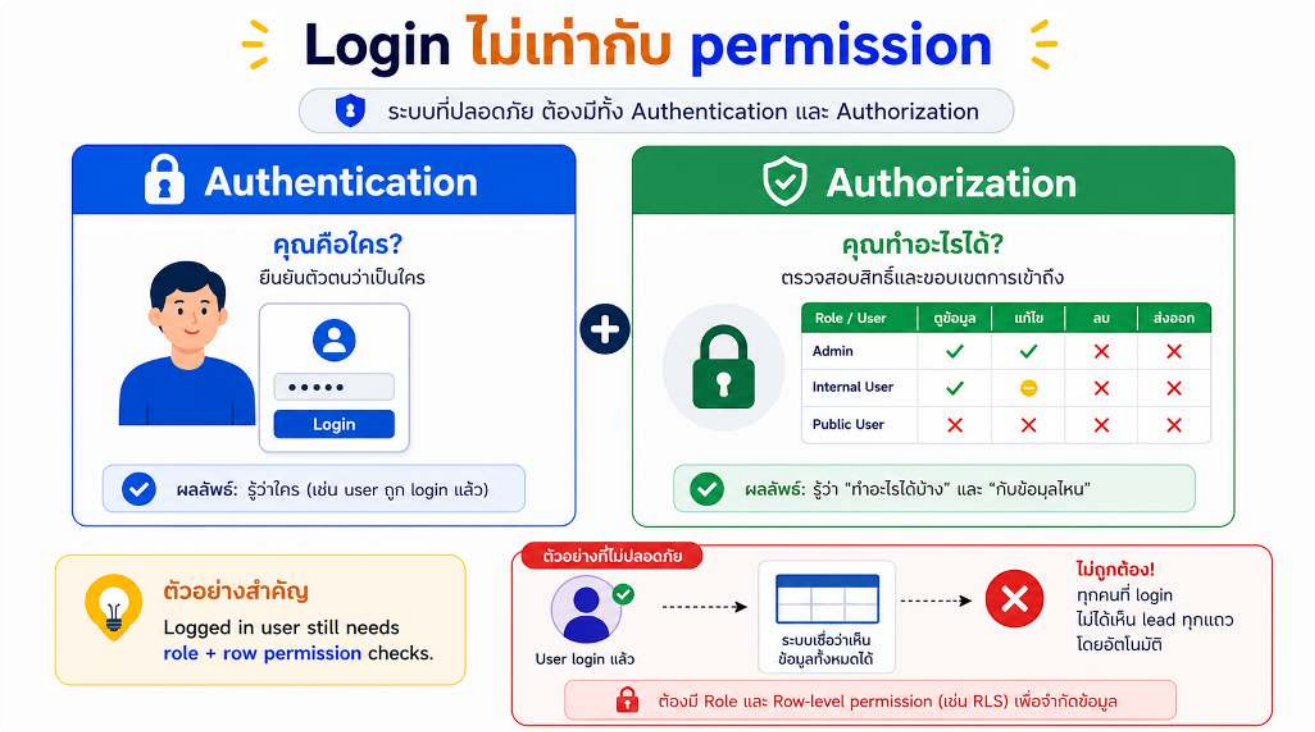
```
required field ครบไหม
email เป็น email จริงไหม
ความยาวไม่เกินที่กำหนดไหม
มี field แปลก ๆ ส่งมาได้ไหม
error message เปิดเผยข้อมูลระบบเกินไปไหม
```

Prompt:

```
ช่วย review input validation ของ feature นี้
ตรวจทั้ง frontend และ server/API/database layer
ตอบเป็น:
1. input แต่ละตัวคืออะไร
2. ต้อง validate อะไร
3. ตอนนี้ validate ที่ไหนแล้ว
4. ยังขาดอะไร
5. test case ที่ควรลอง
อย่าเพิ่งแก้ code จนกว่าจะเสนอ plan
```

---

6. Login ไม่เท่ากับ permission



Login ไม่เท่ากับ permission

login บอกว่า user คือใคร แต่ permission ต้องตอบว่า user คนนั้นทำอะไรได้บ้าง

Login ตอบว่า “คุณคือใคร”

Permission ตอบว่า “คุณมีสิทธิ์ทำอะไร”

มี login แล้วไม่ได้แปลว่าปลอดภัยแล้ว

ตัวอย่าง: User A login แล้วควรเห็นข้อมูลตัวเอง แต่ไม่ควรเห็นข้อมูลของ User B

Prompt:

```
ช่วยแยก auth กับ authorization ของ app นี้
ตอบเป็นตาราง:
- role/user type
- เข้าหน้าไหนได้
- อ่านข้อมูลอะไรได้
- สร้างข้อมูลอะไรได้
- แก้/ลบอะไรได้
- enforce ตรงไหนใน code หรือ RLS
- ช่องโหว่ที่อาจเกิดขึ้น
อธิบายแบบ non-coder
```

ถ้า Claude แก้ permission ด้วยการใช้ admin token ทุกที่ ให้หยุดทันที

---

7. อย่าติดตั้ง package มั่ว

เวลา error เกิดขึ้น AI อาจเสนอให้ติดตั้ง package เพิ่ม

บางครั้งถูก แต่บางครั้งเพิ่มความเสี่ยงโดยไม่จำเป็น

ก่อนติดตั้ง ถาม 5 ข้อนี้:

```
จำเป็นจริงไหม
มี built-in ทำได้ไหม
package นี้น่าเชื่อถือไหม
ใช้ทำอะไรแค่ไหน
ถ้าเอาออก app ยังทำงานได้ไหม
```

Prompt:

```
ก่อนติดตั้ง package เพิ่ม ช่วยอธิบายว่า:
1. package นี้แก้ปัญหาอะไร
2. จำเป็นจริงไหม
3. มีทางเลือกที่ไม่ต้องเพิ่ม dependency ไหม
4. มีผลต่อ security หรือ deploy อย่างไร
5. ถ้าติดตั้งแล้ว ต้อง test อะไร
อย่าเพิ่งรัน install
```

สำหรับ project เล็ก ๆ หลายครั้ง code ธรรมดาดีกว่า dependency เยอะ ๆ

---

8. แยก local, preview, production ให้ชัด

คำที่ต้องรู้:

```
local = เครื่องคุณ
preview = ทดสอบก่อนจริง
production = ของจริงที่ user ใช้
```

สิ่งที่ต้องแยก:

- environment variables
- database URL/key
- payment mode
- email provider
- test data กับ real data

Vercel ใช้ environment variables เพื่อเก็บค่าที่เปลี่ยนตาม environment และการเปลี่ยน env var มักมีผลกับ deployment ใหม่ ไม่ใช่ deployment เก่าทันที

Prompt ก่อน deploy:

```
ช่วยตรวจ deploy readiness ของ project นี้
แยก local / preview / production
ตรวจว่า:
- env var ที่จำเป็นมีอะไรบ้าง
- ค่าไหนเป็น public ได้
- ค่าไหนเป็น secret
- ค่าไหนยังเป็น test/dev value
- ต้อง redeploy หลังเปลี่ยน env var ไหม
- มี rollback plan ไหม
อย่า print ค่า secret จริง
```

---

9. Log ต้องไม่ทำข้อมูลหลุด

Log มีไว้ debug แต่ log ที่แย่ทำข้อมูลลับหลุดได้

ห้าม log สิ่งเหล่านี้:

- password
- secret key
- token
- payment info
- private customer data ที่ไม่จำเป็น
- request body ทั้งก้อนถ้ามีข้อมูล sensitive

ตัวอย่างอันตราย:

```
console.log(process.env.SUPABASE_SECRET_KEY)
console.log(fullUserObject)
console.log(webhookPayload)
```

Prompt:

```
ช่วย review logging ใน project นี้
ตรวจว่ามี log ที่อาจเปิดเผย secret หรือข้อมูลส่วนตัวไหม
กติกา:
- อย่า print ค่า secret จริงซ้ำ
- เสนอวิธี mask/redact
- แยก log ที่ควรเก็บกับ log ที่ควรลบ
- แนะนำ error message ที่ user เห็นได้โดยไม่เผยข้อมูลระบบ
```

---

10. ปิดสิ่งที่ไม่ใช้ และรู้ว่าใครเป็นเจ้าของ

App เล็ก ๆ ก็สร้างหนี้ด้าน security ได้ โดยเฉพาะ project ที่ทำไว้แล้วลืม

Checklist หลัง launch:

```
ใครเป็น owner ของ project นี้
ถ้าคุณไม่ว่าง ใครเข้า Supabase/Vercel/GitHub ได้
มี key เก่าที่ไม่ใช้แล้วไหม
มี database/table/test app ที่เปิด public ทิ้งไว้ไหม
มี integration ที่ไม่ใช้แล้วไหม
มี domain หรือ deployment เก่าที่ควรปิดไหม
```

Prompt:

```
ช่วยทำ asset inventory สำหรับ project นี้
แยกเป็น:
- repository
- deploy platform
- database
- API keys / env vars
- third-party services
- domains
- scheduled jobs/webhooks
- owners/admins
บอกด้วยว่าอะไรควรปิด ลบ rotate หรือ review
```

Red flags ที่ต้องหยุดทันที

ถ้าเจอสิ่งเหล่านี้ อย่าเพิ่ง accept code:

1. ปิด RLS เพื่อให้ app ใช้ได้
2. secret/service_role อยู่ใน browser
3. commit `.env` ที่มีค่าจริง
4. dashboard public อ่านข้อมูลทุกคนได้
5. แก้ bug ด้วย admin permission หรือ service role ทุกที่
6. เพิ่ม package เยอะเพื่อแก้ปัญหาเล็ก
7. Claude อธิบายไม่ได้ว่าใครมีสิทธิ์ทำอะไร

ประโยคที่ควรใช้เมื่อเจอ red flag:

```
หยุดก่อน
อธิบายความเสี่ยงของวิธีนี้แบบ non-coder
มีทางเลือกที่ปลอดภัยกว่าและเปลี่ยนน้อยกว่าไหม
อย่าเพิ่งแก้ไฟล์
```

Security review prompt ก่อน deploy

ใช้ prompt นี้ก่อนแชร์ link จริง

```
ช่วยทำ security review ก่อน deploy สำหรับ project นี้
ฉันเป็น non-coder ขอให้อธิบายแบบเข้าใจง่าย

ขอบเขต:
- frontend
- API/server actions
- Supabase/database/RLS
- environment variables
- authentication/authorization
- input validation
- logging
- dependencies
- deployment settings

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อย่า print secret จริง
- แยกผลเป็น Critical / High / Medium / Low
- ทุก issue ต้องมี: หลักฐาน, ความเสี่ยง, วิธีแก้ที่แนะนำ
- ถ้าไม่แน่ใจ ให้บอกว่าไม่แน่ใจ
- สรุปท้ายว่า launch ได้ไหมแบบ Green / Yellow / Red
```

หลังได้ report ให้แก้ Critical ก่อน แล้วค่อย High

อย่าให้ Claude แก้ทุกอย่างรวดเดียวถ้ามีหลาย issue

Green / Yellow / Red launch gate

Green — launch ได้แบบจำกัด scope

ควรมีสิ่งเหล่านี้ครบ:

- ไม่มี secret ใน code/frontend/GitHub
- RLS เปิดและ policy แคบพอ
- public user อ่านข้อมูลคนอื่นไม่ได้
- form validate input พื้นฐานแล้ว
- env var แยก production ชัด
- มี rollback หรือวิธีปิด app ถ้าเกิดปัญหา

Yellow — launch เฉพาะ beta วงเล็ก

ใช้เมื่อ feature หลักใช้ได้ แต่ยังมี issue medium บางอย่าง และยังไม่มีข้อมูล sensitive มาก

ต้องรู้ว่าจะ monitor อะไร และมีแผนแก้ภายในเวลาสั้น ๆ

Red — ห้าม launch

ตัวอย่าง:

- secret/service role อยู่ใน frontend
- RLS ปิดกับ table ที่มีข้อมูล user
- dashboard public เห็นข้อมูลทุกคน
- payment/login/file upload ยังไม่ได้ review
- Claude ยังอธิบาย permission model ไม่ชัด
- มีข้อมูล sensitive แต่ไม่มีคน technical review

ถ้าได้ Red อย่าให้ FOMO ชนะ checklist

เมื่อไหร่ต้องให้คนช่วย review

ให้หาคน technical ช่วย review ถ้า app มีสิ่งเหล่านี้:

- รับเงินหรือเกี่ยวข้องกับ payment
- เก็บข้อมูลส่วนตัวจำนวนมาก
- เก็บข้อมูลสุขภาพ การเงิน กฎหมาย หรือข้อมูลเด็ก
- มี login หลาย role เช่น user/admin/team
- มี file upload
- มี dashboard สำหรับข้อมูลลูกค้า
- เชื่อมต่อ system ภายในบริษัท
- ใช้ secret/admin key/server-side integration
- ถ้าพังแล้วกระทบรายได้หรือชื่อเสียงมาก

Prompt เตรียมส่งให้ reviewer:

```
ช่วยสร้าง security review brief ให้ human reviewer
สรุป:
- app ทำอะไร
- user types มีใครบ้าง
- data เก็บอะไร
- services ที่ใช้
- auth/permission model
- env vars/secret ที่เกี่ยวข้องโดยไม่เปิดเผยค่าจริง
- จุดที่ฉันกังวล
- คำถามที่อยากให้ reviewer ตรวจ
```

Security ไม่ใช่ครั้งเดียวจบ

หลัง launch แล้วให้มีรอบ review ง่าย ๆ:

```
ก่อน deploy ใหญ่: review checklist 10 ข้อ
ทุกเดือน: ดู key, dependency, logs, RLS, unused assets
ทุกครั้งที่เพิ่ม feature ใหม่: review data + permission ใหม่
ทุกครั้งที่ key หลุด: rotate ทันที
```

อย่าคิดว่า app เล็กไม่เป็นเป้า

หลายครั้ง attacker ไม่ได้เลือกคุณเพราะดัง แต่เลือกเพราะระบบ public และตั้งค่าพลาด

Prompt รวมท้ายบท

ใช้ prompt นี้เป็นด่านสุดท้ายก่อน production

```
ทำ final pre-launch security checklist ให้ project นี้
ฉันเป็น non-coder

ตรวจ 10 เรื่องนี้:
1. data minimization
2. secrets/env vars
3. Supabase publishable vs secret/service_role keys
4. RLS และ database policies
5. input validation
6. auth vs authorization
7. dependencies/packages
8. local/preview/production config
9. logs/error messages
10. asset ownership/lifecycle

กติกา:
- อย่าแก้ไฟล์ก่อน
- อย่า print secret จริง
- ถ้าเจอ critical issue ให้หยุดและอธิบายก่อน
- ทำ report เป็นตาราง
- ให้ launch gate เป็น Green / Yellow / Red
- เสนอ next steps ไม่เกิน 5 ข้อ
```

สรุป

Security สำหรับ vibe coder เริ่มจากนิสัย 5 อย่าง:

1. ไม่ blind trust AI
2. ไม่เก็บข้อมูลเกินจำเป็น
3. ไม่ปล่อย secret หลุด
4. ไม่ปิด security เพื่อให้ app ใช้ได้เร็ว
5. ให้ AI อธิบาย permission เป็นภาษาคนก่อน deploy

ถ้าคุณทำแค่นี้ได้ คุณจะปลอดภัยกว่าการ vibe coding แบบกด accept ทุกอย่างมากแล้ว

บทสุดท้าย เราจะคุยเรื่องการ launch, เก็บ feedback, iterate, ใช้ PR/preview เป็น safety net และส่งต่อให้ developer เมื่อความเสี่ยงเริ่มเกินสิ่งที่ non-coder ควรแบกเอง

$ บทที่ 11

Launch, Iterate, and Hand Off

แก่นของบทนี้

Vibe coding ที่ดีไม่ได้จบที่คำว่า “deploy แล้ว”

ของจริงเริ่มหลังจากมีคนใช้

เพราะตอนนั้นคุณจะเจอ:

- user ไม่เข้าใจสิ่งที่คุณคิดว่าชัด
- form ที่คุณ test เองผ่าน แต่ user กรอกไม่เหมือนคุณ
- feature ที่คิดว่าสำคัญ กลับไม่มีใครใช้
- bug ที่เกิดเฉพาะ production
- ค่าใช้จ่ายและ maintenance ที่เริ่มตามมา

บทนี้คือบทปิด workflow:

> `launch เล็ก → เก็บ feedback → แปลงเป็น backlog → แก้เป็นรอบ → version → handoff ถ้าจำเป็น`

เป้าหมายไม่ใช่ launch ให้ใหญ่ที่สุด

แต่คือ launch แบบควบคุมความเสี่ยงได้

ถ้าคุณใช้ GitHub + Vercel ให้เปิด **Appendix G — GitHub PR + Preview Release Workflow** คู่กับบทนี้ เพื่อใช้ PR และ preview deployment เป็นด่านตรวจงานก่อน production

---

อย่า launch ใหญ่ตั้งแต่วันแรก

คนที่ใช้ Claude Code แล้วได้ของเร็ว มักอยากเปิดตัวทันที

นี่เข้าใจได้

แต่สำหรับ non-coder วิธีที่ปลอดภัยกว่าคือ small launch

> ไม่ใช่ “เปิดให้ทุกคนใช้”
> แต่คือ “เปิดให้กลุ่มเล็กที่ให้ feedback ได้”

ตัวอย่าง small launch:

- ส่งให้เพื่อนร่วมทีม 3–5 คนลอง
- เปิด beta ให้ลูกค้าเก่า 10 คน
- แชร์ link เฉพาะในกลุ่มเล็ก
- ใช้เองภายใน 1 สัปดาห์ก่อนเปิด public
- เปิด waitlist แต่ยังไม่เปิด feature หลักทั้งหมด

ข้อดี:

- bug กระทบคนน้อย
- feedback อ่านไหว
- ยัง rollback หรือปิด app ได้ง่าย
- คุณเรียนรู้ก่อนเสียเครดิตใหญ่

Prompt ก่อน small launch:

```
ช่วยวาง small launch plan สำหรับ app นี้
ฉันเป็น non-coder

ให้ตอบเป็น:
1. ควรเปิดให้ใครลองก่อน
2. จำนวน user แรกที่เหมาะสม
3. feature ไหนควรเปิด/ปิด
4. สิ่งที่ต้อง monitor
5. ถ้าเกิดปัญหา ต้องปิดหรือ rollback อย่างไร
6. เกณฑ์ว่าพร้อมขยาย audience หรือยัง
```

Launch checklist แบบไม่ยาว

ก่อนส่ง link ให้คนอื่น ใช้ checklist นี้

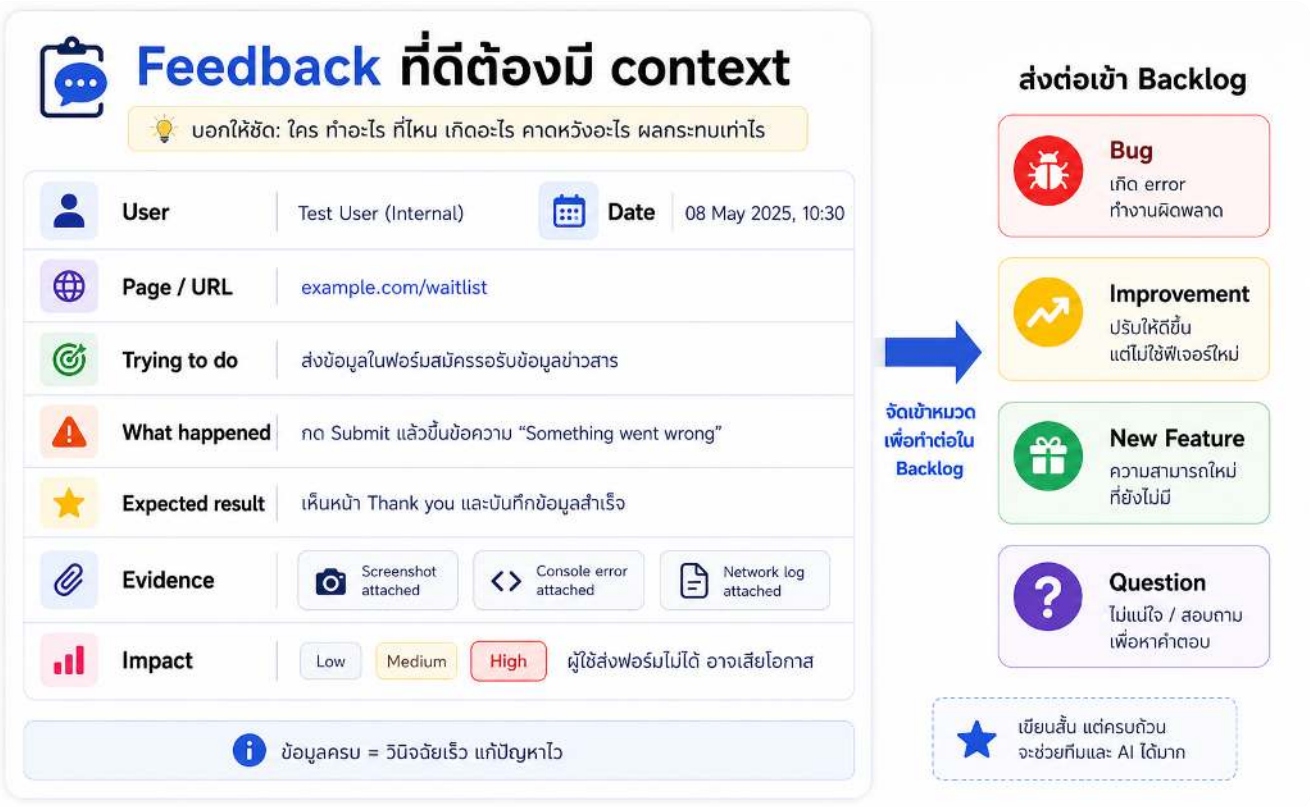
```
[ ] หน้า/flow หลักใช้ได้ตั้งแต่ต้นจนจบ
[ ] mobile ใช้ได้
[ ] form มี success/error state
[ ] ข้อความ error เป็นภาษาคน ไม่โชว์ข้อมูลระบบ
[ ] ไม่มี secret ใน frontend/GitHub
[ ] RLS/policy ผ่าน review
[ ] env var production ตั้งครบ
[ ] มีวิธี rollback หรือปิด app
[ ] มีที่เก็บ feedback
[ ] มีคนรับผิดชอบดูหลัง launch
```

อย่าเพิ่ม checklist จน launch ไม่ได้

แต่ห้ามข้ามข้อที่เกี่ยวกับ secret, permission, data และ rollback

---

เก็บ feedback ให้เป็นระบบ



Feedback ที่ดีต้องมี context

feedback ที่ดีต้องบอกว่า user พยายามทำอะไร เกิดอะไรขึ้น คาดหวังอะไร และมีหลักฐานอะไร

Feedback ที่ดีต้องช่วยให้คุณตัดสินใจได้

ไม่ใช่แค่ข้อความกว้าง ๆ เช่น:

```
ใช้งานยาก
อยากให้สวยขึ้น
เพิ่ม AI หน่อย
```

ให้เก็บ feedback แบบมี context:

```
ใครเจอปัญหา
เขาพยายามทำอะไร
เกิดอะไรขึ้น
คาดหวังอะไร
มี screenshot หรือ URL ไหม
กระทบมากแค่ไหน
```

Template สำหรับ feedback:

```
## Feedback
ผู้ใช้:
วันที่:
หน้า/URL:

## What were you trying to do?
[เขาพยายามทำอะไร]

## What happened?
[เกิดอะไรขึ้น]

## Expected result
[เขาคาดหวังอะไร]

## Evidence
[screenshot / error / screen recording / steps]

## Impact
[low / medium / high]
```

ใช้ Google Form, Notion, GitHub Issues หรือ sheet ธรรมดาก็ได้

เครื่องมือไม่สำคัญเท่ารูปแบบข้อมูล

ให้ Claude แปลง feedback เป็น backlog

หลังมี feedback อย่ารีบแก้ทันที

ให้ Claude ช่วยจัดกลุ่มก่อน

Prompt:

```
ฉันมี feedback จากผู้ใช้ชุดนี้
ช่วยแปลงเป็น backlog แบบเป็นระบบ

กติกา:
- อย่าเพิ่งเสนอ code
- รวม feedback ที่ซ้ำกัน
- แยกเป็น Bug / Improvement / New Feature / Question
- ให้ priority เป็น P0 / P1 / P2 / Later
- บอกเหตุผลของ priority แบบ non-coder
- ถ้าข้อมูลไม่พอ ให้ระบุว่าต้องถาม user อะไรเพิ่ม

Feedback:
[paste feedback]
```

สิ่งสำคัญคือแยก 4 ประเภทนี้ให้ชัด:

Bug

ของที่ควรทำงานแต่ไม่ทำงาน

เช่น submit form แล้ว error

Improvement

ของที่ทำงานได้ แต่ควรดีขึ้น

เช่น copy ไม่ชัด ปุ่มหาไม่เจอ mobile spacing แคบ

New Feature

ของใหม่ที่ยังไม่เคยอยู่ใน scope

เช่น เพิ่ม login เพิ่ม export CSV เพิ่ม payment

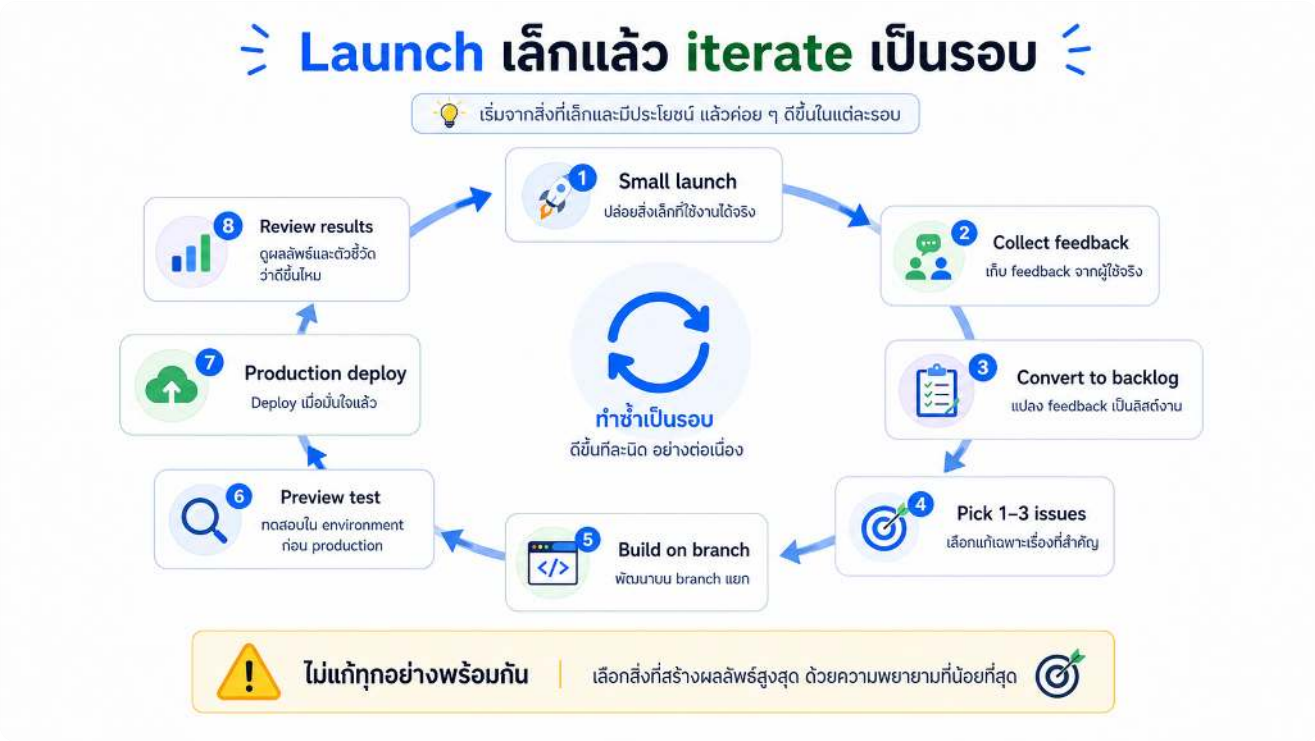
Question

feedback ที่ยังไม่ชัด ต้องถามต่อ

เช่น “อยากให้ dashboard ฉลาดกว่านี้”

อย่าให้ทุก feedback กลายเป็น feature

กฎการ iterate: ทีละรอบเล็ก



Launch เล็กแล้ว iterate เป็นรอบ

หลัง launch เล็ก ให้เก็บ feedback แล้วเลือกแก้ทีละ 1–3 issue ไม่ใช่แก้ทุกอย่างพร้อมกัน

การ iterate ที่ดีไม่ใช่แก้ทุกอย่างพร้อมกัน

ใช้รอบสั้น ๆ:

```
เลือก 1–3 issue → ให้ Claude วาง plan → แก้ → test → commit → deploy preview
→ ตรวจ → merge/deploy
```

ถ้าแก้ 20 อย่างพร้อมกัน คุณจะไม่รู้ว่าอะไรทำให้พัง

Prompt สำหรับรอบแก้ไข:

```
เราจะทำ iteration รอบนี้จาก backlog
เลือกเฉพาะ issue ต่อไปนี้:
- [issue 1]
- [issue 2]

กติกา:
- อย่าแตะ feature อื่น
- ก่อนแก้ ให้สรุปไฟล์ที่คาดว่าจะเปลี่ยน
- หลังแก้ ให้สรุป diff แบบ non-coder
- ระบุ manual test ที่ต้องทำ
- ถ้าเจอ scope เพิ่ม ให้หยุดถามก่อน
```

นี่คือวิธีคุม scope

ไม่ใช่ปล่อยให้ AI “ถือโอกาสปรับให้ดีขึ้นทั้งระบบ”

Versioning แบบ non-coder

คุณไม่ต้องเข้าใจ Git ลึกมาก แต่ต้องเข้าใจ 3 คำนี้

```
commit = save point ของ code
branch = พื้นที่ลองแก้โดยไม่กระทบ main
pull request = หน้าตรวจ changes ก่อนเอาเข้า main
```

สำหรับงานคนเดียว ใช้ pattern ง่าย ๆ:

```
main = version ที่ deploy ได้
feature branch = งานที่กำลังแก้
commit = จบงานย่อยหนึ่งชิ้น
```

ตัวอย่างชื่อ branch:

```
fix-waitlist-submit
improve-mobile-hero
add-lead-export
```

ตัวอย่าง commit message:

```
fix: handle duplicate waitlist emails
feat: add lead status filter
docs: update setup instructions
chore: remove unused package
```

ถ้าคุณไม่รู้จะตั้งชื่อยังไง ให้ถาม Claude:

```
ช่วยตั้งชื่อ branch และ commit message สำหรับ changes นี้
ใช้ Conventional Commits
อธิบายแบบสั้นว่าทำไมเลือกชื่อนี้
```

ใช้ Pull Request เป็นหน้าตรวจงาน

GitHub Pull Request หรือ PR คือหน้าที่ใช้ดูว่า branch นี้เปลี่ยนอะไร ก่อน merge เข้า main

สำหรับ non-coder ให้คิดว่า PR คือ:

```
ใบส่งงานของ AI ที่เราตรวจได้ก่อนปล่อยจริง
```

ใน PR คุณควรดู:

- เปลี่ยนไฟล์อะไรบ้าง
- มีไฟล์แปลก ๆ เพิ่มไหม
- มี `.env` หรือ secret หลุดไหม
- package เพิ่มไหม
- screenshots หรือ preview link เป็นอย่างไร
- test ผ่านไหม

- manual test ทำครบไหม

Prompt ให้ Claude เตรียม PR summary:

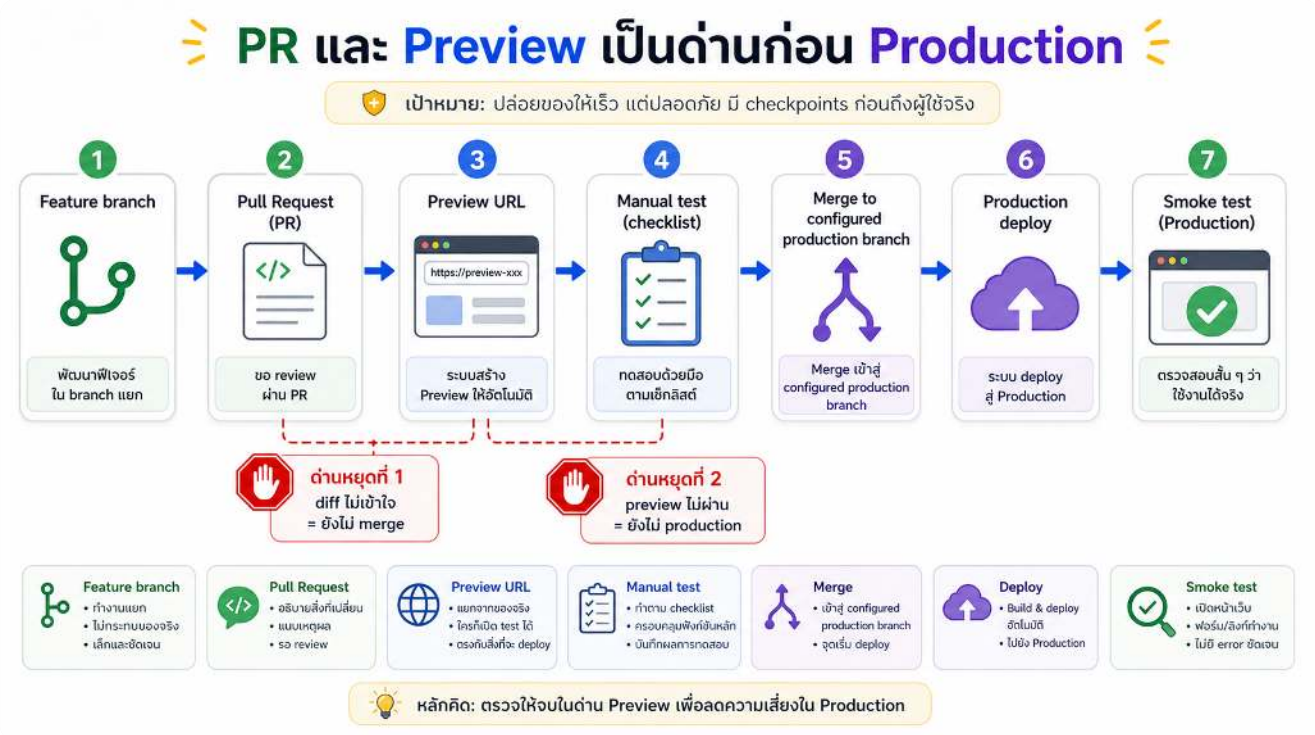
```
ช่วยเขียน PR summary สำหรับ changes นี้

ต้องมี:
1. ทำอะไร
2. ทำไมต้องทำ
3. ไฟล์สำคัญที่เปลี่ยน
4. วิธี test
5. screenshots/preview link ที่ควรแนบ
6. risks หรือสิ่งที่ reviewer ควรดูเป็นพิเศษ
7. checklist ก่อน merge

ใช้ภาษาที่ non-coder อ่านเข้าใจ
```

ถ้าคุณทำงานคนเดียว PR ก็ยังมีประโยชน์ เพราะมันบังคับให้หยุดดู diff ก่อน merge

Release gate แบบสั้นก่อน merge:

```
ถ้า diff ยังอ่านไม่เข้าใจ → ยังไม่ merge
ถ้า preview ยังไม่ได้ test → ยังไม่ merge
ถ้า secret/RLS/env ยังไม่เช็ก → ยังไม่ merge
ถ้า rollback plan ยังไม่มี → ยังไม่ launch ใหญ่
```

Preview deployment สำคัญกว่า production



PR และ Preview เป็นด่านก่อน Production

ใช้ PR และ preview URL เป็นด่านตรวจ diff/test ก่อน merge ไป production

Vercel รองรับ preview deployment จาก branch/PR

แปลว่า คุณสามารถดู version ที่แก้แล้วก่อนเอาเข้า production

workflow ที่ดี:

> `แก้ใน branch → push → ดู preview URL → test → merge → production deploy`

อย่าทดสอบครั้งแรกใน production ถ้าเลี่ยงได้

สิ่งที่ควร test บน preview:

- หน้าเปิดได้
- mobile layout
- form submit
- empty/error/success state
- permission ของ user แต่ละ role
- env var ที่ preview ต้องใช้

Prompt:

```
ช่วยทำ preview test plan สำหรับ PR นี้
แยกเป็น:
- smoke test 5 นาที
- manual test สำคัญ
- security/data check
- mobile check
- สิ่งที่ไม่ต้อง test รอบนี้
```

Rollback ไม่ใช่ความพ่ายแพ้

Rollback คือการย้อน production กลับไป version ก่อนหน้าเมื่อของใหม่พัง

Vercel มี Instant Rollback สำหรับ production deployment

แต่ต้องจำว่า rollback อาจเกี่ยวข้องกับ config, env var, cron job, database หรือ external service ด้วย ไม่ใช่แค่หน้าเว็บ

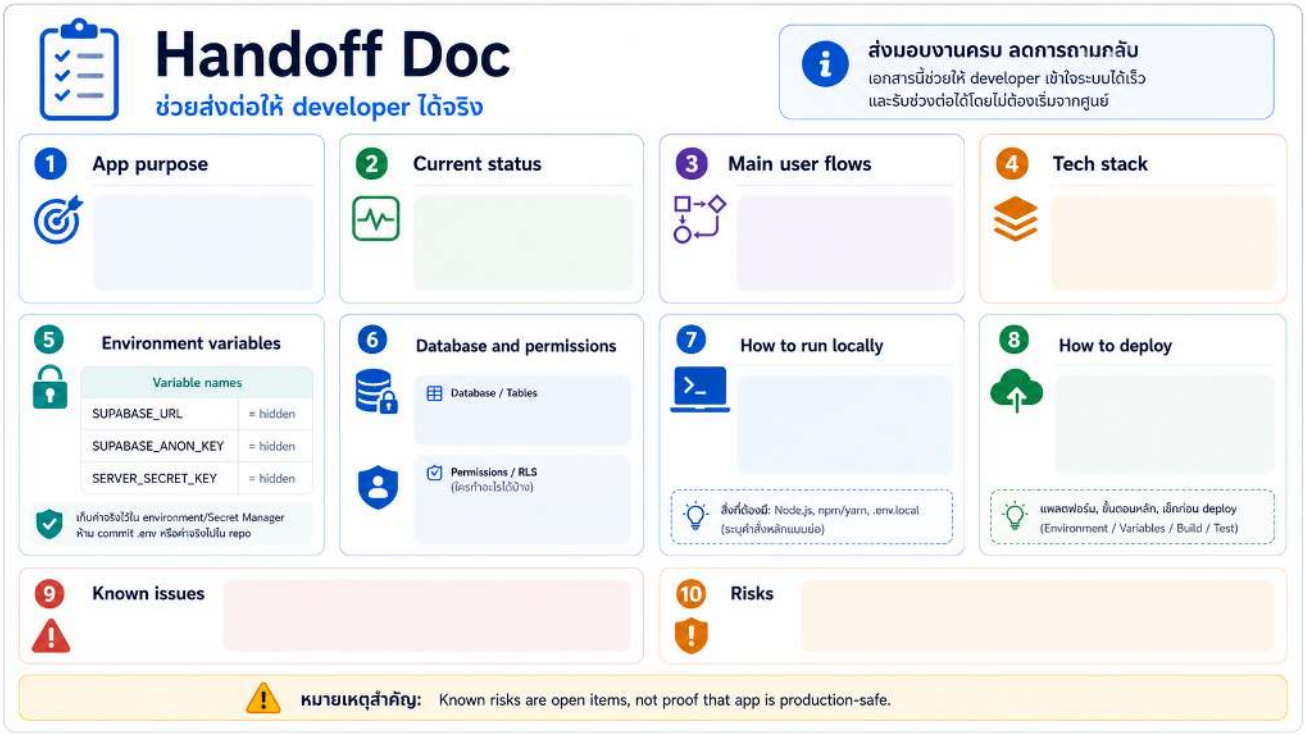
หลักง่าย ๆ:

```
ถ้า production กระทบ user จริง ให้หยุดเลือดก่อน แล้วค่อย debug
```

Prompt ตอนเกิดเหตุ:

```
production มีปัญหาหลัง deploy ล่าสุด
ช่วยทำ incident triage แบบ non-coder

ข้อมูล:
- URL:
- เวลา deploy:
- สิ่งที่เปลี่ยนล่าสุด:
- user กระทบอย่างไร:
- error/log:

กติกา:
- ก่อนเสนอ fix ให้ประเมินว่าควร rollback ไหม
- ถ้า rollback ให้บอกผลกระทบที่ต้องเช็ก
- หลังระบบกลับมา ค่อยเสนอ root cause analysis
```

Handoff: ส่งต่อให้ developer อย่างไร



Handoff doc ช่วยส่งต่อให้ developer ได้จริง

handoff doc ที่ดีช่วยให้ developer เข้าใจ purpose, flow, stack, env, database, deploy, risks และ known issues

บาง project โตจนควรส่งต่อให้ developer

การ handoff ที่ดีทำให้ dev ไม่ต้องเดา

สิ่งที่ควรมี:

```
README ที่อธิบาย app
วิธี run local
env vars ที่ต้องใช้ แต่ไม่ใส่ค่าจริง
architecture overview
database schema
RLS/permission model
deploy process
known issues
backlog
decision log
```

Handoff doc template:

```
# Handoff Doc

## App purpose
[app นี้ทำอะไร เพื่อใคร]

## Current status
[prototype / beta / production]

## Main user flows
1. ...
2. ...

## Tech stack
- Frontend:
- Backend/API:
- Database:
- Deploy:

## Environment variables
| Name | Public/Secret | Used for | Where set |
|---|---|---|---|

## Database and permissions
- Tables:
- RLS policies:
- Roles:

## How to run locally
[คำสั่งและขั้นตอน]

## How to deploy
[branch, Vercel project, env notes]

## Known issues
[รายการ]

## Backlog
[P0/P1/P2/Later]

## Risks
[security, data, cost, operational]
```

Prompt:

```
ช่วยสร้าง handoff doc จาก project นี้
ฉันจะส่งต่อให้ developer

กติกา:
- อย่าใส่ค่า secret จริง
- ถ้าไม่รู้ข้อมูลส่วนไหน ให้ใส่ TODO
- อธิบาย architecture แบบภาพรวม
- สรุป risks และ known issues ตรง ๆ
```

Maintenance checklist รายเดือน

ถ้าคุณยังดูแล app เอง ให้มี routine สั้น ๆ เดือนละครั้ง

```
[ ] login/admin account ยังปลอดภัย และเปิด 2FA แล้ว
[ ] secret/API key ที่ไม่ใช้ถูกลบหรือ rotate แล้ว
[ ] Supabase RLS/policies ยังถูกต้องหลังเพิ่ม feature
[ ] package/dependency ไม่มี issue สำคัญที่ต้องแก้
[ ] logs ไม่มีข้อมูล sensitive
[ ] backup/export สำคัญมีแผนแล้ว
[ ] Vercel/Supabase billing ยังอยู่ในงบ
[ ] domain/SSL/deployment ทำงานปกติ
[ ] backlog ถูกจัด priority ใหม่
[ ] unused project/deployment ถูกปิด
```

Prompt:

```
ช่วยทำ monthly maintenance review สำหรับ project นี้
ตรวจจาก checklist ต่อไปนี้
ให้ผลเป็น Green / Yellow / Red
และเสนอ action ไม่เกิน 5 ข้อ

[วาง checklist]
```

อย่าทำให้ maintenance ใหญ่จนไม่ทำ

ทำเล็กแต่สม่ำเสมอดีกว่า

Cost awareness: ของที่สร้างได้เร็วอาจมีค่าใช้จ่ายต่อเนื่อง

Claude Code, Vercel, Supabase, domain, email service, database และ third-party API อาจมีค่าใช้จ่าย

สิ่งที่ non-coder ต้องรู้:

> prototype ฟรีหรือถูก ไม่ได้แปลว่า production จะฟรีเสมอ

ค่าใช้จ่ายที่ต้องดู:

- Claude usage หรือ subscription
- Vercel plan และ bandwidth/function usage
- Supabase database, storage, auth, backup
- domain รายปี
- email/SMS provider
- payment fee
- API ที่เรียกตามจำนวนครั้ง

Prompt:

```
ช่วยทำ cost review สำหรับ project นี้
แยกเป็น:
- ค่าใช้จ่ายตอน prototype
- ค่าใช้จ่ายเมื่อมี user 100 / 1,000 / 10,000 คน
- cost driver ที่ต้อง monitor
- feature ไหนอาจทำให้ค่าใช้จ่ายพุ่ง
- วิธีลด cost โดยไม่ลดคุณค่าหลัก

ถ้าไม่มีข้อมูลราคาแน่นอน ให้บอกว่าเป็น estimate และต้องตรวจ pricing official อีกครั้ง
```

ถ้าจะใช้ตัวเลขราคา ต้องกลับไปดู official pricing ล่าสุดเสมอ เพราะราคาเปลี่ยนได้

ตัดสินใจว่าอะไรทำเองต่อ และอะไรควรส่งต่อ

หลัง launch คุณจะมี 3 ทางเลือก

1. ทำเองต่อ

เหมาะเมื่อ:

- app ยังเล็ก
- user น้อย
- data ไม่ sensitive มาก
- feature ใหม่ไม่ซับซ้อน
- คุณยัง test เองได้ครบ

2. ทำเอง + ให้ dev review เป็นรอบ

เหมาะเมื่อ:

- app เริ่มมี user จริง
- มี auth/database/permission
- มี feature ใหม่ทุกเดือน
- เริ่มมี revenue หรือทีมใช้จริง

3. ส่งต่อให้ developer/team

เหมาะเมื่อ:

- app เป็นระบบหลัก
- มี payment หรือข้อมูล sensitive
- มีหลาย role/permission
- ต้อง uptime สูง
- bug กระทบลูกค้าหรือรายได้
- คุณเริ่มไม่เข้าใจความเสี่ยงของ changes

ประโยคที่ควรจำ:

> การส่งต่อไม่ใช่ล้มเหลว
> มันคือสัญญาณว่า `prototype` เริ่มมีค่า

Workflow สุดท้ายของเล่มนี้

ถ้าต้องจำทั้งเล่มเป็น flow เดียว ให้จำอันนี้:

```
Idea
→ Spec
→ CLAUDE.md/context
→ Plan before edit
→ Build small
→ Test manually
→ Review diff
→ Commit
→ Preview deploy
→ Security check
→ Small launch
→ Feedback
→ Backlog
→ Iterate
→ Handoff when needed
```

นี่คือบทบาทของคุณ:

```
คุณไม่ต้องเป็น programmer
แต่คุณต้องเป็นคนกำกับ product, scope, test, risk และ decision
```

Claude Code คือคนลงมือที่เร็วมาก

คุณคือคนบอกว่าอะไรควรทำ อะไรไม่ควรทำ และเมื่อไหร่ต้องหยุด

Prompt รวมท้ายบท

ใช้ prompt นี้หลัง launch รอบแรก

```
ช่วยทำ post-launch review สำหรับ app นี้
ฉันเป็น non-coder

ข้อมูล:
- app ทำอะไร:
- เปิดให้ใครใช้:
- feedback ที่ได้รับ:
- bug ที่พบ:
- metrics หรือ observation:
- สิ่งที่กังวล:

ให้ช่วย:
1. สรุปสิ่งที่เรียนรู้
2. แยก feedback เป็น Bug / Improvement / New Feature / Question
3. จัด priority P0 / P1 / P2 / Later
4. เสนอ iteration รอบถัดไปไม่เกิน 3 งาน
5. บอก risk ก่อนแก้
6. แนะนำว่าควรทำเองต่อหรือควรให้ developer review

กติกา:
- อย่าเพิ่งแก้ code
- อย่าเพิ่ม scope เกิน feedback
- อธิบายแบบ non-coder
```

สรุป

Vibe coding ที่มีประโยชน์จริงไม่ใช่แค่สร้าง demo ได้เร็ว

แต่คือการสร้างของเล็ก ๆ ที่ใช้งานได้ ทดลองกับคนจริง เรียนรู้ และพัฒนาต่อแบบรับผิดชอบ

จำ 5 ข้อนี้:

1. Launch เล็กก่อน
2. เก็บ feedback แบบมีหลักฐาน
3. แยก bug, improvement, feature ให้ชัด
4. ใช้ commit, PR, preview และ rollback เป็น safety net
5. ส่งต่อให้ developer เมื่อความเสี่ยงเริ่มเกินสิ่งที่คุณควบคุมได้

ถ้าคุณทำได้ คุณไม่ใช่แค่คน “สั่ง AI เขียนโค้ด”

คุณกำลังทำหน้าที่ AI Product Director อย่างแท้จริง

$ APPENDIX A

Prompt Templates สำหรับ Vibe Coding

วิธีใช้ appendix นี้

ส่วนนี้คือ prompt ที่เอาไป copy ใช้กับ Claude Code ได้ทันที

ไม่จำเป็นต้องใช้ทุก prompt

ให้เลือกตามสถานการณ์:

```
เริ่มไอเดีย → ใช้ Prompt 1-4
กำลัง build → ใช้ Prompt 5-9
เจอ error → ใช้ Prompt 10-11
ก่อน deploy → ใช้ Prompt 12-14
หลัง launch → ใช้ Prompt 15-17
ส่งต่อ dev → ใช้ Prompt 18
```

กติกาสำคัญ:

- แทนที่ข้อความใน `[วงเล็บ]` ด้วยข้อมูลของคุณ
- อย่า paste secret/API key จริงลง prompt
- ถ้าเป็นงานใหญ่ ให้สั่ง “วางแผนก่อนแก้ไฟล์” เสมอ
- ถ้าไม่แน่ใจ ให้สั่ง “อธิบายแบบ non-coder”
- อย่าให้ Claude แก้หลายเรื่องพร้อมกันโดยไม่จำเป็น

Prompt 1 — สัมภาษณ์ไอเดียให้กลายเป็น project

ใช้เมื่อมีแค่ไอเดียกว้าง ๆ

```
ช่วยสัมภาษณ์ฉันเพื่อเปลี่ยนไอเดียนี้เป็น project ที่ Claude Code สร้างได้

ไอเดียของฉัน:
[อธิบายไอเดียสั้น ๆ]

บริบท:
- ฉันเป็น non-coder
- อยากทำ project ขนาดเล็กที่จบได้
- ยังไม่อยากทำระบบ production ซับซ้อน

ให้ถามทีละชุด ไม่เกิน 5 คำถามต่อรอบ
โฟกัสเรื่อง:
1. เป้าหมายของ project
2. user คือใคร
3. ปัญหาที่แก้
4. feature ที่จำเป็นจริง ๆ
5. feature ที่ยังไม่ควรทำ
6. data ที่ต้องเก็บ
7. ความเสี่ยงเรื่อง privacy/security

หลังถามครบ ให้สรุปเป็น project brief แบบสั้น
```

Prompt 2 — ทำ Project Brief หนึ่งหน้า

ใช้ก่อนเริ่มเขียน spec หรือ `CLAUDE.md`

```
ช่วยสร้าง project brief หนึ่งหน้าจากข้อมูลนี้

ข้อมูล:
[วางข้อมูล project]

รูปแบบที่ต้องการ:

# Project Brief

## Project name
[ชื่อ project]

## Goal
[เป้าหมายหลัก]

## Target user
[user คือใคร]

## Core problem
[ปัญหาที่แก้]

## Core user flow
1. ...
2. ...
3. ...

## Must-have features
- ...

## Nice-to-have features
- ...

## Non-goals
- สิ่งที่ยังไม่ทำตอนนี้

## Data collected
- field name
- reason to collect
- sensitivity level

## Risks
- security/privacy/product risks
```

```
เขียนให้ non-coder อ่านเข้าใจ
อย่าเพิ่ม feature เองถ้าไม่ได้จำเป็น
```

Prompt 3 — เขียน Product Spec ที่ AI เอาไป build ได้

ใช้หลังมี brief แล้ว

```
ช่วยแปลง brief นี้เป็น product spec สำหรับให้ Claude Code build

Brief:
[วาง project brief]

Spec ต้องมี:
1. Goal
2. Target users
3. Pages/screens
4. User flows
5. Data fields
6. Empty/loading/error states
7. Validation rules
8. Acceptance criteria
9. Non-goals
10. Security/privacy notes
11. Manual test checklist

กติกา:
- เขียนแบบชัดเจน ไม่ยืดยาว
- ถ้าข้อมูลไม่พอ ให้ใส่คำถามท้าย spec
- อย่าเดาเรื่อง sensitive data
- แยก must-have กับ later ให้ชัด
```

Prompt 4 — สร้าง `CLAUDE.md` สำหรับ project

ใช้เพื่อให้ Claude Code รู้บริบทและกติกาของ project

```
ช่วยสร้างไฟล์ CLAUDE.md สำหรับ project นี้

Project brief/spec:
[วาง brief หรือ spec]

CLAUDE.md ต้องมี:

# Project Overview
- app นี้ทำอะไร
- user คือใคร
- goal หลัก

# Tech Stack
- framework:
- database:
- deploy:
- package manager:

# Working Rules
- ก่อนแก้ไฟล์ใหญ่ ให้เสนอ plan ก่อน
- เปลี่ยนทีละ scope
- หลังแก้ให้สรุป diff แบบ non-coder
- ห้ามใส่ secret/API key จริงใน code
- ห้ามปิด security เช่น RLS เพื่อแก้ปัญหาถาวร

# Test Commands
- install:
- dev:
- build:
- lint/test ถ้ามี:

# Manual QA Checklist
- core flow
- mobile
- form validation
- error state
- security checks

# Non-goals
- สิ่งที่ไม่ทำตอนนี้

กติกา:
```

```
- อย่าใส่ secret จริง
- ถ้าไม่รู้ stack ให้ใส่ TODO
- เขียนให้สั้นและชัด
```

Prompt 5 — Explore ก่อนแก้ไฟล์

ใช้เมื่อเปิด project เดิม หรือไม่แน่ใจว่า code ทำงานอย่างไร

```
ช่วยสำรวจ project นี้ก่อน

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อ่านโครงสร้าง project เท่าที่จำเป็น
- สรุปแบบ non-coder

ช่วยบอก:
1. project นี้น่าจะทำอะไร
2. folder/file สำคัญมีอะไรบ้าง
3. stack ที่ใช้คืออะไร
4. command ที่น่าจะใช้รัน/test/build
5. จุดเสี่ยงหรือสิ่งที่ควรระวัง
6. คำถามที่ควรถามก่อนเริ่มแก้
```

Prompt 6 — Plan before editing

ใช้ก่อนให้ Claude Code แก้ feature หรือ bug ที่เกิน 1 ไฟล์

```
ช่วยวาง implementation plan ก่อนแก้ไฟล์

Task:
[อธิบายสิ่งที่ต้องการ]

Context:
[วาง spec/bug/feedback ที่เกี่ยวข้อง]

กติกา:
- อย่าเพิ่งแก้ไฟล์
- เสนอ plan เป็นขั้นตอน
- ระบุไฟล์ที่คาดว่าจะเปลี่ยน
- ระบุ risk หรือ side effect
- ระบุ manual test ที่ต้องทำหลังแก้
- ถ้า scope ใหญ่เกิน ให้เสนอ scope ที่เล็กลง
- ถ้าต้องตัดสินใจ ให้ถามฉันก่อน
```

Prompt 7 — Build landing page

ใช้สร้าง landing page หน้าแรก

```
ช่วย build landing page จาก brief นี้

Brief:
[วาง landing page brief]

ต้องมี section:
1. Hero พร้อม headline/subheadline/CTA
2. Problem
3. Benefits 3-5 ข้อ
4. How it works
5. Social proof หรือ placeholder ถ้ายังไม่มี
6. FAQ
7. Final CTA

กติกา:
- ทำให้ responsive/mobile-friendly
- copy ต้องชัด ไม่ hype เกินจริง
- อย่าเพิ่ม tracking/payment/form ที่ไม่ได้ขอ
- ใช้ style เรียบ อ่านง่าย
- หลังแก้ให้สรุปว่าเปลี่ยนไฟล์อะไร
- บอก manual test ที่ควรทำ
```

Prompt 8 — Revise UI/copy แบบคุม scope

ใช้ปรับหน้าที่ build แล้ว

```
ช่วยปรับ UI/copy ของหน้านี้ตาม feedback

Feedback:
[วาง feedback]

กติกา:
- อย่าเปลี่ยน layout ทั้งหมดถ้าไม่จำเป็น
- อย่าเพิ่ม feature ใหม่
- เน้น clarity, mobile readability, CTA
- ถ้ามีหลายทางเลือก ให้เสนอ 2 options ก่อน
- หลังแก้ให้สรุป diff แบบ non-coder

เป้าหมายของการแก้รอบนี้:
[เช่น hero ชัดขึ้น / CTA เด่นขึ้น / mobile spacing ดีขึ้น]
```

Prompt 9 — Supabase schema สำหรับ waitlist

ใช้ก่อนสร้าง database/table

```
ช่วยออกแบบ Supabase table สำหรับ waitlist app

App context:
[อธิบาย app]

Data ที่คิดว่าจะเก็บ:
[รายการ field]

ช่วยตอบเป็น:
1. table name ที่แนะนำ
2. columns พร้อม type แบบเข้าใจง่าย
3. field ไหนจำเป็น / optional
4. field ไหนไม่ควรเก็บตอนนี้
5. validation rules
6. RLS policy แนวคิดแบบ non-coder
7. public user ควรทำ action อะไรได้บ้าง
8. public user ไม่ควรทำอะไรได้บ้าง

กติกา:
- ใช้ data minimization
- ห้ามเสนอให้ปิด RLS เพื่อความง่าย
- อย่าใช้ service_role ใน frontend
```

Prompt 10 — ต่อ Supabase เข้ากับ app แบบปลอดภัย

ใช้ตอนให้ Claude Code เชื่อม form กับ database

```
ช่วยต่อ form นี้เข้ากับ Supabase

Context:
- form อยู่ที่: [หน้า/ไฟล์ ถ้ารู้]
- table: [ชื่อ table]
- fields: [field list]

กติกา:
- ใช้ publishable key เฉพาะที่ public ได้
- secret/service_role ห้ามอยู่ใน frontend/browser
- ใช้ environment variables
- เพิ่ม validation ขั้นพื้นฐาน
- เพิ่ม success/error state แบบ user อ่านเข้าใจ
- อย่า log secret หรือข้อมูลส่วนตัวเกินจำเป็น
- หลังแก้ให้บอกว่า env var ต้องตั้งชื่ออะไรบ้าง แต่ห้ามใส่ค่าจริง
- ระบุ manual test checklist
```

Prompt 11 — Debug report

ใช้เมื่อเจอ error

```
ฉันต้องการ debug ปัญหานี้แบบเป็นระบบ

## What happened
[เกิดอะไรขึ้น]

## Expected behavior
[ควรเกิดอะไร]

## Steps to reproduce
1. ...
2. ...
3. ...

## Error/log
[paste error/log เต็ม ๆ]

## Recent changes
[เปลี่ยนอะไรล่าสุด]

## Environment
- local / preview / production:
- command used:
- URL/page:

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อธิบาย error แบบ non-coder
- ชี้ log บรรทัดที่สำคัญ
- เสนอ root causes ที่เป็นไปได้ 2–3 ข้อ
- บอกวิธีตรวจแต่ละข้อ
- แนะนำ fix ที่เปลี่ยนน้อยที่สุด
```

Prompt 12 — แก้ bug แบบเปลี่ยนน้อยที่สุด

ใช้หลัง Claude วิเคราะห์ error แล้ว

```
จาก analysis ก่อนหน้า ช่วยแก้ bug นี้โดยเปลี่ยนน้อยที่สุด

กติกา:
- แก้เฉพาะ root cause ที่ยืนยันแล้ว
- อย่า refactor ส่วนอื่น
- อย่าเพิ่ม package ถ้าไม่จำเป็น
- ถ้าต้องเพิ่ม package ให้หยุดอธิบายก่อน
- หลังแก้ให้สรุปไฟล์ที่เปลี่ยน
- บอก command หรือ manual test ที่ต้องรัน
- บอก risk ที่ยังเหลือ
```

Prompt 13 — Review diff ก่อน commit

ใช้ก่อน commit ทุกครั้ง

```
ช่วย review changes ตอนนี้ก่อน commit

ตรวจว่า:
1. เปลี่ยนไฟล์อะไรบ้าง
2. แต่ละไฟล์เปลี่ยนเพื่ออะไร
3. มี change นอก scope ไหม
4. มี secret/API key/.env หลุดไหม
5. มี package ใหม่ไหม และจำเป็นไหม
6. มีผลต่อ security หรือ data permission ไหม
7. ต้อง test อะไรเพิ่มไหม
8. commit message ควรเป็นอะไร ใช้ Conventional Commits

กติกา:
- อธิบายแบบ non-coder
- ถ้ามี red flag ให้บอกให้หยุดก่อน commit
```

Prompt 14 — Security review ก่อน deploy

ใช้ก่อนเปิดให้คนอื่นใช้จริง

```
ช่วยทำ security review ก่อน deploy สำหรับ project นี้
ฉันเป็น non-coder ขอให้อธิบายแบบเข้าใจง่าย

ขอบเขต:
- frontend
- API/server actions
- Supabase/database/RLS
- environment variables
- authentication/authorization
- input validation
- logging
- dependencies
- deployment settings

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อย่า print secret จริง
- แยกผลเป็น Critical / High / Medium / Low
- ทุก issue ต้องมี: หลักฐาน, ความเสี่ยง, วิธีแก้ที่แนะนำ
- ถ้าไม่แน่ใจ ให้บอกว่าไม่แน่ใจ
- สรุปท้ายว่า launch ได้ไหมแบบ Green / Yellow / Red
```

Prompt 15 — Deploy readiness

ใช้ก่อน deploy preview หรือ production

```
ช่วยตรวจ deploy readiness ของ project นี้

แยก local / preview / production

ตรวจว่า:
1. command build คืออะไร
2. env var ที่จำเป็นมีอะไรบ้าง
3. ค่าไหน public ได้
4. ค่าไหนเป็น secret
5. production ยังมี test/dev value ไหม
6. database และ RLS พร้อมไหม
7. rollback plan คืออะไร
8. manual test หลัง deploy ต้องทำอะไร

กติกา:
- อย่า print ค่า secret จริง
- ถ้ามี blocker ให้แยกเป็นรายการชัดเจน
- สรุปเป็น Green / Yellow / Red
```

Prompt 16 — Small launch plan

ใช้ก่อนส่ง link ให้ user ชุดแรก

```
ช่วยวาง small launch plan สำหรับ app นี้
ฉันเป็น non-coder

App context:
[อธิบาย app]

ให้ตอบเป็น:
1. ควรเปิดให้ใครลองก่อน
2. จำนวน user แรกที่เหมาะสม
3. feature ไหนควรเปิด/ปิด
4. สิ่งที่ต้อง monitor
5. feedback ที่ควรถาม user
6. ถ้าเกิดปัญหา ต้องปิดหรือ rollback อย่างไร
7. เกณฑ์ว่าพร้อมขยาย audience หรือยัง

อย่าเสนอ launch ใหญ่ถ้ายังมี risk สำคัญ
```

Prompt 17 — Feedback to backlog

ใช้หลังได้ feedback จาก user

```
ฉันมี feedback จากผู้ใช้ชุดนี้
ช่วยแปลงเป็น backlog แบบเป็นระบบ

กติกา:
- อย่าเพิ่งเสนอ code
- รวม feedback ที่ซ้ำกัน
- แยกเป็น Bug / Improvement / New Feature / Question
- ให้ priority เป็น P0 / P1 / P2 / Later
- บอกเหตุผลของ priority แบบ non-coder
- ถ้าข้อมูลไม่พอ ให้ระบุว่าต้องถาม user อะไรเพิ่ม

Feedback:
[paste feedback]
```

Prompt 18 — Post-launch review

ใช้หลัง launch รอบแรก

```
ช่วยทำ post-launch review สำหรับ app นี้
ฉันเป็น non-coder

ข้อมูล:
- app ทำอะไร:
- เปิดให้ใครใช้:
- feedback ที่ได้รับ:
- bug ที่พบ:
- metrics หรือ observation:
- สิ่งที่กังวล:

ให้ช่วย:
1. สรุปสิ่งที่เรียนรู้
2. แยก feedback เป็น Bug / Improvement / New Feature / Question
3. จัด priority P0 / P1 / P2 / Later
4. เสนอ iteration รอบถัดไปไม่เกิน 3 งาน
5. บอก risk ก่อนแก้
6. แนะนำว่าควรทำเองต่อหรือควรให้ developer review

กติกา:
- อย่าเพิ่งแก้ code
- อย่าเพิ่ม scope เกิน feedback
- อธิบายแบบ non-coder
```

Prompt 19 — Handoff document สำหรับส่งต่อ developer

ใช้เมื่อ project เริ่มจริงจัง หรือคุณต้องให้ dev ช่วยต่อ

```
ช่วยสร้าง handoff doc จาก project นี้
ฉันจะส่งต่อให้ developer

ต้องมี:

# Handoff Doc

## App purpose
[app นี้ทำอะไร เพื่อใคร]

## Current status
[prototype / beta / production]

## Main user flows
1. ...
2. ...

## Tech stack
- Frontend:
- Backend/API:
- Database:
- Deploy:

## Environment variables
| Name | Public/Secret | Used for | Where set |
|---|---|---|---|

## Database and permissions
- Tables:
- RLS policies:
- Roles:

## How to run locally
[คำสั่งและขั้นตอน]

## How to deploy
[branch, Vercel project, env notes]

## Known issues
[รายการ]

## Backlog
```

```
[P0/P1/P2/Later]

## Risks
[security, data, cost, operational]

กติกา:
- อย่าใส่ค่า secret จริง
- ถ้าไม่รู้ข้อมูลส่วนไหน ให้ใส่ TODO
- อธิบาย architecture แบบภาพรวม
- สรุป risks และ known issues ตรง ๆ
```

Prompt 20 — Monthly maintenance review

ใช้เดือนละครั้งถ้ายังดูแล app เอง

```
ช่วยทำ monthly maintenance review สำหรับ project นี้
ฉันเป็น non-coder

ตรวจเรื่อง:
1. account/admin access และ 2FA
2. secret/API key ที่ไม่ใช้แล้ว
3. Supabase RLS/policies
4. package/dependency risk
5. logs มีข้อมูล sensitive ไหม
6. backup/export ที่จำเป็น
7. Vercel/Supabase billing
8. domain/SSL/deployment
9. backlog priority
10. unused project/deployment/integration

กติกา:
- ให้ผลเป็น Green / Yellow / Red
- เสนอ action ไม่เกิน 5 ข้อ
- แยกสิ่งที่ต้องทำทันทีออกจากสิ่งที่รอได้
- อย่า print ค่า secret จริง
```

Prompt 21 — Cost review

ใช้เมื่อเริ่มมี user จริง หรือก่อนเปิด public

```
ช่วยทำ cost review สำหรับ project นี้

Context:
[อธิบาย app, services ที่ใช้, จำนวน user คาดการณ์]

แยกเป็น:
1. ค่าใช้จ่ายตอน prototype
2. ค่าใช้จ่ายเมื่อมี user 100 / 1,000 / 10,000 คน
3. cost driver ที่ต้อง monitor
4. feature ไหนอาจทำให้ค่าใช้จ่ายพุ่ง
5. วิธีลด cost โดยไม่ลดคุณค่าหลัก
6. จุดที่ต้องกลับไปเช็กราคา official ล่าสุด

กติกา:
- ถ้าไม่มีข้อมูลราคาแน่นอน ให้บอกว่าเป็น estimate
- อย่าคิดราคาเองแบบมั่นใจถ้าไม่ได้ตรวจ official pricing ล่าสุด
```

Prompt 22 — Stop and ask when scope grows

ใช้ใส่ท้าย prompt ใหญ่ ๆ เพื่อกัน AI ขยายงานเอง

```
ข้อจำกัดสำคัญ:
ถ้าระหว่างทำงานพบว่า scope ใหญ่ขึ้น เช่น
- ต้องเพิ่ม package ใหม่
- ต้องแก้ database schema
- ต้องเปลี่ยน auth/permission
- ต้องแตะไฟล์เกินที่คาด
- ต้องเปลี่ยน architecture
- ต้องใช้ secret/admin key

ให้หยุดและถามฉันก่อน
อย่าตัดสินใจเอง
```

Prompt 23 — Explain like I am a non-coder

ใช้ทุกครั้งที่คำตอบเริ่ม technical เกินไป

```
ช่วยอธิบายใหม่แบบ non-coder

ต้องตอบโดย:
- ใช้ภาษาคนทั่วไป
- ไม่ลงรายละเอียด code เกินจำเป็น
- เปรียบเทียบกับสิ่งที่เข้าใจง่าย
- บอกว่าฉันต้องตัดสินใจอะไร
- บอกว่าความเสี่ยงคืออะไร
- ถ้ามีศัพท์ technical ให้แปลเป็นภาษาง่าย ๆ
```

Prompt 24 — Final pre-launch checklist รวมทุกอย่าง

ใช้เป็นด่านสุดท้าย

```
ทำ final pre-launch checklist ให้ project นี้
ฉันเป็น non-coder

ตรวจ:
1. product flow
2. mobile/responsive
3. form validation
4. empty/loading/error states
5. secrets/env vars
6. Supabase publishable vs secret/service_role keys
7. RLS และ database policies
8. auth vs authorization
9. logs/error messages
10. dependencies/packages
11. preview/production config
12. rollback plan
13. feedback collection
14. owner/maintenance plan

กติกา:
- อย่าแก้ไฟล์ก่อน
- อย่า print secret จริง
- ถ้าเจอ critical issue ให้หยุดและอธิบายก่อน
- ทำ report เป็นตาราง
- ให้ launch gate เป็น Green / Yellow / Red
- เสนอ next steps ไม่เกิน 5 ข้อ
```

$ APPENDIX B

Starter Files สำหรับเริ่มโปรเจกต์

วิธีใช้ appendix นี้

ส่วนนี้คือไฟล์ตั้งต้นที่เอาไป copy ใส่ project ได้

เหมาะกับคนที่ไม่ใช่ programmer แต่อยากให้ Claude Code ทำงานเป็นระบบตั้งแต่วันแรก

คุณไม่จำเป็นต้องใช้ทุกไฟล์

ชุดเริ่มต้นที่แนะนำที่สุดคือ:

```
CLAUDE.md
.env.example
.gitignore
README.md
QA_CHECKLIST.md
SECURITY_CHECKLIST.md
HANDOFF.md
```

กติกาสำคัญ:

- อย่าใส่ secret/API key จริงในไฟล์ตัวอย่าง
- ถ้าไม่รู้ค่าไหน ให้เขียน `TODO`
- ทำให้ไฟล์สั้นพอที่คุณอ่านเองได้
- ให้ Claude ช่วยเติมได้ แต่คุณต้อง review ก่อน commit

โครงสร้าง project ที่แนะนำ

สำหรับ project เล็ก ๆ ใช้โครงแบบนี้พอ

```
my-project/
  CLAUDE.md
  README.md
  .env.example
  .gitignore
  docs/
    QA_CHECKLIST.md
    SECURITY_CHECKLIST.md
    HANDOFF.md
    DECISIONS.md
```

ถ้า project ยังเล็กมาก จะใส่ checklist ทั้งหมดไว้ใน README ก่อนก็ได้

แต่เมื่อเริ่มมี database, deploy, user จริง หรือคนอื่นมาช่วย ควรแยกเป็นไฟล์ใน `docs/`

Starter 1 — `CLAUDE.md`

ใช้ไฟล์นี้เป็น context หลักให้ Claude Code อ่านก่อนทำงาน

MARKDOWN

```
# CLAUDE.md

## Project Overview

This project is:

- Name: [project name]
- Purpose: [what this app does]
- Target user: [who uses it]
- Current stage: [prototype / beta / production]

In plain language:

[อธิบาย app นี้ 3-5 บรรทัด]

## Product Goal

The main goal is:

[เป้าหมายหลักของ project]

Success means:

- [acceptance criteria 1]
- [acceptance criteria 2]
- [acceptance criteria 3]

## Non-Goals

Do not build these unless explicitly requested:

- [สิ่งที่ยังไม่ทำ]
- [สิ่งที่อยู่นอก scope]
- [feature ที่ไว้ later]

## Tech Stack

- Framework: [Next.js / Vite / other]
- Styling: [Tailwind / CSS / other]
- Database: [Supabase / none / other]
- Auth: [Supabase Auth / none / other]
- Deploy: [Vercel / other]
- Package manager: [npm / pnpm / yarn]

If unsure, ask before changing stack or adding major dependencies.

## Working Rules

When working on this project:
```

````
1. Explore first, then plan, then edit.
2. Do not make large changes without a plan.
3. Keep changes scoped to the requested task.
4. Do not add new packages unless necessary.
5. If a package is needed, explain why before installing.
6. Do not put secrets or real API keys in code.
7. Do not commit `.env` files.
8. Do not disable security controls such as RLS as a permanent fix.
9. After changes, summarize the diff in non-technical language.
10. Provide manual test steps after every meaningful change.

## Security Rules

Never expose these in frontend/browser code:

- secret keys
- service role keys
- database passwords
- admin tokens
- webhook secrets

For Supabase:

- publishable key may be used in public client code
- secret/service_role keys must stay server-side only
- RLS should be enabled for tables accessed from the client
- public users should only have the minimum permissions needed

## Commands

Common commands:

```bash
# install dependencies
[TODO: npm install]

# run dev server
[TODO: npm run dev]

# build
[TODO: npm run build]

# lint/test if available
[TODO]
````

Before assuming a command, inspect `package.json`.
````

Manual QA Checklist

Before saying a task is done, check:

- [ ] Main user flow works
- [ ] Mobile layout works
- [ ] Empty state works
- [ ] Error state works
- [ ] Form validation works
- [ ] No secret is printed or exposed
- [ ] Build passes if build command exists

Communication Style

Explain results for a non-coder:

- what changed
- why it changed
- what to test
- what risk remains
- what decision is needed from the user

````
---

## Starter 2 — `.env.example`

ไฟล์นี้บอกว่าต้องตั้งค่า env var อะไรบ้าง แต่ห้ามใส่ค่าจริง

```bash
# .env.example
# Copy this file to .env.local for local development.
# Do not put real secrets in this example file.

# App
NEXT_PUBLIC_APP_URL=http://localhost:3000

# Supabase public values
NEXT_PUBLIC_SUPABASE_URL=your-supabase-project-url
NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY=your-supabase-publishable-key

# Server-only secrets
# Do NOT expose these in frontend/browser code.
SUPABASE_SECRET_KEY=your-server-only-secret-key
SUPABASE_SERVICE_ROLE_KEY=do-not-use-in-browser

# Optional email provider
EMAIL_FROM=hello@example.com
EMAIL_API_KEY=your-email-api-key

# Optional Stripe
STRIPE_SECRET_KEY=your-stripe-secret-key
STRIPE_WEBHOOK_SECRET=your-stripe-webhook-secret
NEXT_PUBLIC_STRIPE_PUBLISHABLE_KEY=your-stripe-publishable-key
```
````

ถ้า project ไม่ใช้ Supabase, Email หรือ Stripe ให้ลบส่วนที่ไม่ใช้

อย่าเก็บ env var เยอะเกินจำเป็น

Starter 3 — `.gitignore`

ใช้กันไม่ให้ commit ไฟล์ที่ไม่ควรอยู่ใน GitHub

• GITIGNORE

```
# dependencies
node_modules/

# build outputs
.next/
dist/
build/
out/

# local env files
.env
.env.local
.env.development.local
.env.test.local
.env.production.local

# logs
npm-debug.log*
yarn-debug.log*
yarn-error.log*
pnpm-debug.log*

# OS files
.DS_Store
Thumbs.db

# editor files
.vscode/
.idea/

# temporary files
.tmp/
tmp/
.cache/

# screenshots or exports that may contain secrets
secret-screenshots/
private-exports/
```

หมายเหตุ: บางทีมอาจ commit `.vscode/settings.json` ได้ แต่สำหรับมือใหม่ ตัดออกก่อนปลอดภัยกว่า

Starter 4 — `README.md` สำหรับ project

README คือหน้าแรกที่อธิบาย project ให้คุณในอนาคต และให้คนอื่นที่มาช่วยเข้าใจเร็ว

```markdown
# [Project Name]

## What this app does

[อธิบาย app นี้แบบภาษาคน 3-5 บรรทัด]

## Current status

- Stage: [prototype / beta / production]
- Owner: [ชื่อคนรับผิดชอบ]
- Last reviewed: [date]

## Main user flows

1. [user ทำอะไร step 1]
2. [step 2]
3. [step 3]

## Tech stack

- Frontend: [TODO]
- Database: [TODO]
- Auth: [TODO]
- Deploy: [TODO]

## Getting started

```bash
# install
[TODO: npm install]

# run locally
[TODO: npm run dev]
```

Open:

> `http://localhost:3000`
```

Environment variables

Copy `.env.example` to `.env.local` and fill in values.

Never commit `.env.local`.

Required variables:

- `NEXT_PUBLIC_APP_URL`
- `NEXT_PUBLIC_SUPABASE_URL`
- `NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY`
- [TODO]

Deployment

- Platform: [Vercel / other]
- Production branch: [main]
- Preview deployments: [yes/no]
- Production URL: [TODO]

Manual QA

Before deploy, run through:

- `docs/QA_CHECKLIST.md`
- `docs/SECURITY_CHECKLIST.md`

Known issues

- [TODO]

Backlog

- [TODO]

Starter 5 — `docs/QA_CHECKLIST.md`

ใช้ตรวจงานแบบคนใช้งานจริง ไม่ต้องอ่าน code

```markdown
# QA Checklist

## Project

- Project name: [name]
- Date tested: [date]
- Tester: [name]
- Environment: [local / preview / production]
- URL: [URL]

## Smoke test

- [ ] App opens without blank screen
- [ ] Main page loads within a reasonable time
- [ ] Navigation links work
- [ ] No obvious broken layout
- [ ] No console/error screen shown to user

## Main user flow

Flow tested:

[อธิบาย flow เช่น user กรอก waitlist form]

Steps:

1. [step]
2. [step]
3. [step]

Result:

- [ ] Works as expected
- [ ] Fails
- [ ] Needs review
```

```
Notes:

[notes]

## Form validation

- [ ] Required fields cannot be empty
- [ ] Invalid email is rejected
- [ ] Very long text does not break layout
- [ ] Duplicate submission behaves correctly
- [ ] Success message is clear
- [ ] Error message is clear and safe

## Mobile check

Test device/browser:

- [ ] iPhone size
- [ ] Android size
- [ ] Desktop narrow window

Check:

- [ ] Text readable
- [ ] Buttons tappable
- [ ] No horizontal scroll
- [ ] Forms usable
- [ ] CTA visible

## Empty/loading/error states

- [ ] Empty state is understandable
- [ ] Loading state appears when needed
- [ ] Error state tells user what to do next
- [ ] Error does not expose technical secret/config

## Data check

- [ ] Submitted data appears in the right place
- [ ] Data fields match expected format
- [ ] User cannot see data they should not see
```

```
- [ ] Test data is clearly marked or removed

## Final decision

- [ ] Pass
- [ ] Pass with minor issues
- [ ] Blocked

Blocking issues:

- [TODO]
```

Starter 6 — `docs/SECURITY_CHECKLIST.md`

ใช้เป็น checklist ก่อน deploy หรือก่อนเปิดให้ user จริง

```markdown
```

```
# Security Checklist

## Project

- Project name: [name]
- Date reviewed: [date]
- Reviewer: [name]
- Environment: [preview / production]

## Data minimization

- [ ] เก็บเฉพาะข้อมูลที่จำเป็น
- [ ] ไม่มีข้อมูล sensitive ที่ไม่จำเป็น
- [ ] มีเหตุผลชัดเจนสำหรับแต่ละ field
- [ ] test data ถูกลบหรือแยกจาก production

## Secrets and environment variables

- [ ] ไม่มี `.env` ถูก commit
- [ ] ไม่มี secret/API key จริงใน GitHub
- [ ] ไม่มี secret ใน frontend/browser code
- [ ] `.env.example` ใช้ placeholder เท่านั้น
- [ ] production env vars ตั้งครบ
- [ ] key ที่ไม่ใช้ถูกลบหรือ rotate แล้ว

## Supabase / Database

- [ ] RLS เปิดสำหรับ table ที่ client เข้าถึง
- [ ] public user อ่านข้อมูลคนอื่นไม่ได้
- [ ] public user update/delete ข้อมูลไม่ได้ถ้าไม่จำเป็น
- [ ] insert policy แคบเท่าที่ feature ต้องใช้
- [ ] service_role/secret key ไม่อยู่ใน browser
- [ ] dashboard/admin route ไม่ public

## Auth and authorization

- [ ] login flow ใช้งานได้
- [ ] role/permission ชัดเจน
- [ ] user เห็นเฉพาะข้อมูลที่ควรเห็น
- [ ] admin-only page ถูกป้องกัน
- [ ] logout ทำงานถูกต้อง

## Input validation

- [ ] required field validate แล้ว
- [ ] email/URL format validate แล้ว
- [ ] length limit มีแล้ว
- [ ] unexpected input ไม่ทำให้ app พัง
- [ ] error message ไม่เปิดเผยระบบภายใน
```

```
## Dependencies

- [ ] ไม่มี package ใหม่ที่ไม่จำเป็น
- [ ] package ใหม่มีเหตุผลชัดเจน
- [ ] ไม่มี dependency ที่ไม่ใช้แล้วถอดได้ง่าย

## Logging

- [ ] ไม่ log password/token/secret
- [ ] ไม่ log personal data เกินจำเป็น
- [ ] production error message ปลอดภัยสำหรับ user

## Deploy and rollback

- [ ] preview URL ถูก test แล้ว
- [ ] production env var ถูกต้อง
- [ ] มี rollback plan
- [ ] owner รู้วิธีปิด app ชั่วคราวถ้าจำเป็น

## Launch gate

Choose one:

- [ ] Green — launch ได้แบบจำกัด scope
- [ ] Yellow — launch เฉพาะ beta/waitlist
- [ ] Red — ห้าม launch จนกว่าจะแก้ blocker

Blockers:

- [TODO]
```

Starter 7 — `docs/HANDOFF.md`

ใช้ตอนส่งต่อให้ developer หรือทีม technical

MARKDOWN

```
# Handoff Document

## App purpose

[app นี้ทำอะไร เพื่อใคร]

## Current status

- Stage: [prototype / beta / production]
- Owner: [name]
- Last updated: [date]
- Production URL: [URL]
- Repository: [URL]

## Main user flows

1. [flow 1]
2. [flow 2]
3. [flow 3]

## Tech stack

- Frontend: [TODO]
- Backend/API: [TODO]
- Database: [TODO]
- Auth: [TODO]
- Deploy: [TODO]
- Third-party services: [TODO]

## How to run locally

```bash
# install
[TODO]

# run
[TODO]

# build
[TODO]
```

Environment variables

Do not put real values in this document.
```

Name	Public/Secret	Used for	Where set	Notes
`NEXT_PUBLIC_APP_URL`	Public	App URL	Vercel/local	TODO
`NEXT_PUBLIC_SUPABASE_URL`	Public	Supabase	Vercel/local	TODO
`NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY`	Public	Supabase client	Vercel/local	TODO
`SUPABASE_SECRET_KEY`	Secret	Server-side only	Vercel/local	TODO

Database

Tables:

-

Important fields:

-

Permissions / RLS

Explain in plain language:

- Public users can: [TODO]
- Logged-in users can: [TODO]
- Admins can: [TODO]

Known permission risks:

- [TODO]

Deployment

- Platform: [Vercel]
- Production branch: [main]
- Preview deployments: [yes/no]

- Rollback process: [summary]

Known issues

Issue	Impact	Priority	Notes
TODO	TODO	TODO	TODO

Backlog

P0

- [TODO]

P1

- [TODO]

P2 / Later

- [TODO]

Security notes

- [TODO]

Product decisions

Link to `docs/DECISIONS.md` if available.

````
---

## Starter 8 — `docs/DECISIONS.md`

ใช้เก็บเหตุผลสำคัญว่าทำไมเลือกแบบนี้

```markdown
# Decision Log

ใช้ไฟล์นี้บันทึก decision สำคัญของ project

## Template

### [YYYY-MM-DD] — [Decision title]

Decision:

[ตัดสินใจอะไร]

Context:

[ทำไมต้องตัดสินใจ]

Options considered:

1. [option 1]
2. [option 2]
3. [option 3]

Why this option:

[เหตุผล]

Risks:

[ความเสี่ยง]

Review later:

[ต้องกลับมาดูเมื่อไหร่]

---
````
````

```
## Decisions

### [YYYY-MM-DD] — Example: Use Supabase for waitlist database

Decision:

Use Supabase as the database for waitlist submissions.

Context:

The app needs to store emails and use cases from early users.

Options considered:

1. Google Sheet
2. Supabase
3. Custom backend

Why this option:

Supabase gives a real database and can support RLS when the app grows.

Risks:

Need to configure RLS correctly and avoid exposing secret keys.

Review later:

Review before opening the app to public users.
```

Starter 9 — `docs/FEEDBACK.md`

ใช้เก็บ feedback จาก user ชุดแรก ถ้ายังไม่ใช้ GitHub Issues หรือ Notion

MARKDOWN

```
# Feedback Log

## How to write feedback

Each feedback item should include:

- Who reported it
- What they were trying to do
- What happened
- What they expected
- Evidence if available
- Impact

---

## Template

### [YYYY-MM-DD] — [Short title]

Reported by:

[name / role]

Page/URL:

[URL]

What were they trying to do?

[description]

What happened?

[description]

Expected result:

[description]

Evidence:

[screenshot / error / steps]

Impact:

[low / medium / high]

Category:
```

```
[Bug / Improvement / New Feature / Question]

Priority:

[P0 / P1 / P2 / Later]

Status:

[New / Planned / In progress / Done / Won't do]
```

Starter 10 — `docs/LAUNCH_CHECKLIST.md`

ใช้ก่อนเปิด beta หรือ public

MARKDOWN

```
# Launch Checklist

## Launch scope

- Launch type: [private beta / public beta / production]
- Audience: [who will use it]
- Expected users: [number]
- Launch date: [date]
- Owner: [name]

## Product readiness

- [ ] Main flow works
- [ ] Copy is understandable
- [ ] Mobile is usable
- [ ] Feedback channel is ready
- [ ] Known limitations are written down

## Technical readiness

- [ ] Build passes
- [ ] Preview tested
- [ ] Production env vars set
- [ ] Database ready
- [ ] Rollback plan ready

## Security readiness

- [ ] Security checklist reviewed
- [ ] No secrets exposed
- [ ] RLS/permissions reviewed
- [ ] Error messages safe
- [ ] Logs safe

## Support readiness

- [ ] Someone will monitor after launch
- [ ] User contact/support channel ready
- [ ] FAQ or basic instructions ready
- [ ] Incident plan known

## Decision

- [ ] Launch
- [ ] Launch to smaller group only
- [ ] Do not launch yet

Notes:
```

```
[TODO]
```

ให้ Claude ช่วยสร้างไฟล์พวกนี้ใน project

เมื่อคุณเปิด project จริงใน Claude Code ให้ใช้ prompt นี้

```
ช่วยสร้าง starter documentation files สำหรับ project นี้

ไฟล์ที่ต้องการ:
- CLAUDE.md
- .env.example
- .gitignore
- README.md
- docs/QA_CHECKLIST.md
- docs/SECURITY_CHECKLIST.md
- docs/HANDOFF.md
- docs/DECISIONS.md
- docs/FEEDBACK.md
- docs/LAUNCH_CHECKLIST.md

กติกา:
- ใช้ข้อมูลจาก project ปัจจุบันเท่าที่อ่านได้
- ถ้าไม่รู้ ให้ใส่ TODO แทนการเดา
- อย่าใส่ secret/API key จริง
- อย่า overwrite ไฟล์เดิมโดยไม่บอกก่อน
- หลังสร้าง ให้สรุปว่าแต่ละไฟล์มีไว้ทำอะไร
```

ชุดไฟล์ขั้นต่ำถ้าอยากให้สั้นมาก

ถ้าไม่อยากมีไฟล์เยอะ ให้เริ่มแค่นี้:

```
CLAUDE.md
.env.example
.gitignore
README.md
```

แล้วรวม checklist ไว้ท้าย README ก่อน

เมื่อ project เริ่มมี user จริง ค่อยแยกเพิ่ม:

```
docs/QA_CHECKLIST.md
docs/SECURITY_CHECKLIST.md
docs/HANDOFF.md
```

อย่าให้ documentation กลายเป็นงานใหญ่จนไม่เริ่ม

เป้าหมายคือให้คุณและ Claude Code ไม่ลืมกติกาสำคัญ

$ APPENDIX D

Vercel Deploy Playbook สำหรับ Non-Coder

ใช้ appendix นี้เมื่อไหร่

ใช้เมื่อคุณมีเว็บหรือ app ที่รันในเครื่องได้แล้ว และอยากเอาขึ้นออนไลน์ด้วย Vercel

เป้าหมายของ appendix นี้คือจับมือทำแบบ practical:

```
local works → GitHub ready → import to Vercel → set env vars → deploy →
test → fix logs → preview → rollback if needed
```

ไม่ใช่สอน Vercel ทุก feature

เราจะใช้เฉพาะสิ่งที่ non-coder ต้องรู้เพื่อ deploy project เล็ก ๆ อย่างปลอดภัย

ภาพรวมแบบภาษาคน

Vercel คือ platform สำหรับเอาเว็บขึ้นออนไลน์

ถ้าเชื่อมกับ GitHub แล้ว workflow จะประมาณนี้:

```
push code ไป GitHub
→ Vercel เห็น code ใหม่
→ Vercel build project
→ ถ้าผ่าน จะได้ URL
→ ถ้า push เข้า main จะเป็น production
→ ถ้า push เข้า branch/PR จะเป็น preview
```

คำสำคัญ:

คำ	แปลแบบง่าย
Local	เครื่องคุณเอง
Preview	เว็บทดสอบก่อนของจริง
Production	เว็บจริงที่ user ใช้
Build	ขั้นตอนแปลง project ให้พร้อม deploy
Environment variable	ค่า config เช่น URL/key ที่ไม่ควร hardcode
Deployment	version หนึ่งของเว็บที่ Vercel build แล้ว
Rollback	ย้อน production กลับ version ก่อนหน้า

ก่อน deploy: เช็กว่า project พร้อมไหม

อย่าเริ่มจากหน้า Vercel ทันที

เช็กใน project ก่อน

1. Project รันในเครื่องได้

ควรเปิดเว็บ local ได้ เช่น:

```
http://localhost:3000
```

ถ้ายังรันในเครื่องไม่ได้ อย่าเพิ่ง deploy

ให้แก้ local ก่อน เพราะ deploy จะ debug ยากกว่า

2. มี Git repository แล้ว

ควรมี repo ที่ commit แล้ว

ตรวจด้วย prompt นี้:

```
ช่วยตรวจว่า project นี้พร้อม push ไป GitHub และ deploy ไป Vercel ไหม
ตรวจ:
- package.json มี scripts อะไร
- build command คืออะไร
- มีไฟล์ .env หรือ secret ถูก track ไหม
- มี .gitignore ครบไหม
- มี README หรือ deployment notes ไหม
อย่าเพิ่งแก้ไฟล์ ให้รายงานก่อน
```

3. ไม่มี secret อยู่ใน code

ก่อน push ไป GitHub ให้เช็ก:

```
.env
.env.local
API key จริง
service role key
database password
secret token
```

ถ้าเจอ secret ใน Git history หรือ GitHub แล้ว ไม่ใช่แค่ลบออกจากไฟล์

ควร rotate key ด้วย

Step 1 — Push project ไป GitHub

ถ้าคุณยังไม่ใช้ GitHub ให้คิดง่าย ๆ ว่า GitHub คือที่เก็บ code บน cloud

Vercel จะดึง code จาก GitHub ไป deploy

สิ่งที่ต้องมี

- GitHub account
- repository สำหรับ project นี้
- project ถูก commit แล้ว

ถาม Claude ให้ช่วยตรวจ

```
ช่วยเตรียม project นี้ก่อน push ไป GitHub

กติกา:
- อย่า push เอง
- ตรวจ .gitignore
- ตรวจว่าไม่มี .env หรือ secret ถูก track
- สรุปไฟล์ที่ควร commit
- เสนอ commit message
- บอกคำสั่งที่ฉันต้องรันทีละบรรทัด
```

ถ้าคุณไม่ชำนาญ command line ให้ Claude อธิบาย command ทีละบรรทัดก่อนรัน

Step 2 — Import repository เข้า Vercel

หลัง code อยู่ใน GitHub แล้ว ไปที่ Vercel Dashboard

flow ทั่วไป:

```
Vercel Dashboard
→ Add New / New Project
→ Import Git Repository
→ เลือก repo
→ Configure Project
→ Deploy
```

สิ่งที่ต้องดูในหน้า Configure Project:

ช่อง	ควรทำอะไร
Project Name	ตั้งชื่ออ่านง่าย
Framework Preset	ให้ Vercel detect ก่อน ถ้าเป็น Next.js มัก detect ได้
Root Directory	ถ้า project อยู่ root ใช้ default; ถ้าอยู่ subfolder ต้องเลือกให้ถูก
Build Command	ใช้ตาม `package.json` เช่น `npm run build`

ช่อง	ควรทำอะไร
Output Directory	ปกติปล่อย default ถ้า framework detect ถูก
Environment Variables	ใส่ค่าที่ app ต้องใช้

สำหรับ Next.js บน Vercel มัก zero-config ถ้า project structure ปกติ

แต่ถ้า deploy fail อย่าเดา ให้ดู build log

Step 3 — ตรวจ build settings จาก `package.json`

ก่อนแก้ build settings ใน Vercel ให้ดู `package.json`

ตัวอย่าง:

```json
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  }
}
```

แปลว่า:

Script	ใช้ทำอะไร
`dev`	รันในเครื่อง
`build`	ให้ Vercel build
`start`	รัน production server บางกรณี

Prompt:

```
ช่วยอ่าน package.json แล้วอธิบายว่า Vercel ควรใช้ build command อะไร
ถ้า framework preset ควรเป็นอะไรให้บอกด้วย
อย่าแก้ไฟล์
```

ถ้า Claude บอกให้เปลี่ยน build command ให้ถามต่อ:

```
มีหลักฐานจาก package.json หรือ error log ตรงไหนว่าต้องเปลี่ยน build command
```

Step 4 — ใส่ Environment Variables ใน Vercel

Environment variables คือค่าที่ app ต้องใช้ แต่ไม่ควรเขียน hardcode ใน code

เช่น:

```
NEXT_PUBLIC_SUPABASE_URL
NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY
SUPABASE_SECRET_KEY
STRIPE_SECRET_KEY
```

ใน Vercel ให้ใส่ที่ project settings ของ project นั้น

จำหลักนี้:

```
local ใช้ .env.local
Vercel ใช้ Environment Variables ใน dashboard
```

Public vs Secret

ประเภท	ตัวอย่าง	อยู่ frontend ได้ไหม
Public	`NEXT_PUBLIC_SUPABASE_URL`	ได้
Public	`NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY`	ได้ แต่ต้องมี RLS/policy
Secret	`SUPABASE_SECRET_KEY`	ไม่ได้

ประเภท	ตัวอย่าง	อยู่ frontend ได้ไหม
Secret	`SUPABASE_SERVICE_ROLE_KEY`	ไม่ได้
Secret	`STRIPE_SECRET_KEY`	ไม่ได้

คำว่า `NEXT_PUBLIC_` ใน Next.js แปลว่าค่านี้อาจถูกใช้ฝั่ง browser ได้

อย่าใส่ secret จริงในตัวแปรที่ขึ้นต้นด้วย `NEXT_PUBLIC_`

เลือก environment ให้ถูก

Vercel มี environment หลัก:

```
Development
Preview
Production
```

สำหรับมือใหม่ ให้เริ่มแบบนี้:

- ถ้า app ต้องใช้ค่าบน production URL ให้ใส่ Production
- ถ้าอยาก test บน branch/PR ก่อน ให้ใส่ Preview ด้วย
- ถ้าใช้ Vercel CLI local ค่อยสนใจ Development

ข้อสำคัญจาก Vercel docs: การเปลี่ยน environment variable จะมีผลกับ deployment ใหม่ ไม่ใช่ deployment เก่าทันที

ดังนั้นถ้าเพิ่มหรือแก้ env var แล้วเว็บยังพัง ให้ redeploy ใหม่

Prompt:

```
ช่วยทำ env var checklist สำหรับ deploy ไป Vercel

ตอบเป็นตาราง:
- variable name
- public หรือ secret
- ใช้ทำอะไร
- ต้องใส่ใน Preview ไหม
- ต้องใส่ใน Production ไหม
- ห้ามอยู่ frontend ไหม

กติกา:
- อย่า print ค่า secret จริง
- ถ้าไม่แน่ใจ ให้บอกว่าไม่แน่ใจ
```

Step 5 — Deploy ครั้งแรก

หลัง import repo และตั้งค่า env var แล้ว กด deploy

ระหว่าง deploy ให้ดูสถานะ:

```
Queued / Building / Ready / Error
```

ถ้า Ready:

- เปิด URL ที่ Vercel ให้มา
- test flow หลัก
- test mobile
- ถ้ามี form ให้ submit test

ถ้า Error:

- อย่ากดแก้มั่ว
- เปิด build log
- copy error ส่วนสำคัญให้ Claude วิเคราะห์

Prompt:

```
Vercel deploy fail
ช่วยวิเคราะห์ build log นี้แบบ non-coder

Expected:
- build ผ่าน
- ได้ deployment URL

Build log:
[paste log]

กติกา:
- อย่าเพิ่งแก้ไฟล์
- ชี้บรรทัดสำคัญใน log
- แยกสาเหตุที่เป็นไปได้ 2-3 แบบ
- บอกวิธีตรวจแต่ละแบบ
- เสนอ fix ที่เปลี่ยนน้อยที่สุด
```

Step 6 — Test Production URL

Deploy ผ่านไม่ได้แปลว่า app ใช้ได้จริง

ต้อง test production URL

Checklist 5 นาที:

```
[ ] หน้าเปิดได้
[ ] ไม่มี blank screen
[ ] mobile อ่านได้
[ ] CTA/link สำคัญกดได้
[ ] form submit ได้
[ ] success message แสดงถูก
[ ] invalid input แสดง error ที่เข้าใจได้
[ ] ไม่มีข้อมูล test แปลก ๆ โผล่
[ ] ไม่มี secret/key โผล่ในหน้าเว็บหรือ error
```

ถ้ามี database เช่น Supabase ให้เพิ่ม:

```
[ ] row เข้า table ถูกต้อง
[ ] public user อ่านข้อมูลคนอื่นไม่ได้
[ ] test data ถูกลบหรือ mark ไว้
[ ] RLS ยังเปิดอยู่
```

Prompt หลัง test:

```
ช่วยสรุป production test result นี้
แยกเป็น Pass / Needs Fix / Blocker

Test result:
[วาง checklist พร้อม note]

ถ้ามี blocker ให้บอกว่าควร rollback, fix แล้ว redeploy, หรือปิด feature ชั่วคราว
```

Step 7 — Preview Deployment คืออะไร

หลัง deploy ครั้งแรกแล้ว อย่าแก้ production ตรง ๆ ถ้าเลี่ยงได้

Vercel จะสร้าง Preview Deployment เมื่อคุณ push ไป branch ที่ไม่ใช่ production branch หรือทำ PR

แปลแบบง่าย:

```
Preview = URL ทดสอบของ change ใหม่
Production = URL จริงของ user
```

workflow ที่ควรใช้:

```
สร้าง branch
→ แก้ feature/bug
→ push branch
→ Vercel สร้าง preview URL
→ test preview
→ merge เข้า main
→ Vercel deploy production
```

สำหรับ non-coder ให้จำประโยคนี้:

> ของใหม่ควรผ่าน preview ก่อน production

Prompt:

```
ช่วยทำ preview test plan สำหรับ change นี้

Context:
[อธิบาย change]

ให้แยกเป็น:
1. smoke test 5 นาที
2. manual test สำคัญ
3. mobile check
4. data/security check
5. สิ่งที่ไม่ต้อง test รอบนี้
```

Step 8 — ถ้า deploy fail ให้ดู log แบบไหน

Deploy fail ที่พบบ่อย:

อาการ	อาจเกิดจาก
build command fail	command ผิด, dependency ไม่ครบ, TypeScript/lint error
module not found	import ผิด, package ยังไม่ได้ติดตั้ง
env var missing	ลืมใส่ env ใน Vercel
works local but not Vercel	local/production env ต่างกัน
page loads but form fail	API/database/RLS/env/runtime issue

เวลา copy log ให้ Claude:

```
copy ตั้งแต่ command ที่ fail
copy error แรกที่ชัด
copy file path/line number ถ้ามี
copy ว่าเกิดบน Preview หรือ Production
```

อย่า copy แค่บรรทัดสุดท้ายถ้าคุณยังไม่รู้ว่าอะไรสำคัญ

Step 9 — Redeploy เมื่อไหร่

ต้อง redeploy เมื่อ:

- แก้ code แล้ว push ใหม่
- เพิ่ม/แก้ environment variable แล้วอยากให้ deployment ใหม่ใช้ค่าใหม่
- แก้ build setting
- restore จาก failure แล้วอยากลองใหม่

จำจาก Vercel docs:

```
env var ที่เปลี่ยนใน settings ไม่ได้เปลี่ยน deployment เก่าทันที
ต้องมี deployment ใหม่
```

ถ้าเพิ่งแก้ env var แล้วเว็บยังใช้ค่าเก่า ให้ถาม Claude:

```
ฉันเพิ่งแก้ env var ใน Vercel แต่ production ยังพัง
ช่วยเช็กว่าต้อง redeploy ใหม่ไหม
และควร test อะไรหลัง redeploy
```

Step 10 — Rollback เมื่อ production พัง

Rollback คือการย้อน production กลับไป deployment ก่อนหน้า

ใช้เมื่อ:

- production พังหลัง deploy ใหม่
- user จริงใช้งานไม่ได้

- bug กระทบ flow สำคัญ
- ยังไม่รู้ root cause และต้องหยุดผลกระทบก่อน

Vercel มี Instant Rollback สำหรับ production deployment

แต่ rollback ไม่ใช่เวทมนตร์

ข้อควรจำ:

- rollback กลับไป deployment เก่า
- config ที่ rollback ไปอาจเก่า
- env var ที่เพิ่งแก้ใน settings จะไม่ทำให้ deployment เก่าถูก rebuild ใหม่
- ถ้ามี database migration หรือ external API ต้องระวังผลกระทบ
- plan บางแบบ rollback ได้จำกัดกว่า plan อื่น

Prompt ก่อน rollback:

```
production มีปัญหาหลัง deploy ล่าสุด
ช่วยตัดสินใจว่าควร rollback ไหม

ข้อมูล:
- production URL:
- เวลา deploy:
- change ล่าสุด:
- user กระทบอย่างไร:
- error/log:
- มี database/env/API change ไหม:

กติกา:
- ถ้าควร rollback ให้บอกเหตุผล
- ถ้า rollback มี risk อะไรต้องเช็ก
- ถ้าไม่ rollback ให้เสนอ hotfix plan ที่เปลี่ยนน้อยที่สุด
```

หลัง rollback:

```
[ ] production กลับมาเปิดได้
[ ] form/flow หลักใช้ได้
[ ] note เวลา rollback
[ ] เก็บ deployment ที่พังไว้ตรวจ
[ ] ทำ root cause analysis ก่อน deploy ใหม่
```

Step 11 — Custom domain ควรทำเมื่อไหร่

อย่ารีบต่อ domain ตั้งแต่ app ยังไม่นิ่ง

ควรต่อ domain เมื่อ:

- production URL test ผ่านแล้ว
- app ไม่มี blocker
- security checklist ผ่าน
- owner รู้วิธี rollback/pause
- launch จริงต้องใช้ domain แบรนด์

ก่อนต่อ domain ให้เขียน note:

```
Domain:
[domain]

Vercel project:
[project]

Production branch:
[main]

Rollback plan:
[ทำอย่างไร]
```

ถ้าคุณยังไม่แน่ใจ ให้ใช้ Vercel URL ไปก่อนสำหรับ beta วงเล็ก

Common problems สำหรับ non-coder

Problem 1: Local ใช้ได้ แต่ Vercel build fail

ถาม Claude:

```
local ใช้ได้ แต่ Vercel build fail
ช่วยเปรียบเทียบความต่างที่เป็นไปได้ระหว่าง local กับ Vercel
เช่น node version, env var, build command, dependency, case-sensitive file path
อย่าเพิ่งแก้ไฟล์
```

Problem 2: Deploy ผ่าน แต่หน้า blank

ถาม Claude:

```
Vercel deploy ผ่าน แต่ production URL เป็น blank page
ช่วยบอก evidence ที่ต้องเก็บก่อน debug
เช่น browser console, network error, Vercel runtime log, build log
อธิบายแบบ non-coder
```

Problem 3: Form submit ไม่เข้า Supabase

ถาม Claude:

```
production form submit ไม่เข้า Supabase แต่ local ใช้ได้
ช่วยทำ checklist แยกสาเหตุ:
- Vercel env var
- Supabase URL/key
- RLS policy
- table/column name
- validation
- network/API error
อย่าเพิ่งแก้ code
```

Problem 4: เพิ่งแก้ env var แต่ยังพัง

ถาม Claude:

```
ฉันแก้ environment variable ใน Vercel แล้ว แต่ app ยังพังเหมือนเดิม
ช่วยเช็กว่าต้อง redeploy ไหม
และบอกวิธีตรวจว่า deployment ล่าสุดใช้ค่าใหม่แล้วหรือยัง โดยไม่ print ค่า secret
```

Problem 5: ไม่รู้ว่า URL ไหนคือ preview หรือ production

ถาม Claude:

> ช่วยอธิบายความต่างระหว่าง preview URL กับ production URL ของ Vercel
> และช่วยทำ checklist ว่าฉันควร test อะไรบนแต่ละ URL

Final Vercel deploy checklist

ใช้ checklist นี้ก่อนบอกว่า “deploy เสร็จแล้ว”

```
Git/GitHub
[ ] code ล่าสุด commit แล้ว
[ ] push ขึ้น GitHub แล้ว
[ ] ไม่มี .env หรือ secret ถูก commit

Vercel config
[ ] import repo ถูกตัว
[ ] framework preset ถูก
[ ] root directory ถูก
[ ] build command ถูก
[ ] env var ตั้งครบใน Preview/Production ตามที่ต้องใช้

Build/deploy
[ ] deployment ผ่าน
[ ] ไม่มี build error
[ ] production URL เปิดได้
[ ] preview URL ใช้ test change ใหม่ได้

App test
[ ] main flow ผ่าน
[ ] mobile ผ่าน
[ ] form validation ผ่าน
[ ] error/success state ผ่าน
[ ] ถ้ามี database: production data เข้า table ถูกต้อง

Security/data
[ ] ไม่มี secret ในหน้าเว็บ/error/log
[ ] RLS/permission ผ่าน review
[ ] test data ถูกจัดการแล้ว

Recovery
[ ] รู้ว่า deployment ล่าสุดคืออันไหน
[ ] รู้ว่าจะ rollback จากตรงไหน
[ ] มี launch note หรือ deploy note
```

Deploy note template

หลัง deploy ให้จดแบบนี้

```markdown
# Deploy Note

## Date/time
[date/time]

## Project
[project name]

## Environment
[preview / production]

## URL
[deployment URL]

## Git commit / branch
[commit hash / branch]

## What changed
- [change 1]
- [change 2]

## Env vars changed?
- [no / yes: list variable names only, no values]

## Tests performed
- [ ] page loads
- [ ] mobile
- [ ] form submit
- [ ] validation
- [ ] security check

## Result
[pass / needs fix / rolled back]

## Notes
[anything important]
```

Prompt รวมท้าย appendix

ใช้ prompt นี้เมื่อคุณพร้อม deploy

```
ช่วยเป็น deploy assistant สำหรับ Vercel รอบนี้
ฉันเป็น non-coder

Project context:
[อธิบาย project]

ก่อน deploy ให้ตรวจ:
1. Git status และไฟล์ที่ต้อง commit
2. .gitignore และ secret risk
3. package.json scripts
4. build command
5. environment variables ที่ต้องตั้งใน Vercel
6. Preview vs Production config
7. manual test checklist
8. rollback plan

กติกา:
- อย่า push/deploy เองถ้าฉันยังไม่ approve
- อย่า print ค่า secret จริง
- ถ้าเจอ blocker ให้หยุด
- ให้คำสั่งทีละบรรทัดพร้อมคำอธิบาย
- หลัง deploy ให้ช่วยสรุป deploy note
```

$ APPENDIX E

Supabase Waitlist Playbook สำหรับ Non-Coder

ใช้ appendix นี้เมื่อไหร่

ใช้เมื่อคุณมี landing page แล้วอยากเพิ่ม form เก็บ email เช่น:

```
waitlist
lead capture
early access
newsletter signup
contact interest form
```

เป้าหมายคือทำ database flow ที่เล็ก ปลอดภัย และ test ได้:

```
form → validate → insert into Supabase → success/error state → test local
→ deploy → test production
```

ไม่ใช่สอน Supabase ทั้งหมด

เราจะทำเฉพาะ waitlist version แรกที่ non-coder คุมความเสี่ยงได้

ภาพรวมแบบภาษาคน

Supabase ใน playbook นี้มี 4 ชิ้นที่ต้องรู้:

ชิ้น	แปลแบบง่าย
Project	พื้นที่หนึ่งสำหรับ database/app ของคุณ
Table	ตารางเก็บข้อมูล คล้าย sheet

ชิ้น	แปลแบบง่าย
API key	key ให้ app ต่อกับ Supabase
RLS / Policy	กฎว่าใครทำอะไรกับข้อมูลได้บ้าง

สำหรับ waitlist ที่ปลอดภัยแบบพื้นฐาน:

```
public user กรอก form ได้
public user insert row ได้
public userอ่านรายชื่อทั้งหมดไม่ได้
public userแก้/ลบข้อมูลไม่ได้
admin/owner ดูข้อมูลใน Supabase dashboard ได้
```

ก่อนเริ่ม: ลด scope ให้เล็กที่สุด

อย่าเริ่มจาก “ระบบสมาชิก + dashboard + email automation + payment” พร้อมกัน

เริ่มจาก waitlist v1:

```
email
name optional
use_case optional
created_at automatic
```

ไม่ควรเก็บตอนแรกถ้าไม่จำเป็น:

```
เบอร์โทร
ที่อยู่
วันเกิด
เลขบัตร
password
ข้อมูลสุขภาพ/การเงิน/กฎหมาย
```

Prompt:

```
ช่วย review waitlist fields ของ app นี้
แยกเป็น:
1. จำเป็นจริง ๆ
2. optional
3. ไม่ควรเก็บตอนนี้

อธิบายความเสี่ยงด้าน privacy/security แบบ non-coder
อย่าเพิ่งสร้าง table
```

Step 1 — สร้าง Supabase project

ไปที่ Supabase Dashboard แล้วสร้าง project ใหม่

สิ่งที่ต้องจดไว้ใน note ส่วนตัว:

```
Project name:
Organization:
Region:
Project URL:
Date created:
Owner:
```

อย่าจด database password หรือ secret key ลงใน manuscript, README, หรือ chat prompt ถ้าไม่จำเป็น

หลังสร้าง project แล้ว ให้เข้าใจว่า Supabase project มี Postgres database ให้คุณใช้

Supabase docs ระบุว่า Table Editor ช่วยให้ใช้งาน database ได้คล้าย spreadsheet ซึ่งเหมาะกับมือใหม่

Step 2 — เลือกวิธีสร้าง table

มี 2 วิธีหลัก:

วิธี A: Table Editor

เหมาะกับ non-coder เพราะเห็นหน้าตาเป็นตาราง

ใช้เมื่อ:

- table ง่าย
- field น้อย
- ยังไม่มั่นใจ SQL

วิธี B: SQL Editor

เหมาะเมื่ออยากให้ setup ทำซ้ำได้ และให้ Claude review ได้ชัด

ใช้เมื่อ:

- ต้องการสร้าง table + RLS + policy เป็นชุดเดียว
- ต้องการเก็บ script ไว้ใน docs
- มี developer ช่วย review ได้

สำหรับหนังสือเล่มนี้ แนะนำให้ Claude สร้าง SQL แล้วให้คุณ review ก่อนรัน

ไม่ใช่ให้ Claude กดหรือรันใน dashboard แทนคุณโดยไม่เข้าใจ

Step 3 — Table schema สำหรับ waitlist v1

Schema เริ่มต้นที่พอสำหรับ waitlist:

Field	Type แบบเข้าใจง่าย	จำเป็นไหม	ใช้ทำอะไร
`id`	auto number	yes	id ของ row
`email`	text	yes	ติดต่อกลับ
`name`	text	no	เรียกชื่อ
`use_case`	text	no	รู้ว่า user สนใจอะไร
`created_at`	timestamp	yes	รู้ว่าสมัครเมื่อไหร่

ไม่ต้องมี `password`

waitlist ไม่ใช่ระบบ account

Step 4 — SQL starter สำหรับ waitlist insert-only

ตัวอย่างนี้เป็น starter สำหรับ waitlist ที่ public user insert ได้ แต่ไม่มี policy ให้ public select/update/delete

ให้ใช้เป็นจุดเริ่มต้น ไม่ใช่กฎหมายตายตัว

• SQL

```sql
-- Waitlist table: public can submit, but cannot read all rows

create table if not exists public.waitlist_leads (
  id bigint generated always as identity primary key,
  email text not null,
  name text,
  use_case text,
  created_at timestamptz not null default now()
);

-- Optional: prevent exact duplicate emails.
-- This is simple and case-sensitive. If you need case-insensitive uniqueness,
-- ask a developer to review the best approach.
create unique index if not exists waitlist_leads_email_unique
on public.waitlist_leads (email);

-- Enable Row Level Security.
alter table public.waitlist_leads enable row level security;

-- Allow anonymous/public clients to insert only.
grant insert on public.waitlist_leads to anon;

-- RLS policy: public can submit a row that passes basic checks.
create policy "Public can submit waitlist"
on public.waitlist_leads
for insert
to anon
with check (
  email is not null
  and length(email) ⩽ 320
  and (name is null or length(name) ⩽ 120)
  and (use_case is null or length(use_case) ⩽ 1000)
);
```

สิ่งที่ intentionally ไม่มี:

```
ไม่มี select policy สำหรับ anon
ไม่มี update policy สำหรับ anon
ไม่มี delete policy สำหรับ anon
```

แปลว่า public user ไม่ควรอ่าน/แก้/ลบข้อมูลคนอื่นผ่าน API ได้

ก่อนรัน SQL ให้ใช้ prompt นี้:

```
ช่วย review SQL waitlist นี้ก่อนรันใน Supabase
ฉันเป็น non-coder

ตรวจว่า:
1. table เก็บข้อมูลเท่าที่จำเป็นไหม
2. RLS เปิดไหม
3. anon ทำได้เฉพาะ insert ใช่ไหม
4. anon select/update/delete ได้ไหม
5. มี secret หรือ admin key เกี่ยวข้องไหม
6. มี risk อะไรที่ต้องรู้ก่อนใช้จริง

อย่าเสนอปิด RLS
```

Step 5 — หา Supabase URL และ publishable key

Supabase docs ระบุว่าส่วนใหญ่เอา key ได้จาก Connect dialog หรือ Settings > API Keys

สำหรับ waitlist frontend ทั่วไป คุณต้องใช้:

```
Project URL
Publishable key
```

ใน legacy project อาจเห็นชื่อ `anon key`

ให้จำแบบนี้:

Key	ใช้ตรงไหน	ปลอดภัยใน browser ไหม
Publishable key	frontend/public app	ใช้ได้ ถ้ามี RLS/policy
anon key legacy	frontend/public app	ใช้ได้ ถ้ามี RLS/policy
Secret key	backend/server เท่านั้น	ห้าม browser
service_role legacy	backend/server/admin เท่านั้น	ห้าม browser

กฎสำคัญ:

> `publishable key ไม่ใช่ password สำหรับป้องกัน data`
> `RLS/policy ต่างหากที่ป้องกัน data`

ถ้า Claude เสนอให้ใช้ `service_role` เพื่อให้ insert ผ่าน ให้หยุดทันที

Step 6 — ใส่ค่าใน `.env.local`

สำหรับ local development ให้สร้าง `.env.local`

ตัวอย่างสำหรับ Next.js:

```bash
NEXT_PUBLIC_SUPABASE_URL=your-project-url
NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY=your-publishable-key
```

ห้าม commit `.env.local`

ควรมี `.env.example` แบบ placeholder:

```bash
NEXT_PUBLIC_SUPABASE_URL=your-supabase-project-url
NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY=your-supabase-publishable-key
```

Prompt:

```
ช่วยตรวจ env setup สำหรับ Supabase waitlist

กติกา:
- อย่า print ค่า key จริง
- ตรวจว่าควรมี variable ชื่ออะไร
- ตรวจว่า .env.local ไม่ควรถูก commit
- ตรวจว่า .env.example มีแต่ placeholder
- บอกว่า variable ไหน public และ variable ไหนห้ามอยู่ frontend
```

Step 7 — ให้ Claude ต่อ form กับ Supabase

ก่อนให้ Claude แก้ code ให้สั่งวาง plan

```
ช่วยวาง plan เพื่อต่อ waitlist form เข้ากับ Supabase

Context:
- table: waitlist_leads
- fields: email, name, use_case
- env vars: NEXT_PUBLIC_SUPABASE_URL,
NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY

กติกา:
- อย่าเพิ่งแก้ไฟล์
- ใช้ publishable key เท่านั้นใน frontend
- ห้ามใช้ secret/service_role ใน frontend
- ต้องมี validation
- ต้องมี success/error state
- ต้องไม่ log key หรือข้อมูลส่วนตัวเกินจำเป็น
- ระบุไฟล์ที่จะเปลี่ยน
- ระบุ manual test หลังแก้
```

หลัง plan ผ่าน ค่อยให้แก้:

```
โอเค ทำตาม plan ได้
เปลี่ยนเฉพาะ waitlist form และ Supabase client ที่จำเป็น
หลังแก้ให้สรุป diff และ manual test checklist
```

Step 8 — Validation ที่ควรมี

Validation ขั้นต่ำ:

```
email ต้องไม่ว่าง
email ต้องหน้าตาคล้าย email
email ยาวไม่เกิน 320 ตัวอักษร
name optional แต่ถ้ามีต้องไม่ยาวเกิน
use_case optional แต่ถ้ามีต้องไม่ยาวเกิน
```

อย่า rely เฉพาะ placeholder ใน form

ต้องมี logic ตรวจจริง

Prompt:

```
ช่วย review waitlist form validation
ตรวจทั้ง frontend และ database/RLS policy
บอกว่า:
- ตอนนี้ validate อะไรแล้ว
- ยังขาดอะไร
- error message user อ่านเข้าใจไหม
- มี input ที่ทำให้ app พังได้ไหม
อย่าเพิ่งแก้ code
```

Step 9 — Test local

เมื่อ Claude ต่อ form แล้ว ให้ test ในเครื่องก่อน

Checklist:

```
[ ] เปิด local URL ได้
[ ] submit email ที่ถูกต้องแล้ว success
[ ] row เข้า Supabase table
[ ] email ว่างแล้ว error
[ ] email ไม่ถูก format แล้ว error
[ ] submit ซ้ำแล้ว behavior ชัดเจน
[ ] name/use_case ยาวมากไม่ทำให้ app พัง
[ ] error message ไม่โชว์ technical secret
[ ] console ไม่ log key หรือข้อมูลเกินจำเป็น
```

ถ้ามี duplicate email แล้ว error ให้ตัดสินใจว่าจะให้ user เห็นข้อความแบบไหน

ตัวอย่างข้อความที่ดี:

```
อีเมลนี้อยู่ใน waitlist แล้ว
```

ไม่ใช่:

```
duplicate key value violates unique constraint
waitlist_leads_email_unique
```

Step 10 — ตรวจว่า public อ่าน data ไม่ได้

จุดสำคัญที่สุดของ waitlist คือ public submit ได้ แต่ไม่ควรอ่าน email ทุกคนได้

ให้ Claude ช่วยตรวจด้วย prompt:

```
ช่วยตรวจว่า public/anon user อ่านข้อมูล waitlist ทั้งหมดไม่ได้

กติกา:
- อย่าใช้ service_role ในการ test ฝั่ง public
- อธิบายว่าจะ test อย่างไร
- ตรวจ RLS policies
- ตรวจว่าไม่มี frontend page/API route ที่ expose list ทั้งหมด
- ถ้าต้องรัน command ให้บอกก่อนว่าทำอะไร
```

ถ้า Claude บอกว่า “สร้าง select policy ให้ anon เพื่อ debug” ให้หยุด

สำหรับ waitlist public ไม่จำเป็นต้องให้ anon select รายชื่อทั้งหมด

Step 11 — ตรวจ secret ก่อน commit

ก่อน commit ให้ตรวจ:

```
[ ] ไม่มี .env.local ถูก track
[ ] ไม่มี key จริงใน source code
[ ] ไม่มี service_role ใน frontend
[ ] ไม่มี screenshot ที่มี key จริง
[ ] .env.example มี placeholder เท่านั้น
[ ] README ไม่ใส่ key จริง
```

Prompt:

```
ช่วย review changes ก่อน commit สำหรับ Supabase waitlist
ตรวจว่า:
1. เปลี่ยนไฟล์อะไร
2. มี secret/key จริงหลุดไหม
3. มี service_role หรือ secret key ใน frontend ไหม
4. RLS/policy ยังปลอดภัยไหม
5. manual test ที่ต้องทำก่อน deploy คืออะไร
6. commit message ควรเป็นอะไร

อย่า print ค่า secret จริง
```

Step 12 — ใส่ env vars ใน Vercel

ถ้า deploy ไป Vercel ต้องใส่ env vars ใน Vercel ด้วย

ไม่ใช่แค่ใน `.env.local`

สำหรับ waitlist frontend-only:

Variable	Vercel Environment	Public/Secret	หมายเหตุ
`NEXT_PUBLIC_SUPABASE_URL`	Preview + Production	Public	Supabase project URL
`NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY`	Preview + Production	Public	ต้องมี RLS/policy

ถ้ามี server-side function จริง ๆ อาจมี secret key แต่สำหรับ waitlist v1 ไม่ควรต้องใช้

ถ้า Claude บอกว่าต้องใช้ secret key ให้ถาม:

```
ทำไม waitlist v1 ต้องใช้ secret key
มีวิธีใช้ publishable key + RLS policy แทนไหม
ถ้าต้องใช้ secret จริง ๆ มันอยู่ server-side เท่านั้นใช่ไหม
```

หลังเพิ่ม env var ใน Vercel ต้อง redeploy เพื่อให้ deployment ใหม่ใช้ค่าใหม่

Step 13 — Test production

หลัง deploy ผ่าน ให้ test production URL จริง

Checklist:

```
[ ] production page เปิดได้
[ ] form submit ได้
[ ] row เข้า Supabase table
[ ] invalid email แสดง error
[ ] duplicate email แสดงข้อความที่เข้าใจได้
[ ] mobile ใช้ได้
[ ] ไม่มี technical error โผล่ให้ user เห็น
[ ] ไม่มี key/secret โผล่ในหน้าเว็บ/log/error
[ ] ลบหรือ mark test row หลัง test
```

ถ้า production ไม่เข้า Supabase แต่ local เข้า ให้ดู 5 จุดนี้:

```
1. Vercel env var ตั้งครบไหม
2. env var อยู่ใน Production environment ไหม
3. redeploy หลังตั้ง env var แล้วไหม
4. Supabase URL/key ถูก project เดียวกันไหม
5. RLS policy allow insert ให้ anon/publishable path ไหม
```

Prompt:

```
production waitlist submit ไม่เข้า Supabase แต่ local ใช้ได้
ช่วย debug แบบ non-coder

ข้อมูล:
- production URL:
- env var names ที่ตั้งใน Vercel:
- table name:
- RLS policies:
- error message/log:

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อย่า print ค่า key จริง
- แยกสาเหตุ env / RLS / table schema / validation / network
- เสนอวิธีตรวจทีละข้อ
```

Step 14 — ดูข้อมูล lead อย่างไร

สำหรับ version แรก เจ้าของ project ดู lead ผ่าน Supabase Dashboard ได้

อย่าเพิ่งสร้าง public dashboard ถ้ายังไม่มี auth/permission ชัด

ถ้าต้อง export lead:

```
export เฉพาะข้อมูลที่จำเป็น
เก็บไฟล์ export ให้ปลอดภัย
ลบไฟล์ export ที่ไม่ใช้
อย่าส่ง CSV ที่มี email ผ่านช่องทางไม่ปลอดภัย
```

ถ้าจะสร้าง admin dashboard ให้กลับไปใช้บท Internal Dashboard และ Security Checklist ก่อน

Common problems สำหรับ waitlist

Problem 1: Submit แล้วขึ้น error เรื่อง RLS

สาเหตุที่เป็นไปได้:

- RLS เปิด แต่ยังไม่มี insert policy
- policy ใช้ role ผิด
- grant insert ให้ anon ยังไม่ถูก
- field ไม่ผ่าน `with check`

Prompt:

```
Supabase insert ถูก RLS block
ช่วยอ่าน error นี้และ policy ปัจจุบัน
อธิบายว่า block เพราะอะไร
อย่าเสนอปิด RLS
เสนอ fix ที่ทำให้ public insert ได้เฉพาะ waitlist row เท่านั้น
```

Problem 2: Duplicate email error อ่านไม่รู้เรื่อง

ควรแปลงเป็นข้อความ user-friendly

```
อีเมลนี้อยู่ใน waitlist แล้ว
```

Prompt:

```
ช่วยปรับ duplicate email error ให้ user อ่านเข้าใจ
อย่าเปลี่ยน database schema ถ้าไม่จำเป็น
อย่าเปิดเผย constraint name หรือ SQL error ให้ user เห็น
```

Problem 3: Production ใช้ key ผิด project

อาการ:

- local เข้า project A
- production เข้า project B

- ดู table แล้วไม่เห็น row

Checklist:

```
[ ] URL/key ใน .env.local และ Vercel เป็น project เดียวกันไหม
[ ] table มีอยู่ใน project production ไหม
[ ] Vercel redeploy แล้วไหม
```

Problem 4: Claude เสนอ service_role ใน frontend

ให้หยุดทันที

```
หยุดก่อน
service_role/secret key ห้ามอยู่ frontend ใช่ไหม
ช่วยเสนอ architecture ที่ไม่ expose secret
สำหรับ waitlist v1 ใช้ publishable key + RLS insert policy ได้ไหม
```

Problem 5: User submit ได้ แต่คุณไม่เห็น row

สาเหตุที่เป็นไปได้:

- ดูผิด Supabase project
- ดูผิด table
- insert fail แต่ UI แสดง success ผิด
- production ใช้ env var ผิด
- duplicate error ถูกกลืนไว้

Prompt:

```
UI แสดง success แต่ฉันไม่เห็น row ใน Supabase
ช่วยทำ debug checklist
ตรวจทั้ง frontend response, Supabase project/table, env var, error handling
อย่าเพิ่งแก้ code
```

Waitlist launch checklist

ก่อนเปิดให้คนอื่นกรอกจริง:

```
Data
[ ] เก็บเฉพาะ email/name/use_case หรือ field ที่จำเป็น
[ ] ไม่มี sensitive data ที่ไม่จำเป็น

Supabase
[ ] table ถูกต้อง
[ ] RLS enabled
[ ] anon insert ได้
[ ] anon select/update/delete ไม่ได้
[ ] Security Advisor ไม่มี issue สำคัญที่เกี่ยวกับ table นี้

Keys/env
[ ] ใช้ publishable key ใน frontend
[ ] ไม่มี secret/service_role ใน frontend
[ ] .env.local ไม่ถูก commit
[ ] Vercel env vars ตั้งครบใน Production
[ ] redeploy หลังแก้ env var แล้ว

App
[ ] validation ทำงาน
[ ] success/error state ชัด
[ ] duplicate email handled
[ ] mobile ใช้ได้
[ ] production submit แล้ว row เข้า

Ops
[ ] รู้ว่าจะดู lead ที่ไหน
[ ] test data ถูกลบ/mark แล้ว
[ ] มี launch note
[ ] มีวิธีปิด form ชั่วคราวถ้า spam หรือ error
```

Launch note template สำหรับ waitlist

MARKDOWN

```markdown
# Waitlist Launch Note

## Date/time
[date/time]

## URL
[production URL]

## Supabase project
[project name only, no keys]

## Table
waitlist_leads

## Data collected
- email
- name optional
- use_case optional

## RLS summary
- public/anon can insert
- public/anon cannot select/update/delete

## Env vars used
- NEXT_PUBLIC_SUPABASE_URL
- NEXT_PUBLIC_SUPABASE_PUBLISHABLE_KEY

## Production test
- [ ] submit valid email
- [ ] invalid email rejected
- [ ] duplicate handled
- [ ] row visible in dashboard
- [ ] mobile checked

## Known limitations
- [TODO]

## Owner
[name]
```

Prompt รวมท้าย appendix

ใช้ prompt นี้ก่อนเปิด waitlist ให้คนอื่นใช้

```
ช่วยเป็น Supabase waitlist launch reviewer
ฉันเป็น non-coder

ตรวจ project นี้ตั้งแต่ form ถึง database:
1. data fields
2. Supabase table schema
3. RLS enabled หรือไม่
4. policies: anon insert / no anon select-update-delete
5. publishable vs secret/service_role keys
6. .env.local / .env.example / Vercel env vars
7. validation
8. success/error states
9. local test
10. production test
11. launch note

กติกา:
- อย่า print ค่า key จริง
- อย่าเสนอปิด RLS
- ถ้าเจอ secret/service_role ใน frontend ให้บอกเป็น Critical
- แยกผลเป็น Green / Yellow / Red
- เสนอ next steps ไม่เกิน 5 ข้อ
```

$ APPENDIX F

Debug Evidence Playbook สำหรับ Non-Coder

ใช้ appendix นี้เมื่อไหร่

ใช้เมื่อ app พัง แต่คุณยังไม่รู้ว่าเกิดจากอะไร

เป้าหมายไม่ใช่ให้คุณอ่าน code เป็น

เป้าหมายคือให้คุณเก็บหลักฐานถูกชุดก่อนให้ Claude Code ช่วยแก้

จำประโยคนี้:

```
Error ที่ไม่มีหลักฐาน = AI ต้องเดา
Error ที่มีหลักฐาน = AI debug ได้เป็นระบบ
```

Debug workflow แบบสั้นที่สุด

ทุกครั้งที่เจอปัญหา ใช้ flow นี้:

```
หยุด
→ จดว่า expected คืออะไร
→ จดว่า actual คืออะไร
→ reproduce ให้ได้
→ เก็บ screenshot/log/error
→ บอก environment
→ ให้ Claude วิเคราะห์ก่อนแก้
→ แก้ทีละจุด
→ test ซ้ำ
```

ห้ามเริ่มด้วย:

> มันพัง แก้หน่อย

ให้เริ่มด้วย:

> ช่วยวิเคราะห์จากหลักฐานนี้ก่อน อย่าเพิ่งแก้ไฟล์

Error 6 แบบที่ควรแยกให้ได้

ประเภท	เกิดตอนไหน	หลักฐานที่ต้องเก็บ
Install error	ตอน `npm install`	terminal output
Build error	ตอน `npm run build` หรือ Vercel build	build log
Runtime error	เปิดเว็บ/กดปุ่มแล้วพัง	browser console + screenshot
API/data error	form submit ไม่เข้า, 401/403/500	network tab + server log
Permission/RLS error	Supabase block	error message + policy summary
Deploy error	Vercel deploy fail	Vercel build/deployment log

แค่บอกถูกว่าอยู่กลุ่มไหน ก็ช่วย Claude ได้เยอะแล้ว

Evidence template หลัก

ใช้ template นี้ทุกครั้ง

```
ฉันต้องการ debug ปัญหานี้แบบเป็นระบบ
อย่าเพิ่งแก้ไฟล์

## Environment
- local / preview / production:
- URL:
- browser/device:
- time happened:

## What I expected
[ควรเกิดอะไร]

## What actually happened
[เกิดอะไรขึ้นจริง]

## Steps to reproduce
1. ...
2. ...
3. ...

## Evidence
- screenshot:
- browser console error:
- network error/status:
- terminal log:
- Vercel log:
- Supabase error/policy info:

## Recent changes
[เพิ่งแก้อะไร / commit / deploy อะไร]

กติกา:
- อธิบายแบบ non-coder
- ชี้หลักฐานที่สำคัญที่สุด
- เสนอ root causes 2-3 แบบ
- บอกว่าจะตรวจอะไรเพื่อยืนยัน
- อย่าแก้ code จนกว่าจะมี plan
```

เก็บ Browser Console Error

ใช้เมื่อ:

- หน้า blank
- ปุ่มกดแล้วไม่เกิดอะไร
- form submit แล้วพัง
- UI ค้าง
- browser แสดง error

วิธีคิดแบบง่าย:

```
Browser console = เสียงบ่นของหน้าเว็บฝั่ง user
```

สิ่งที่ต้อง copy:

```
error message สีแดง
file name ถ้ามี
line number ถ้ามี
stack trace ถ้ามี
warning ที่เกี่ยวกับ env/API/network
```

อย่า copy แค่คำว่า “error”

Prompt:

```
นี่คือ browser console error
ช่วยอธิบายแบบ non-coder ว่า error นี้น่าจะเกี่ยวกับอะไร

Console error:
[paste error]

Context:
- หน้า/URL:
- กดอะไรแล้วเกิด:
- local/production:

กติกา:
- อย่าเพิ่งแก้ไฟล์
- แยกว่าเป็น UI bug, missing env, API error, หรือ code runtime error
- บอก evidence ที่ควรเก็บเพิ่ม
```

เก็บ Network Error

ใช้เมื่อ form/API ไม่ทำงาน

ตัวอย่าง:

```
submit แล้วไม่เข้า database
login fail
dashboard โหลดข้อมูลไม่ได้
```

สิ่งที่ต้องดูใน Network tab:

```
request URL
status code เช่น 400 / 401 / 403 / 404 / 500
response body/message
method เช่น GET / POST
เวลาเกิด error
```

ห้าม paste token หรือ secret ที่อยู่ใน headers

ถ้าไม่แน่ใจ ให้บอก Claude:

```
ฉันจะ paste เฉพาะ status code, URL path, และ response body ที่ไม่มี token
```

Prompt:

```
นี่คือ network error จาก browser
ช่วยวิเคราะห์ก่อนแก้

Request:
- method:
- URL/path:
- status code:
- response body:

Context:
- user action:
- local/preview/production:

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อธิบาย status code แบบ non-coder
- แยกว่าเป็น validation, auth, permission, env, หรือ server error
- บอกหลักฐานที่ต้องเก็บเพิ่ม
```

Status code แบบคนทั่วไป

Code	แปลแบบง่าย	มักเช็กอะไร
400	request ไม่ถูกต้อง	field, validation, payload
401	ยังไม่ authenticated	login/session/key
403	ไม่มี permission	role, RLS, authorization
404	หาไม่เจอ	URL, route, API path, table/resource
409	conflict	duplicate email, record ซ้ำ
429	rate limit	ส่งถี่เกิน quota/limit

Code	แปลแบบง่าย	มักเช็กอะไร
500	server พัง	server log, env, code error
502/503/504	upstream/server unavailable	platform/API outage, timeout

อย่าใช้ตารางนี้ฟันธง

ใช้เพื่อรู้ว่าควรเก็บหลักฐานอะไรต่อ

เก็บ Terminal Error

ใช้เมื่อ command ในเครื่องพัง เช่น:

```bash
npm install
npm run dev
npm run build
npm test
```

สิ่งที่ต้อง copy:

> command ที่รัน
> error ตั้งแต่เริ่ม fail
> บรรทัดที่บอก file/line
> บรรทัดแรกที่เป็น root error

ถ้า output ยาวมาก ให้ copy ส่วนท้าย 100–200 บรรทัดก่อน แล้วถาม Claude ว่าต้องการส่วนไหนเพิ่ม

Prompt:

```
command นี้ fail ใน terminal
ช่วยวิเคราะห์ก่อนแก้

Command:
[command]

Output:
[paste output]

กติกา:
- อย่าเพิ่งแก้ไฟล์
- ชี้บรรทัดที่เป็น root error
- แยก install/build/runtime issue
- เสนอ fix ที่เปลี่ยนน้อยที่สุด
```

เก็บ Vercel Build Log

ใช้เมื่อ deploy fail หรือ build fail บน Vercel

สิ่งที่ต้องเก็บ:

```
deployment environment: preview หรือ production
branch/commit
build command
error log ส่วนที่ fail
file path/line ถ้ามี
```

Deploy fail มักเกิดจาก:

- build command fail
- dependency ไม่ครบ
- env var missing
- TypeScript/lint error
- file path case sensitivity
- local มีไฟล์ที่ไม่ได้ commit

Prompt:

```
Vercel build fail
ช่วยวิเคราะห์จาก build log นี้

Environment:
- preview / production:
- branch:
- commit:

Build log:
[paste log]

กติกา:
- อย่าเพิ่งแก้ไฟล์
- ชี้บรรทัด root error
- แยกสาเหตุ env / dependency / build command / code error
- บอกวิธีตรวจแต่ละสาเหตุ
- เสนอ fix ที่เปลี่ยนน้อยที่สุด
```

เก็บ Vercel Runtime Evidence

บางครั้ง deploy ผ่าน แต่เว็บพังตอน user ใช้

เรียกว่า runtime error

ตัวอย่าง:

```
หน้าเปิดได้ แต่ submit form แล้ว 500
API route fail
server action fail
```

หลักฐานที่ควรเก็บ:

```
production URL
user action
browser network status
response body
Vercel runtime/function log ถ้ามี
เวลาเกิดปัญหา
```

Prompt:

```
Vercel deploy ผ่าน แต่ runtime error ตอนใช้งาน
ช่วย debug จาก evidence นี้

URL:
User action:
Network status/response:
Runtime log:
Time:
Recent deploy/change:

กติกา:
- อย่าเพิ่งแก้ไฟล์
- ประเมินว่าควร rollback ไหมถ้ากระทบ user จริง
- แยก env var, server code, database permission, external API
- เสนอขั้นตอนตรวจทีละข้อ
```

เก็บ Supabase Evidence

ใช้เมื่อเกี่ยวกับ database, RLS, key, auth

ตัวอย่าง:

```
insert ไม่เข้า
select ไม่ได้
permission denied
RLS block
401/403 จาก Supabase
```

สิ่งที่ต้องเก็บโดยไม่เปิดเผย secret:

```
table name
operation: select/insert/update/delete
role ที่คาดว่าใช้: anon/authenticated/admin
RLS enabled ไหม
policy names หรือ policy summary
error message
ใช้ local/production
```

ห้าม paste:

```
secret key
service_role key
JWT token
database password
```

Prompt:

```
Supabase operation fail
ช่วยวิเคราะห์แบบ non-coder

Operation:
- table:
- action: select / insert / update / delete
- expected role: anon / authenticated / admin
- local/production:

Evidence:
- error message:
- RLS enabled:
- policy summary:
- network status if any:

กติกา:
- อย่าใช้ service_role เพื่อ bypass โดยไม่จำเป็น
- อย่าเสนอปิด RLS
- แยกสาเหตุ key / RLS policy / table schema / validation / env var
- เสนอวิธีตรวจแบบปลอดภัย
```

Local ใช้ได้ แต่ Production พัง

นี่เป็นเคสยอดฮิต

ให้เทียบ 6 จุดนี้:

```
1. env var local กับ Vercel เหมือนกันไหม
2. Vercel redeploy หลังแก้ env var แล้วไหม
3. local มีไฟล์ที่ยังไม่ได้ commit ไหม
4. production ใช้ Supabase project/table เดียวกันไหม
5. build command บน Vercel เหมือนที่รัน local ไหม
6. RLS/policy ต่างกันไหมระหว่าง project
```

Prompt:

```
local ใช้ได้ แต่ production พัง
ช่วยทำ differential diagnosis

Local result:
[ใช้ได้อย่างไร]

Production result:
[พังอย่างไร]

Evidence:
- browser console:
- network status:
- Vercel log:
- Supabase info:
- recent changes:

กติกา:
- อย่าเพิ่งแก้ไฟล์
- ทำตาราง Local vs Production
- บอกหลักฐานที่ขาด
- เสนอ test ที่เร็วที่สุดเพื่อแยกสาเหตุ
```

Production พัง: rollback หรือ debug ก่อน?

ถ้ามี user จริง ให้คิดแบบ incident ก่อน

Rollback ก่อน ถ้า:

```
user ทำ flow หลักไม่ได้
มี payment/data/security impact
error กระทบ user หลายคน
ยังไม่รู้ root cause
rollback มี risk ต่ำกว่า hotfix
```

Debug ก่อน อาจพอได้ ถ้า:

```
ยังไม่มี user จริง
เป็น preview/beta วงเล็ก
bug ไม่กระทบ flow หลัก
fix ชัดมากและ test ได้เร็ว
```

Prompt:

```
production พังหลัง deploy ล่าสุด
ช่วยตัดสินใจว่า rollback หรือ debug ก่อน

Impact:
[ใครได้รับผลกระทบ]

Evidence:
[log/error]

Recent change:
[deploy/commit]

กติกา:
- คิดแบบลดผลกระทบ user ก่อน
- ถ้าแนะนำ rollback ให้บอกสิ่งที่ต้องเช็กหลัง rollback
- ถ้าแนะนำ hotfix ให้บอก test ขั้นต่ำก่อน deploy ใหม่
```

ให้ Claude แก้หลังมี evidence แล้ว

เมื่อวิเคราะห์พอแล้ว ค่อยใช้ prompt แก้

```
จาก evidence และ analysis ก่อนหน้า
ช่วยแก้ root cause โดยเปลี่ยนน้อยที่สุด

กติกา:
- แก้เฉพาะ issue นี้
- อย่า refactor ส่วนอื่น
- อย่าเพิ่ม package ถ้าไม่จำเป็น
- ถ้าต้องเพิ่ม env var ให้บอกชื่อเท่านั้น ไม่ใส่ค่า
- หลังแก้ให้สรุป diff
- ระบุ manual test ที่ต้องทำ
- ระบุ risk ที่ยังเหลือ
```

หลังแก้ อย่าลืม test ซ้ำด้วย evidence เดิม

ถ้า error เดิมหาย แต่ error ใหม่มา ให้เริ่ม evidence template ใหม่

Bug report template สำหรับส่งให้ developer

ถ้าต้องส่งต่อให้ dev ใช้ template นี้

MARKDOWN

```
# Bug Report

## Summary
[สรุปปัญหา 1-2 บรรทัด]

## Environment
- local / preview / production:
- URL:
- browser/device:
- time:
- commit/deployment:

## Expected
[ควรเกิดอะไร]

## Actual
[เกิดอะไรขึ้น]

## Steps to reproduce
1. ...
2. ...
3. ...

## Evidence
- Screenshot:
- Browser console:
- Network status/response:
- Terminal log:
- Vercel log:
- Supabase error/policy summary:

## Recent changes
[commit/PR/deploy/feature]

## Impact
[low / medium / high]

## Tried already
[ลองอะไรแล้ว]

## Notes
[ข้อสังเกต]
```

Checklist ก่อนบอกว่า fixed

```
[ ] reproduce bug ได้ก่อนแก้
[ ] เข้าใจ root cause หรือมี evidence ชัด
[ ] fix เปลี่ยนน้อยที่สุด
[ ] test case เดิมผ่านแล้ว
[ ] ไม่มี secret/log แปลก ๆ เพิ่ม
[ ] build ผ่าน
[ ] preview/production test ตาม environment จริง
[ ] ถ้าเป็น production incident มี note ว่าเกิดอะไรและแก้อย่างไร
```

ถ้าคุณ reproduce ไม่ได้เลย ให้ระวังการบอกว่า fixed

อาจเป็นแค่ bug ที่ยังไม่เกิดซ้ำในรอบนั้น

Prompt รวมท้าย appendix

ใช้ prompt นี้เป็น default เวลาพัง

```
ช่วยเป็น debug evidence assistant
ฉันเป็น non-coder

เป้าหมาย:
เก็บหลักฐานให้ครบก่อนแก้ code

ปัญหา:
[อธิบายสั้น ๆ]

ช่วยถามฉันทีละขั้นว่าเราต้องเก็บ evidence อะไรบ้างจาก:
1. browser console
2. network tab
3. terminal
4. Vercel build/runtime logs
5. Supabase table/RLS/error
6. recent commits/deployments

กติกา:
- อย่าเพิ่งแก้ไฟล์
- อย่าขอให้ฉัน paste secret/token/key
- ถ้าหลักฐานพอแล้ว ให้สรุป root causes ที่เป็นไปได้
- เสนอวิธีตรวจที่เร็วที่สุด
- ค่อยเสนอ fix หลังจากมี evidence พอ
```

$ APPENDIX G

GitHub PR + Preview Release Workflow สำหรับ Non-Coder

ใช้ appendix นี้เมื่อไหร่

ใช้เมื่อ project เริ่มมีคนใช้ หรือเริ่ม deploy ไป Vercel แล้ว

เป้าหมายคือไม่แก้ทุกอย่างบน `main` ตรง ๆ

แต่ใช้ workflow ที่ปลอดภัยกว่า:

```
branch → change → commit → push → PR → preview → review → merge →
production
```

สำหรับ non-coder ให้คิดแบบนี้:

```
branch = ห้องทดลอง
PR = ใบส่งงานให้ตรวจ
preview = URL ทดสอบ
main = ของจริงที่พร้อม deploy
```

ทำไมไม่แก้บน main ตรง ๆ

ถ้า Vercel ผูกกับ GitHub แล้ว `main` มักเป็น production branch

แปลว่า:

```
push เข้า main → Vercel อาจ deploy production
```

ถ้าคุณแก้ผิดแล้ว push เข้า main โดยไม่ test preview ก่อน user จริงอาจเจอ bug ทันที

ดังนั้นสำหรับงานที่ไม่ใช่แก้ typo เล็กมาก ให้ใช้ branch + PR

คำศัพท์ที่ต้องรู้

คำ	แปลแบบง่าย
Repository	กล่องเก็บ code บน GitHub
main	branch หลัก มักใช้ deploy production
branch	พื้นที่แยกสำหรับแก้งานหนึ่งเรื่อง
commit	save point ของ code
push	ส่ง commit จากเครื่องขึ้น GitHub
pull request / PR	หน้าตรวจ change ก่อน merge
merge	เอา change เข้า main
preview deployment	URL ทดสอบที่ Vercel สร้างจาก branch/PR
production deployment	URL จริงที่ user ใช้

Workflow ภาพรวม

```
1. เลือกงานเล็ก 1 เรื่อง
2. สร้าง branch ใหม่
3. ให้ Claude แก้เฉพาะ scope นั้น
4. review diff
5. commit
6. push branch
7. เปิด PR บน GitHub
8. ดู Vercel preview URL
9. test preview
10. ถ้าผ่าน merge เข้า main
11. test production หลัง deploy
12. จด release note
```

อย่ารวม 5 feature ใน PR เดียวถ้าไม่จำเป็น

PR ที่ดีควรตอบได้ว่า:

> รอบนี้แก้อะไร และทำไม

Step 1 — เลือก scope ให้เล็ก

ตัวอย่าง scope ที่ดี:

```
fix duplicate email message
improve mobile hero spacing
add waitlist success state
hide dashboard link for public users
```

ตัวอย่าง scope ที่ใหญ่เกิน:

```
ปรับเว็บให้ดีขึ้นทั้งหมด
ทำ dashboard ใหม่ทั้งระบบ
เพิ่ม login payment และ admin panel
```

Prompt:

```
ช่วยแยกงานนี้ให้เป็น PR scope ที่เล็กพอ

งานที่อยากทำ:
[อธิบาย]

กติกา:
- แยกเป็น PR ย่อย ๆ ถ้าจำเป็น
- แต่ละ PR ต้องมี goal เดียว
- บอก risk ของแต่ละ PR
- แนะนำลำดับทำก่อนหลัง
```

Step 2 — สร้าง branch

ชื่อ branch ควรอ่านแล้วรู้ว่าทำอะไร

ตัวอย่าง:

```
fix-waitlist-duplicate-message
improve-mobile-landing-page
add-preview-test-checklist
secure-dashboard-access
```

ถ้าใช้ command line:

```bash
git checkout main
git pull
git checkout -b fix-waitlist-duplicate-message
```

ถ้าคุณไม่มั่นใจ ให้ Claude ช่วยแปล command ก่อนรัน:

```
ช่วยอธิบายคำสั่ง Git เหล่านี้ทีละบรรทัดแบบ non-coder
และบอกว่ามีความเสี่ยงอะไรไหม

Commands:
[paste commands]
```

Step 3 — ให้ Claude แก้เฉพาะ branch นี้

ก่อนแก้ ให้บอก scope ชัด

```
เรากำลังทำ branch: [branch name]

Task:
[สิ่งที่ต้องแก้]

กติกา:
- แก้เฉพาะ scope นี้
- อย่าเพิ่ม feature ใหม่
- อย่า refactor ส่วนอื่น
- ถ้าต้องเพิ่ม package/env var/database change ให้หยุดถามก่อน
- หลังแก้ให้สรุป diff และ manual test
```

ถ้า Claude เริ่มแตะไฟล์เยอะผิดปกติ ให้หยุดและถาม:

```
ทำไม task นี้ต้องแก้ไฟล์เยอะขนาดนี้
มีวิธีแก้ที่เปลี่ยนน้อยกว่านี้ไหม
```

Step 4 — Review diff ก่อน commit

ก่อน commit ต้องดูว่า AI เปลี่ยนอะไร

ให้ Claude ช่วยสรุป:

```
ช่วย review diff ปัจจุบันก่อน commit

ตรวจว่า:
1. เปลี่ยนไฟล์อะไรบ้าง
2. แต่ละไฟล์เปลี่ยนเพื่ออะไร
3. มี change นอก scope ไหม
4. มี secret/.env/key หลุดไหม
5. มี package ใหม่ไหม
6. มีผลต่อ database/auth/RLS/deploy ไหม
7. ต้อง test อะไร
8. commit message ควรเป็นอะไร

ถ้ามี red flag ให้บอกให้หยุดก่อน commit
```

Checklist เอง:

```
[ ] ไม่มี .env
[ ] ไม่มี secret key
[ ] ไม่มี service_role ใน frontend
[ ] ไม่มี package แปลก ๆ
[ ] ไม่แตะไฟล์นอก scope
[ ] test checklist ชัดเจน
```

Step 5 — Commit

Commit คือ save point

ควร commit เมื่อ:

```
งานย่อยจบ
diff เข้าใจได้
test ขั้นต่ำผ่าน
ไม่มี red flag
```

ตัวอย่าง commit message:

```
fix: handle duplicate waitlist emails
feat: add waitlist success state
docs: add deploy checklist
chore: remove unused dependency
```

Prompt:

```
ช่วยตั้ง commit message สำหรับ changes นี้
ใช้ Conventional Commits
ให้เลือก 2-3 ตัวเลือก พร้อมเหตุผลสั้น ๆ
```

ถ้าใช้ command line:

```bash
git status
git add [files]
git commit -m "fix: handle duplicate waitlist emails"
```

อย่า `git add .` ถ้าคุณยังไม่ได้ดูว่ามีไฟล์อะไรบ้าง

Step 6 — Push branch ขึ้น GitHub

หลัง commit แล้ว push branch:

```bash
git push -u origin fix-waitlist-duplicate-message
```

ถ้า error ให้ copy error มาให้ Claude วิเคราะห์ก่อน

Prompt:

> ```
> git push fail
> ช่วยอธิบาย error นี้แบบ non-coder และบอกวิธีแก้ที่ปลอดภัย
>
> Command:
> [paste command]
>
> Error:
> [paste error]
>
> อย่าเสนอ force push เว้นแต่จำเป็นจริง ๆ และอธิบายความเสี่ยงก่อน
> ```

Step 7 — เปิด Pull Request

บน GitHub หลัง push branch มักมีปุ่มให้เปิด PR

PR ควรมีข้อมูล:

> ทำอะไร
> ทำไมต้องทำ
> เปลี่ยนไฟล์สำคัญอะไร
> test อะไรแล้ว
> preview URL คืออะไร
> risk ที่ reviewer ควรดู

PR template:

```markdown
## What changed
- [change 1]
- [change 2]

## Why
[เหตุผล]

## Scope
This PR only covers:
- [scope]

Out of scope:
- [not included]

## Test plan
- [ ] local test
- [ ] preview test
- [ ] mobile check
- [ ] form validation
- [ ] security/data check if relevant

## Preview URL
[URL]

## Risks / reviewer notes
- [risk 1]
- [risk 2]

## Screenshots
[before/after if UI]
```

Prompt:

```
ช่วยเขียน PR description สำหรับ changes นี้
ใช้ template:
- What changed
- Why
- Scope
- Test plan
- Preview URL placeholder
- Risks / reviewer notes
- Screenshots

เขียนให้ non-coder และ reviewer เข้าใจ
```

Step 8 — ดู Vercel Preview URL

ถ้า Vercel เชื่อมกับ GitHub แล้ว เมื่อเปิด PR หรือ push branch Vercel จะสร้าง preview deployment

Preview URL อาจอยู่ใน:

```
GitHub PR checks/comments
Vercel dashboard > Deployments
```

ใช้ preview URL เพื่อ test ก่อน merge

อย่าคิดว่า “build ผ่าน = ใช้ได้”

ต้องเปิด URL และลอง flow จริง

Step 9 — Preview test checklist

ใช้ checklist นี้กับ PR ทุกอัน

```
Basics
[ ] Preview URL เปิดได้
[ ] ไม่มี blank screen
[ ] ไม่มี obvious layout break

Scope
[ ] change ที่ตั้งใจทำงานจริง
[ ] ไม่มี feature นอก scope โผล่

Mobile
[ ] mobile size อ่านได้
[ ] ปุ่มกดง่าย
[ ] form ใช้ได้

Data/security ถ้าเกี่ยวข้อง
[ ] ไม่มี secret ใน UI/error
[ ] form validation ทำงาน
[ ] user เห็นเฉพาะข้อมูลที่ควรเห็น
[ ] RLS/auth ไม่ถูกปิดหรือ bypass

Regression
[ ] flow หลักเดิมยังทำงาน
[ ] หน้าอื่นที่เกี่ยวข้องไม่พัง
```

Prompt:

```
ช่วยทำ preview test plan สำหรับ PR นี้

PR scope:
[scope]

ให้แยกเป็น:
1. test ที่ต้องทำแน่นอน
2. test ที่เกี่ยวกับ security/data
3. regression test ที่ควรทำ
4. test ที่ไม่จำเป็นในรอบนี้
```

Step 10 — ถ้า preview fail

อย่า merge

ให้เก็บ evidence:

```
preview URL
สิ่งที่กด
expected
actual
console/network error
Vercel log ถ้ามี
```

Prompt:

```
Preview deployment fail หรือใช้แล้วพัง
ช่วย debug จาก evidence นี้

PR:
[PR link/title]
Preview URL:
[URL]
Expected:
[ควรเกิดอะไร]
Actual:
[เกิดอะไร]
Evidence:
[paste log/error]

กติกา:
- อย่า merge
- วิเคราะห์ก่อนแก้
- แก้เฉพาะ PR scope
- หลังแก้ให้บอกว่าต้อง push update แล้ว test preview อะไรซ้ำ
```

เมื่อแก้แล้ว commit/push ใหม่ใน branch เดิม Vercel จะสร้าง preview ใหม่

Step 11 — Merge เข้า main

merge เมื่อ:

```
[ ] PR scope ชัด
[ ] preview test ผ่าน
[ ] ไม่มี blocker
[ ] ไม่มี secret/data risk
[ ] ถ้ามี reviewer เขา approve แล้ว
```

หลัง merge เข้า main Vercel มักสร้าง production deployment จาก main

อย่าปิดคอมทันที

ต้อง test production หลัง merge

Step 12 — Production smoke test หลัง merge

หลัง production deploy เสร็จ ให้ test 5 นาที

```
[ ] production URL เปิดได้
[ ] change ที่ merge มาอยู่จริง
[ ] flow หลักยังทำงาน
[ ] form/API/database ยังทำงาน
[ ] mobile ไม่พัง
[ ] ไม่มี error ใหม่
```

ถ้าพัง:

```
ถ้ากระทบ user จริง → พิจารณา rollback
ถ้าไม่กระทบหนัก → hotfix branch ใหม่
```

Prompt:

```
production หลัง merge มีปัญหา
ช่วยประเมิน rollback vs hotfix

PR ที่ merge:
[link/title]
Production issue:
[อธิบาย]
Evidence:
[log/error]
Impact:
[ใครกระทบ]

กติกา:
- ลดผลกระทบ user ก่อน
- ถ้า hotfix ให้ใช้ branch ใหม่ ไม่แก้มั่วบน main
```

Release note หลัง merge

จดสั้น ๆ เพื่อให้คุณในอนาคตรู้ว่า deploy อะไร

```markdown
# Release Note

## Date/time
[date]

## PR
[PR link]

## Production URL
[URL]

## What shipped
- [change]

## Tests done
- [ ] preview test
- [ ] production smoke test
- [ ] mobile
- [ ] data/security if relevant

## Result
[pass / issue / rollback]

## Follow-up
- [TODO]
```

ถ้าทำงานคนเดียว ยังต้อง PR ไหม?

ไม่จำเป็นทุกครั้ง

แต่แนะนำใช้ PR ถ้า change มีสิ่งเหล่านี้:

- database
- auth/permission
- env var
- deploy config
- payment
- dashboard/admin
- user data

- package ใหม่
- UI flow สำคัญ

ถ้าเป็น typo หรือ copy เล็กมาก อาจ commit เข้า main ได้ แต่ต้องรู้ว่า production อาจ deploy ตาม

ใช้ Claude Code Review ได้ไหม?

Claude Code มี feature Code Review สำหรับ GitHub PR ในบางแผน/องค์กร และสถานะบางอย่างอาจเป็น preview/research preview ตาม official docs

สำหรับหนังสือเล่มนี้ ให้ถือว่าเป็น optional

สิ่งที่ควรทำเสมอแม้ไม่มี automated review:

```
ให้ Claude ใน local review diff
ดู PR files changed ด้วยตัวเอง
test preview URL
ไม่ merge ถ้ายังมี red flag
```

Prompt local review:

```
ช่วย review PR นี้จาก diff ปัจจุบัน
หา:
- bug ที่เป็นไปได้
- regression
- security/data risk
- missing validation
- env var issue
- test ที่ขาด

ตอบเป็น Critical / High / Medium / Low
อย่าแก้ไฟล์ก่อน
```

Common problems

Problem 1: ไม่รู้ว่าอยู่ branch ไหน

ใช้:

```bash
git branch --show-current
```

ถาม Claude:

> ช่วยอธิบายผล `git branch --show-current` นี้
> และบอกว่าฉันกำลังแก้บน `main` หรือ `branch` อื่น

Problem 2: มีไฟล์แปลก ๆ ติดมาใน PR

อย่า commit ก่อนเข้าใจ

> ช่วยดู `git status` และบอกว่าไฟล์ไหนเกี่ยวกับ `task` นี้ ไฟล์ไหนน่าสงสัย
> อย่า `stage` หรือ `commit` เอง

Problem 3: merge แล้ว production พัง

ใช้ Appendix F เก็บ evidence และ Appendix D เรื่อง rollback

อย่า push hotfix มั่ว ๆ เข้า main โดยไม่มี test

Problem 4: PR ใหญ่เกินไป

ให้แตก PR

> `PR` นี้ใหญ่เกินไป
> ช่วยแยก `changes` เป็น `PR` ย่อยตาม `scope` และ `risk`
> บอกว่าอันไหนควรทำก่อนหลัง

Problem 5: conflict ตอน merge

สำหรับ non-coder ถ้า conflict เยอะ ให้หยุดและให้ Claude อธิบายก่อน

```
เกิด merge conflict
ช่วยอธิบายว่า conflict อยู่ไฟล์ไหน และเกี่ยวกับอะไร
อย่าแก้เองจนกว่าจะเสนอ plan
ถ้า conflict เสี่ยงกับ data/auth/security ให้บอกให้ human review
```

Final PR + Preview checklist

```
Before work
[ ] scope เล็กและชัด
[ ] branch ใหม่จาก main ล่าสุด

Before commit
[ ] diff review แล้ว
[ ] ไม่มี secret/.env
[ ] ไม่มี change นอก scope
[ ] test local ที่จำเป็นผ่าน

Before PR
[ ] push branch แล้ว
[ ] PR description ชัด
[ ] test plan ชัด

Before merge
[ ] preview deployment ผ่าน
[ ] preview URL test แล้ว
[ ] security/data check ผ่านถ้าเกี่ยวข้อง
[ ] ไม่มี blocker

After merge
[ ] production deploy ผ่าน
[ ] production smoke test ผ่าน
[ ] release note เขียนแล้ว
[ ] ถ้าพัง มี rollback/hotfix plan
```

Prompt รวมท้าย appendix

ใช้ prompt นี้เมื่อจะเริ่มงานรอบใหม่

```
ช่วยเป็น release workflow assistant
ฉันเป็น non-coder

Task:
[อธิบายงาน]

ช่วยพาฉันทำแบบ branch → PR → preview → merge

ก่อนเริ่ม:
- แนะนำ branch name
- แนะนำ scope ที่เล็กพอ
- บอก risk
- บอก test plan

ระหว่างทำ:
- review diff ก่อน commit
- เตือนถ้ามี secret/package/env/database/auth/deploy change
- ช่วยเขียน commit message และ PR description

ก่อน merge:
- สร้าง preview test checklist
- บอก blocker ถ้ามี

หลัง merge:
- สร้าง production smoke test checklist
- ช่วยเขียน release note

กติกา:
- อย่า push/merge/deploy เองถ้าฉันยังไม่ approve
- ถ้าต้อง force push ให้หยุดและอธิบายความเสี่ยงก่อน
- ถ้าเกี่ยวกับ data/security/payment ให้แนะนำ human review ถ้าจำเป็น
```

$ APPENDIX H

Setup Playbook สำหรับ Non-Coder

ใช้ appendix นี้เมื่อไหร่

ใช้เมื่อต้องเริ่มจากเครื่องเปล่า หรือยังไม่มั่นใจว่าเครื่องพร้อมใช้ Claude Code หรือยัง

เป้าหมายคือทำให้คุณมี environment ขั้นต่ำสำหรับสร้าง project เล็ก ๆ:

```
Terminal → Node.js/npm → Claude Code → Git/GitHub → project folder →
first safety check
```

ไม่ใช่สอน command line ทั้งหมด

เราจะใช้เฉพาะคำสั่งที่ต้องใช้จริงตอนเริ่ม vibe coding

ภาพรวมเครื่องมือที่ต้องมี

เครื่องมือ	ใช้ทำอะไร
Terminal	หน้าต่างสำหรับพิมพ์คำสั่ง
Node.js	runtime ที่ project web สมัยใหม่จำนวนมากใช้
npm	ตัวจัดการ package ที่มากับ Node.js
Claude Code	AI coding agent ที่ทำงานกับ project
Git	version control / save point
GitHub	ที่เก็บ repo บน cloud
Vercel	deploy web/app ออนไลน์

ถ้าอ่านแล้วเยอะ ให้จำแค่นี้:

```
Terminal = ที่สั่งงาน
Git = save point
GitHub = backup/share/deploy source
Claude Code = agent ที่ช่วย build
Vercel = เอาขึ้นเว็บ
```

Step 0 — อย่าเริ่มจาก project สำคัญ

ก่อนแตะ project จริง ให้สร้างพื้นที่ทดลอง

```
~/vibe-lab
```

หรือชื่ออื่นก็ได้

กฎ:

```
project แรกต้องพังได้
ไม่มีข้อมูลลูกค้าจริง
ไม่มี secret จริงที่สำคัญ
ไม่ใช่ production ของธุรกิจ
```

Prompt:

```
ช่วยวาง setup plan สำหรับฉันที่เป็น non-coder
เป้าหมายคือสร้าง project ทดลองที่พังได้
อย่าให้ฉันแตะ project สำคัญหรือ secret จริงตั้งแต่แรก
```

Step 1 — เปิด Terminal

Terminal คือที่พิมพ์คำสั่ง

ชื่ออาจต่างกันตามเครื่อง:

ระบบ	ชื่อที่มักใช้
macOS	Terminal หรือ iTerm
Windows	PowerShell, CMD, Windows Terminal
Windows + WSL	Ubuntu/WSL terminal
Linux	Terminal

ถ้าคุณใช้ Windows และจะใช้ Claude Code native installation official docs แนะนำ Git for Windows เพื่อให้ใช้ Bash tool ได้ดีขึ้น แต่ถ้าใช้ WSL ให้ติดตั้งและใช้ Claude Code ใน WSL terminal

ถ้าไม่รู้ว่าตัวเองอยู่ shell ไหน ให้ถาม Claude หรือดู prompt:

```
PS C:\Users\Name>     = PowerShell
C:\Users\Name>        = CMD
user@machine:~$       = macOS/Linux/WSL style shell
```

Step 2 — เช็ก Node.js และ npm

ใน terminal รัน:

```bash
node --version
npm --version
```

ถ้าได้ version กลับมา แปลว่ามี Node/npm แล้ว

ตัวอย่าง:

```
v24.16.0
11.x.x
```

ถ้า command not found ให้ติดตั้ง Node.js จาก official Node.js download page

ข้อควรจำ:

> Node.js มักมาพร้อม npm
> Claude Code npm install method ต้องใช้ Node.js 18 หรือใหม่กว่า

แต่ถ้าติดตั้ง Claude Code ด้วย native installer อาจไม่ต้องเริ่มจาก npm ก่อน

Step 2.5 — เช็กสิทธิ์ใช้งาน Claude Code

ก่อนเสียเวลาติดตั้ง ให้เช็กก่อนว่าบัญชีของคุณมีสิทธิ์ใช้ Claude Code ตาม official docs ล่าสุดหรือไม่

ณ source pack รอบนี้ official docs ระบุว่า Claude Code ต้องใช้ account/access ที่รองรับ เช่น Pro, Max, Team, Enterprise, Console account หรือ provider ที่รองรับ และ free Claude.ai plan ไม่รวม Claude Code access

ข้อสำคัญ:

> อย่าจำราคา/plan จากหนังสือเล่มนี้
> ให้เช็ก official docs/pricing ล่าสุดก่อนซื้อหรือก่อน publish หนังสือ

Prompt:

> ช่วยเช็กให้ฉันแบบ non-coder ว่าก่อนใช้ Claude Code ต้องมี account/access แบบไหน
> อย่าเดาราคา
> ให้บอกว่าข้อมูลไหนควรเช็กจาก official docs ล่าสุดก่อนตัดสินใจซื้อ

Prompt:

```
ฉันรัน node --version และ npm --version ได้ผลแบบนี้
ช่วยบอกว่าเครื่องพร้อมสำหรับ web project และ Claude Code ไหม

node:
[paste result]

npm:
[paste result]

อธิบายแบบ non-coder
```

Step 3 — ติดตั้ง Claude Code

Claude Code มีหลายวิธีติดตั้งตาม official docs

สำหรับ non-coder ให้เลือกวิธีที่เหมาะกับระบบของตัวเอง และเช็ก official docs ล่าสุดก่อน final เพราะ install command เปลี่ยนได้

macOS / Linux native installer

```bash
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell

```powershell
irm https://claude.ai/install.ps1 | iex
```

Windows CMD

```cmd
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del install
```

macOS Homebrew

```bash
brew install --cask claude-code
```

Windows WinGet

```powershell
winget install Anthropic.ClaudeCode
```

npm install option

```bash
npm install -g @anthropic-ai/claude-code
```

ข้อควรระวังจาก official docs:

> อย่าใช้ `sudo npm install -g @anthropic-ai/claude-code`

เพราะอาจทำให้เกิด permission issue และ security risk

Step 4 — Verify Claude Code

หลังติดตั้ง รัน:

```bash
claude --version
```

ถ้าได้ version กลับมา แปลว่า command ใช้งานได้

จากนั้นรัน diagnostic:

```bash
claude doctor
```

หรือถ้าอยู่ใน Claude Code session อาจใช้:

```
/doctor
```

ถ้า `claude --version` fail ด้วย `command not found` ให้ไปที่ common setup errors ด้านล่าง

Step 5 — Login Claude Code

รัน:

```bash
claude
```

แล้วทำตาม browser prompt / login flow ที่ขึ้นมา

หลัง login สำเร็จ คุณจะใช้ Claude Code ใน project folder ได้

ถ้า login loop, OAuth error, หรือ 403 ให้เก็บ error แล้วถาม Claude/ดู troubleshooting official docs

Prompt:

```
ฉัน login Claude Code ไม่ผ่าน
ช่วยวิเคราะห์แบบ non-coder

Error:
[paste error]

Environment:
- OS:
- terminal/shell:
- install method:

กติกา:
- อย่าขอ token หรือ credential
- แยกสาเหตุ network, browser login, organization, permission, install issue
```

Step 6 — ติดตั้ง Git และมี GitHub account

Git ใช้ทำ save point

GitHub ใช้ backup/share/deploy

เช็ก Git:

```bash
git --version
```

ถ้าไม่มี ให้ติดตั้งจาก official Git download หรือ GitHub Desktop ก็ได้สำหรับมือใหม่

สำหรับ Windows ถ้าติดตั้ง Git for Windows อย่าลืมปิดและเปิด terminal ใหม่หลังติดตั้ง

เช็ก GitHub:

```
คุณมี GitHub account แล้วหรือยัง
คุณสร้าง repository ได้ไหม
คุณ login ใน browser ได้ไหม
```

ยังไม่ต้องเข้าใจ Git ลึก

ขอแค่รู้ 4 คำ:

```
git status = ดูสถานะ
git add = เลือกไฟล์ที่จะ save
git commit = save point
git push = ส่งขึ้น GitHub
```

Step 7 — สร้าง folder ทดลอง

เลือกที่เก็บ project เช่น home folder

macOS/Linux/WSL:

```bash
mkdir -p ~/vibe-lab
cd ~/vibe-lab
```

Windows PowerShell:

```powershell
mkdir $HOME\vibe-lab
cd $HOME\vibe-lab
```

สร้าง project folder:

```bash
mkdir hello-vibe
cd hello-vibe
```

ตอนนี้คุณมี folder ทดลองแล้ว

Step 8 — เริ่ม Claude Code ใน folder project

ใน folder project รัน:

```bash
claude
```

คำสั่งแรกที่ควรถาม:

```
ช่วยดู folder นี้และบอกว่าตอนนี้เป็น project ว่างหรือมีอะไรอยู่
อย่าเพิ่งสร้างหรือแก้ไฟล์
อธิบายแบบ non-coder
```

จากนั้นให้ Claude ช่วยสร้าง starter files:

```
ช่วยสร้าง starter documentation สำหรับ project ทดลองนี้
ต้องมี:
- README.md
- CLAUDE.md
- .gitignore
- .env.example

กติกา:
- อย่าใส่ secret จริง
- ทำให้สั้นและอ่านง่าย
- project นี้เป็น playground ที่พังได้
```

Step 9 — สร้าง first commit

หลังมีไฟล์เริ่มต้น ให้เช็กสถานะ:

```bash
git status
```

ถ้ายังไม่ได้ init Git:

```
BASH
git init
```

เพิ่มไฟล์ที่ตั้งใจ commit:

```
BASH
git add README.md CLAUDE.md .gitignore .env.example
```

commit:

```
BASH
git commit -m "docs: add project starter files"
```

ถ้า Git ถามชื่อ/email ให้ตั้งค่าตามที่ Git แนะนำ หรือดู Git official docs

Prompt:

```
ช่วยอธิบายผล git status นี้แบบ non-coder
บอกว่าไฟล์ไหนควร commit และไฟล์ไหนไม่ควร commit
อย่า stage หรือ commit เอง

Output:
[paste git status]
```

Step 10 — สร้าง GitHub repo และ push

บน GitHub:

```
New repository
→ ตั้งชื่อ repo
→ เลือก private ถ้ายังทดลองหรือมีงานจริง
→ create repository
```

จากนั้น GitHub จะบอก command สำหรับเชื่อม local repo กับ remote

ก่อนรัน command ให้ถาม Claude:

```
GitHub ให้ command ชุดนี้มา
ช่วยอธิบายทีละบรรทัด และบอกว่าปลอดภัยไหม

Commands:
[paste commands]

กติกา:
- อย่าให้ฉัน force push
- เตือนถ้ามี risk เรื่อง secret หรือ .env
```

สำหรับ project แรก ถ้ายังไม่มั่นใจ จะใช้ GitHub Desktop แทน command line ก็ได้

เป้าหมายคือให้ code อยู่บน GitHub เพื่อ backup และ deploy ต่อได้

Step 11 — First Claude Code task ที่ปลอดภัย

อย่าเริ่มจาก “สร้าง SaaS ให้หน่อย”

เริ่มจาก task เล็ก:

```
สร้าง landing page static 1 หน้า
ไม่มี database
ไม่มี login
ไม่มี payment
ไม่มี secret
```

Prompt:

```
นี่คือ project ทดลอง
ช่วยสร้าง landing page static 1 หน้า

Scope:
- hero
- benefits 3 ข้อ
- CTA placeholder
- FAQ 3 ข้อ

กติกา:
- ไม่มี database
- ไม่มี login
- ไม่มี payment
- ไม่มี API key
- ก่อนแก้ไฟล์ให้เสนอ plan
- หลังแก้ให้บอกวิธี run/test
```

ถ้า Claude จะเพิ่ม package หรือ framework ให้ถามก่อน:

```
ทำไมต้องใช้ package/framework นี้
มีทางเลือกที่ง่ายกว่าสำหรับ project ทดลองไหม
```

Setup checklist

ก่อนถือว่าเครื่องพร้อม:

```
Terminal
[ ] เปิด terminal ได้
[ ] รู้ว่าใช้ macOS/Windows/WSL/Linux

Node/npm
[ ] node --version ได้ผล
[ ] npm --version ได้ผล

Claude Code
[ ] claude --version ได้ผล
[ ] claude doctor ผ่านหรือรู้ issue ที่ต้องแก้
[ ] login ได้

Git/GitHub
[ ] git --version ได้ผล
[ ] มี GitHub account
[ ] สร้าง repo ได้
[ ] commit แรกได้
[ ] push ขึ้น GitHub ได้

Project safety
[ ] มี .gitignore
[ ] มี .env.example แต่ไม่มี secret จริง
[ ] .env/.env.local ไม่ถูก commit
[ ] project แรกเป็น playground ที่พังได้
```

Common setup errors

1. `command not found: claude`

แปลว่า terminal หา command `claude` ไม่เจอ

สาเหตุที่เป็นไปได้:

- install ไม่สำเร็จ
- PATH ยังไม่อัปเดต
- เปิด terminal เดิมหลังติดตั้ง
- ติดตั้งคนละ environment เช่น Windows vs WSL

ลอง:

```
ปิด terminal แล้วเปิดใหม่
รัน claude --version อีกครั้ง
รัน claude doctor ถ้า command ใช้ได้
```

Prompt:

```
ฉันเจอ command not found: claude
ช่วย debug setup นี้

OS:
[macOS/Windows/WSL/Linux]
Install method:
[วิธีที่ใช้]
Terminal:
[PowerShell/CMD/Terminal/WSL]
Error:
[paste error]

อย่าให้ฉันรันคำสั่งเสี่ยงโดยไม่อธิบาย
```

2. ใช้ command ติดตั้งผิด shell บน Windows

ตัวอย่าง:

```
เอา macOS/Linux command ไปรันใน PowerShell
เอา PowerShell command ไปรันใน CMD
```

ให้ใช้ command ให้ตรง shell

PowerShell ใช้:

```powershell
irm https://claude.ai/install.ps1 | iex
```

CMD ใช้:

```
• CMD

curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del install
```

3. npm permission error

ถ้าใช้ npm install แล้วเจอ permission error อย่ารีบใส่ `sudo`

official docs เตือนว่าไม่ควรใช้ `sudo npm install -g` กับ Claude Code

ทางเลือก:

- ใช้ native installer
- แก้ npm permission ตามคำแนะนำ official
- ใช้ Node manager ที่เหมาะสม

4. Node/npm ใน WSL ปนกับ Windows

ถ้าใช้ WSL ให้ระวัง path ที่ขึ้นต้นด้วย `/mnt/c/`

แปลว่าอาจกำลังใช้ Node/npm ฝั่ง Windows ใน WSL

Prompt:

```
ฉันใช้ WSL และสงสัยว่า Node/npm ปนกับ Windows
ช่วยบอก command ที่ควรใช้ตรวจ เช่น which node และ which npm
แล้วอธิบายผลลัพธ์แบบ non-coder
```

5. Git push ไม่ได้

สาเหตุที่เป็นไปได้:

- ยังไม่ได้ login GitHub
- remote URL ผิด
- permission ไม่พอ
- branch name ไม่ตรง
- ยังไม่ได้ commit

Prompt:

```
git push ไม่ผ่าน
ช่วยวิเคราะห์ error นี้แบบ non-coder

Command:
[paste command]

Error:
[paste error]

กติกา:
- อย่าเสนอ force push ก่อน
- อธิบายว่า remote/branch/auth คืออะไรแบบง่าย ๆ
```

สิ่งที่ไม่ควรทำใน setup รอบแรก

อย่าทำสิ่งเหล่านี้ตอนยังไม่มั่นใจ:

```
ใช้ project production จริง
paste secret/API key ลง prompt
commit .env
ใช้ service_role key
deploy dashboard ที่ไม่มี auth
install package จำนวนมากโดยไม่รู้ว่าทำอะไร
ใช้ force push โดยไม่เข้าใจ
ใช้ sudo npm install -g โดยไม่จำเป็น
```

Setup ที่ดีคือ setup ที่ปลอดภัยและย้อนกลับได้

ไม่ใช่ setup ที่ทำเร็วที่สุด

Prompt รวมท้าย appendix

ใช้ prompt นี้ให้ Claude ช่วยเป็น setup assistant

```
ช่วยเป็น setup assistant สำหรับฉันที่เป็น non-coder

เป้าหมาย:
เตรียมเครื่องและ project ทดลองสำหรับใช้ Claude Code แบบปลอดภัย

ช่วยตรวจทีละขั้น:
1. terminal/shell ที่ฉันใช้
2. node --version
3. npm --version
4. claude --version
5. claude doctor
6. git --version
7. project folder
8. .gitignore / .env.example
9. git init / first commit
10. GitHub repo / push

กติกา:
- ถามทีละขั้น อย่าโยน command ยาว ๆ พร้อมกัน
- อธิบายทุก command แบบ non-coder
- อย่าให้ฉัน paste secret/token/password
- ถ้าเจอ error ให้เก็บหลักฐานก่อนแก้
- project แรกต้องเป็น playground ที่พังได้
```