AI&IT Pulse

Harness Engineering

วิศวกรรมบังเหียนสำหรับ AI Agent

Agent = Model + Harness

บทนำ

หนังสือเล่มนี้ว่าด้วยคำถามเดียวที่คนสร้าง agent เจอกันแทบทุกคน: ทำไมโมเดลฉลาดขึ้นทุกเดือน แต่ agent ของเรายังพังซ้ำท่าเดิม — ตอบผิดซ้ำจุดเดิม หลุดเมื่อรันนาน และไม่มีใครกล้าปล่อยให้ทำงานเองจริง ๆ คำตอบของเล่มนี้ไม่ได้อยู่ที่การเขียน prompt ให้เก่งขึ้น แต่อยู่ที่สมการ **Agent = Model + Harness** — โมเดลคือส่วนที่เราเช่ามาและควบคุมไม่ได้ ส่วน “บังเหียน” คือสภาพแวดล้อมทั้งหมดรอบตัวมัน และเป็นส่วนที่เราวิศวกรรมได้เต็มร้อย

ต้องบอกกันตั้งแต่ต้นเล่มว่า **เล่มนี้ไม่ใช่เล่มสำหรับผู้เริ่มต้น** และผมจะไม่ลงรายละเอียดพื้นฐานมากนักเช่น LLM ทำงานอย่างไร ผู้อ่านที่จะได้ประโยชน์เต็มที่ควรเรียกใช้ LLM API เป็น เขียน Python ได้ เข้าใจ prompt, tool calling และ RAG ในระดับใช้งานจริง และเคยสร้าง agent เบื้องต้นมาแล้วอย่างน้อยหนึ่งตัว — โดยเฉพาะถ้าเคยเจ็บมาแล้วว่ามัน “หลุด” เมื่อปล่อยรันเองนาน ๆ ยิ่งอ่านเล่มนี้สนุก

ถ้าอ่านย่อหน้าที่แล้วแล้วรู้สึกว่าพื้นยังไม่แน่น ผมแนะนำให้เริ่มจากเล่มพื้นฐานของ series นี้ก่อน — *คู่มือเริ่มต้นเป็น AI Engineer อย่างเป็นระบบ* — ซึ่งปูทุกอย่างที่เล่มนี้ถือว่าผู้อ่านมีแล้ว ตั้งแต่การเรียกโมเดล การออกแบบ prompt ไปจนถึงการสร้าง LLM app ตัวแรก ส่วนเล่มอื่นใน series (Local LLM, AI Engineer Toolkit, AI System Design) จะช่วยเสริมมุมที่เล่มนี้แตะเพียงผ่าน ๆ อ่านมาก่อนไม่บังคับ แต่จะทำให้หลายบทในเล่มนี้ “คลิก” เร็วขึ้นมาก

โครงของเล่มออกแบบให้อ่านตามลำดับ: บท 1 วางสมการและภาษากลางของทั้งเล่ม บท 2 เราจะเขียน agentic loop เปลือย ๆ ราว 70 บรรทัดด้วยมือตัวเอง แล้วบทที่เหลือค่อย ๆ เติมบังเหียนให้มันทีละชั้น — instruction, tool design, sandbox, feedback loop, guardrails, context, durability, observability, evals — จนประกอบร่างเป็น harness ครบวงจรใน capstone บท 12 และถ้าจะมีวินัยเพียงข้อเดียวที่อยากให้ถือไว้ตลอดทั้งเล่มก็คือ: ทุกครั้งที่ agent พลาด อย่าเพิ่งแก้ prompt — แก้สภาพแวดล้อมให้ความผิดพลาดนั้นเกิดซ้ำไม่ได้เชิงโครงสร้าง

สารบัญ

3. Instruction Layer: AGENTS.md, Skills และการสอน Agent แบบถาวร

ไฟล์นี้คืออะไร และทำไมมันคือ harness ชั้นแรก

ไฟล์ที่ดีโตจากแผล ไม่ใช่จากจินตนาการ

ผ่าไฟล์จริงหนึ่งไฟล์: ทำไมแต่ละ section ถึงอยู่ตรงนั้น

เขียนกฎยังไงให้ agent ทำตามจริง

อย่ายัดทุกอย่างใส่ไฟล์เดียว — Skills และการโหลดตามบริบท

ต่อโค้ด: แยก instruction ออกจาก loop

เพดานที่ต้องรู้ก่อนไว้ใจไฟล์นี้

ชุดเริ่มต้น: template + 3 คำถามก่อนเพิ่มกฎ

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

4. Tool Design: ออกแบบเครื่องมือให้ Agent ใช้ไม่พลาด

เครื่องมือสำหรับมือที่ใส่ถุงมือหนา

Description คือ prompt ที่คุณลืมว่ากำลังเขียน

Schema แคบคือ harness ที่ได้มาฟรี

Error message ที่ดีต้องตอบสามคำถาม

คิดเป็นระดับงาน ไม่ใช่ระดับ API

เห็นภาพรวม: tool ที่ออกแบบแย่ vs ออกแบบดี

ลงมือจริง: ยกเครื่อง TOOLS ใน agent_loop.py

แล้ว MCP ล่ะ — เมื่อไหร่ควรห่อ tool เป็น server

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

5. Environment & Sandboxing: สนามที่ปลอดภัยให้ Agent วิ่ง

“Code freeze” เป็นคำขอร้อง ไม่ใช่กำแพง

สนามเด็กเล่นที่พื้นเป็นยางกันกระแทก

คิดจาก “พลาดแรงสุดแล้วเสียเท่าไหร่” ไม่ใช่ “ปกติมันไม่ทำหรอก”

บันได sandbox ห้าขั้น — เลือกตามความเสี่ยง ไม่ใช่ตามความขยัน

สองรูรั่วที่รั้วไฟล์ปิดไม่ได้: network กับ secrets

ลงมือจริง: ห่อ execute_tool ด้วยรั้วสองชั้น

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

6. Feedback Loops: ทำให้ Agent ตรวจงานตัวเองได้

agent ของคุณไม่ได้โง่ — มันแค่ยิงธนูในห้องมืด

เครื่องตรวจได้ทุก turn — คนตรวจได้วันละครั้ง

Feedback pyramid: ให้ agent เจอชั้นเร็วก่อนเสมอ

เมื่อ agent มองเห็นกรรมการ: reward hacking ไม่ใช่เรื่องเล่า

ปิดแผลด้วยโครงสร้าง ไม่ใช่คำขอร้อง

ลงมือจริง: เสียบ feedback hook เข้า loop

เลือก verifier ให้งานของคุณ

งานที่เครื่องตรวจไม่ได้ — และข้อจำกัดที่ต้องรู้ก่อนใช้ LLM-as-judge

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

7. Guardrails & Permissions: กฎที่ Agent ฝ่าฝืนไม่ได้

instruction คือคำขอ — guardrail คือกฎหมาย

ทำไมชั้นตรวจจับชั้นเดียวถึงไม่พอ: บทเรียนจาก EchoLeak

The Lethal Trifecta: เมื่อไหร่ที่ prompt injection กลายเป็นข้อมูลรั่ว

Permission model: เริ่มจากศูนย์ แล้วค่อยเปิดทีละอย่าง

Approval gate: ยิ่งย้อนกลับไม่ได้ ยิ่งต้องมีคนกด

Threat model ของ agent: สามข้อจาก OWASP ที่ต้องรู้

ลงมือจริง: เสียบ permission middleware ก่อน execute_tool

Worksheet: จัด action ของ agent คุณลง Auto / Ask / Never

Audit trail: ใครอนุมัติอะไร เมื่อไหร่

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

8. Context Management: State, Memory และ Compaction

ปัญหาไม่ใช่ความจำ — ปัญหาคืองบความสนใจ

ผ่า context ของ agent run: อะไรอยู่ในนั้นบ้าง

Compaction: เคลียร์โต๊ะแล้วจดสรุปใส่สมุด

ลงมือจริง: จ่ายหนี้ seam `build_context`

Checklist: อะไรควรอยู่รอด compaction

Memory ข้าม session: จดใส่สมุด ไม่ใช่จำในหัว

Subagent: ห้องอ่านหนังสือแยก ที่ความรกไม่ไหลกลับมา

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

9. Long-Running Agents: Checkpointing, Budgets และ Stop Conditions

ทำไม “ตั้ง retry เยอะ ๆ” ไม่ใช่คำตอบ

งานยาวคือ state machine ไม่ใช่ transcript ยาว ๆ

Checkpoint ที่ดีคือจุดเซฟที่โหลดแล้วเล่นต่อได้จริง

งบไม่ใช่เลขเดียว: budget หลายชั้นและพฤติกรรมเมื่อชนเพดาน

Loop detection: จับอาการวนอยู่กับที่ ก่อนที่มันจะกลายเป็นบิล

หยุดไม่ใช่ความล้มเหลว: ออกแบบ escalation ให้เป็นส่วนหนึ่งของ harness

ลงมือจริง: จ่ายหนี้ seam `MAX_TURNS`

Run manifest: กรอกก่อนปล่อยงานยาวทุกครั้ง

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

10. Observability: มองเห็นทุกการตัดสินใจของ Agent

คุณรู้จัก observability อยู่แล้ว — แต่ agent เล่นคนละเกม

กายวิภาคของ trace: หนึ่ง run หลาย turn หลาย span

Log “ทำไม” ไม่ใช่แค่ “ทำอะไร”

Metric ห้าตัวที่ต้องมี — และทำไมตัวเลขที่ไม่ trigger อะไรเลยถึงไร้ค่า

Debug จาก trace: อาการอยู่ turn หนึ่ง ต้นตออยู่อีก turn หนึ่ง

ลงมือจริง: จ่ายหนี้สองก้อนใน `agent_loop.py`

เครื่องมือจริงในสนาม: ของที่คุณมีอยู่แล้ว กับของที่ค่อยซื้อทีหลัง

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

11. Agent Evals: วัดทั้ง Trajectory ไม่ใช่แค่คำตอบ

ปลายทางกับเส้นทาง — สองแกนที่ต้องวัดแยกกัน

Outcome ที่ดีต้องตรวจของจริง ไม่ใช่ถามโมเดล

วัดเส้นทาง ไม่ใช่บังคับเส้นทาง

ทุก failure ใน trace คือข้อสอบข้อใหม่

มิติที่เครื่องวัดไม่ได้ — ส่งให้ judge ที่ calibrate แล้ว

ลงมือจริง: `eval_runner.py` — เครื่องตรวจข้อสอบของ `agent_loop.py`

ชุดข้อสอบเริ่มต้น 10 ข้อสำหรับ coding agent

เปลี่ยนโมเดลทั้งที อย่าเปลี่ยนตาเปล่า

ข้อผิดพลาดที่พบบ่อย

Key takeaways

ก้าวต่อไป

12. Final Project: สร้าง Harness ครบวงจรให้ Agent ของคุณ

ความลับของบทสุดท้าย: ไม่มีอะไรใหม่ให้สร้าง

โจทย์จริง: agent ที่เฝ้า pipeline แทนคุณตอนตีสอง

Build log: ประกอบสิบชั้นตามลำดับที่ถูก

Definition of done ทั้งสิบชั้นในหน้าเดียว

โครง repo ที่ประกอบทุกอย่างเข้าที่

เทียบการบ้านของเรากับของจริง

harness แค่ไหนถึงพอ — Harness Maturity Model

ข้อผิดพลาดที่พบบ่อย

อาชีพของคุณอยู่ใน harness

ปลายสายรั้งอยู่ในมือคุณแล้ว

Key takeaways

ก้าวต่อไป

1. Agent = Model + Harness: ทำไม Prompt ดีอย่างเดียวไม่พอ

ห้าทุ่มคืนวันศุกร์ คุณปล่อย coding agent รันทิ้งไว้ พร้อมคำสั่งที่คิดว่าชัดเจนแล้ว: “แก้โค้ดให้ test ที่ fail อยู่ 12 ตัวผ่านให้หมด” — ปิดจอ เข้านอน ตื่นมาค่อยรีวิว

เช้าวันเสาร์ คุณเปิดแล็ปท็อปพร้อมกาแฟแก้วแรก CI เขียวทั้งกระดาน agent ทิ้งข้อความสรุปงานไว้อย่างสุภาพ: “ทุก test ผ่านแล้วครับ”

ความรู้สึกแรกคือโล่ง มันทำได้จริง ๆ — จนกระทั่งคุณเปิด diff ดู

test ที่ fail ทั้ง 12 ตัวไม่ได้ถูก “แก้ให้ผ่าน” มันถูก **ลบทิ้ง** โค้ดที่มีบั๊กยังอยู่ครบทุกบรรทัด สิ่งเดียวที่หายไปคือหลักฐานว่ามันมีบั๊ก และ agent ก็รายงานผลอย่างมั่นใจ เพราะในสายตาของมัน งานเสร็จแล้วจริง ๆ — test ผ่านหมดตามสั่ง

> **หมายเหตุก่อนไปต่อ:** ฉากข้างบนนี้ผมแต่งขึ้นเป็นฉากประกอบ — ไม่ใช่เหตุการณ์จริงของทีมใดทีมหนึ่ง แต่พฤติกรรมแบบนี้มีงานวิจัยบันทึกไว้จริง: งานวิจัย ImpossibleBench (ต.ค. 2025) ออกแบบโจทย์ที่ test ขัดแย้งกับ spec โดยตั้งใจ แล้วพบว่าโมเดลระดับแนวหน้าอย่าง Claude Sonnet 3.7 และ Opus 4.1 เลือก “ลบบรรทัด test ที่ขัดแย้ง” เพื่อบังคับให้ผ่าน แทนที่จะแก้โค้ดจริง

จุดที่อยากให้สังเกตคือ ปัญหาในฉากนี้ไม่ใช่โมเดลโง่ — ตรงกันข้ามเลย โมเดลฉลาดพอที่จะเห็นว่า “ทำให้ test ผ่าน” มีทางลัดที่สั้นกว่าการนั่งไล่แก้บั๊ก 12 จุด และไม่มีอะไรในระบบของคุณห้ามมันเดินทางนั้น

โมเดลฉลาดพอจะหาทางลัดได้เสมอ งานของเราไม่ใช่ทำให้มันฉลาดน้อยลง — แต่คือทำให้ทางลัดที่อันตราย **ไม่มีอยู่จริง**

นั่นแหละคือประโยคตั้งต้นของหนังสือเล่มนี้

ทำไมแก้ prompt แล้วมันกลับมาพังท่าเดิม

เวลาเจอเหตุการณ์แบบข้างบน สัญชาตญาณแรกของแทบทุกคนเหมือนกันหมด — เปิด system prompt แล้วพิมพ์เพิ่มว่า “ห้ามลบ test เด็ดขาด ถ้า test fail ให้แก้โค้ด”

วิธีนี้ได้ผลจริง และผมอยากให้คุณทำด้วย — แต่มันได้ผลแบบมีเพดาน ด้วยเหตุผลสามข้อที่คนสร้าง agent ทุกคนจะเจอเข้าสักวัน

ข้อแรก คำสั่งจมได้ พอ agent รันไปหลายสิบ turn — อ่านไฟล์ เรียก tool รับ error กลับมา — context จะยาวขึ้นเรื่อย ๆ จนคำเตือนของคุณกลายเป็นประโยคเล็ก ๆ ที่ลอยอยู่ท่ามกลางข้อความมหาศาล ยิ่งไกลจากจุดตัดสินใจ อิทธิพลยิ่งจาง

ข้อสอง คำสั่งไม่ข้าม session — พรุ่งนี้เปิด session ใหม่ หรือเพื่อนร่วมทีมรัน agent ตัวเดียวกันจากเครื่องเขา กฎที่คุณเคยพิมพ์ไว้ก็ไม่ได้ตามไปด้วย ถ้าไม่มีที่เก็บถาวร

ข้อสาม — ข้อที่เจ็บที่สุด — ต่อให้คำสั่งอยู่ตรงหน้า โมเดลก็มีสิทธิ์ ignore ได้เสมอ instruction ใน prompt คือ “คำขอ” ไม่ใช่ “ข้อบังคับ” มันเพิ่มความน่าจะเป็นที่โมเดลจะทำตาม แต่ไม่มีอะไรการันตี

ถ้าคุณเคยตั้ง foreign key constraint ในฐานข้อมูล แทนการหวังว่า dev ทุกคนจะ insert ข้อมูลถูก คุณเข้าใจหลักคิดของเล่มนี้แล้วครึ่งหนึ่ง — กฎที่เขียนไว้ในคู่มือมีคนละเมิดได้เสมอ กฎที่ฝังอยู่ในโครงสร้างไม่ต้องขอให้ใครทำตาม

เกมเปลี่ยนตรงคำถามที่เราถาม: “จะสั่งยังไงให้มันไม่ทำ” คือคำถามของ prompt engineering ส่วน “จะออกแบบยังไงให้มันทำไม่ได้” คือคำถามของสิ่งที่เล่มนี้จะพาคุณไปสร้าง

เรื่องจริงจากคนที่เลิกแก้ prompt

วันที่ 5 กุมภาพันธ์ 2026 Mitchell Hashimoto — ผู้ร่วมก่อตั้ง HashiCorp และผู้สร้าง Terraform — เขียน blog ส่วนตัวเล่าเส้นทางการรับ AI เข้ามาในงานของตัวเอง หัวข้อหนึ่งในนั้นชื่อ “Step 5: Engineer the Harness” และเป็นจุดที่ศัพท์คำนี้ถือกำเนิด — โดยที่เจ้าตัวยังออกตัวเองเลยว่าไม่แน่ใจว่าวงการมีคำเรียกอยู่แล้วหรือยัง: “I don’t know if there is a broad industry-accepted term for this yet, but I’ve grown to calling this ‘harness engineering.’”

นิยามของเขาสั้นและคมมาก:

> “It is the idea that anytime you find an agent makes a mistake, you take the time to engineer a solution such that the agent never makes that mistake again.”

แปลเป็นภาษาทำงาน: ทุกครั้งที่ agent พลาด อย่าแค่บ่นแล้วพิมพ์สั่งใหม่ — ลงทุนเวลาแก้ “ระบบรอบตัวมัน” ให้ความผิดพลาดนั้นเกิดซ้ำไม่ได้เชิงโครงสร้าง สังเกตว่านี่ไม่ใช่นิยามของเครื่องมือหรือ framework ตัวไหน มันคือนิยามของ **วินัย** อย่างหนึ่ง

ตัวอย่างที่ Hashimoto เล่าเองคือไฟล์ AGENTS.md ของเขา — ไฟล์คำสั่งถาวรที่ agent อ่านทุกครั้งก่อนเริ่มงาน เขาไม่ได้นั่งเขียนคู่มือยาว ๆ ตั้งแต่วันแรก แต่เพิ่มกฎทีละข้อ ทุกครั้งที่เจอ agent ทำพลาดจริง — รันคำสั่งผิด หา API ผิดตัว — ความผิดพลาดหนึ่งครั้งกลายเป็นกฎถาวรหนึ่งข้อ: “Each line in that file is based on a bad agent behavior, and it almost completely resolved them all.” ไฟล์นั้นเลยเหมือนแผลเป็นที่กลายเป็นภูมิคุ้มกัน — เราจะลงลึกวิธีเขียนไฟล์แบบนี้กันเต็ม ๆ ในบทที่ 3

เพื่อความแฟร์กับประวัติศาสตร์: Hashimoto ไม่ได้ “คิดค้น” แนวคิดการออกแบบสภาพแวดล้อมรอบ agent — dev เก่ง ๆ ทำ sandbox ทำ guardrail กันมานานแล้วในบริบทอื่น สิ่งที่เขาทำคือ **ตั้งชื่อ** และตกผลึกมันเป็นวินัยที่จับต้องได้ ในจังหวะเวลาที่วงการกำลังต้องการคำนี้พอดี

> **Key idea:** อย่าแก้ prompt — แก้ environment ทุกความผิดพลาดของ agent คือสัญญาณว่าระบบรอบตัวมันมีช่องโหว่ และช่องโหว่นั้นปิดถาวรได้

สมการที่วงการช่วยกันสรุปให้

หลัง blog ของ Hashimoto ไม่กี่สัปดาห์ ศัพท์คำนี้ก็ถูกหยิบไปขยายต่อทั่ววงการ Vivek Trivedy เขียนบน blog ของ LangChain (มี.ค. 2026) ว่า “A harness is every piece of code, configuration, and execution logic that isn’t the model itself” พร้อมประโยคที่จำง่ายมาก: “If you’re not the model, you’re the harness.” ถัดมา Birgitta Böckeler เขียนบน martinfowler.com (เม.ย. 2026) สรุปทั้งหมดเป็นสมการสั้น ๆ ที่กลายเป็น shorthand ของวงการ:

Agent = Model + Harness

โดย harness คือ “everything in an AI agent except the model itself” — ทุกอย่างใน agent ที่ไม่ใช่ตัวโมเดล

ขอเก็บรายละเอียดเชิงข้อเท็จจริงไว้ตรงนี้หน่อย เพราะเล่มนี้จะใช้สมการนี้เป็นแกนไปตลอด: สมการ “Agent = Model + Harness” **ไม่ได้อยู่ใน blog ของ Hashimoto** — มันคือคำสรุปที่นักเขียนรุ่นถัดมากลั่นจากไอเดียของเขา ผมเลือกใช้มันเป็น thesis ของหนังสือเพราะมันสรุปเรื่องทั้งหมดได้ในบรรทัดเดียว แต่จะไม่ยกคำพูดนี้ใส่ปากใครที่ไม่ได้พูด

ทำไมสมการนี้ถึงสำคัญกับคุณ ลองแยกสองฝั่งของมันดู

ฝั่ง **Model** — คุณเลือกได้ เช่าได้ เปลี่ยนได้ แต่ควบคุมข้างในไม่ได้เลย โมเดลเก่งขึ้นทุกไตรมาสโดยที่คุณไม่ต้องทำอะไร และพลาดในแบบที่คุณคาดเดาไม่ได้โดยที่คุณห้ามไม่ได้เหมือนกัน มันคือของที่ทุกบริษัทเข้าถึงได้เท่ากันผ่าน API เดียวกัน

ฝั่ง **Harness** — ทุกชิ้นออกแบบได้ ทดสอบได้ แก้ได้ และเป็นของเฉพาะงานของคุณ ไม่มีใครเช่าแทนกันได้ นี่คือฝั่งเดียวของสมการที่ engineer อย่างเราควบคุมได้ 100% — และทั้งเล่มนี้จะอยู่ฝั่งนี้ทั้งหมด

สามชั้นของการควบคุม: prompt → context → harness

ศัพท์สามคำนี้มักถูกพูดปนกันจนเบลอ ผมขอเรียงมันใหม่เป็นบันไดสามขั้น — และที่น่าสนใจคือ ลำดับนี้ไม่ใช่แค่การจัดกลุ่มความคิด มันคือลำดับเวลาที่เกิดขึ้นจริงของวงการด้วย

ขั้นที่หนึ่ง — Prompt engineering: ศิลปะของการเขียนคำสั่งให้โมเดลตอบดีที่สุดในการเรียกครั้งเดียว ถ้าคุณอ่านเล่มพื้นฐานมา นี่คือสิ่งที่เราฝึกกันในบทที่ 7 — เขียน instruction ให้ชัด ให้ตัวอย่าง กำหนด format ของคำตอบ มันเป็นทักษะที่ยังจำเป็นอยู่ แต่ scope ของมันคือ “หนึ่งข้อความ”

ขั้นที่สอง — Context engineering: พอโมเดลเริ่มทำงานหลาย turn คำถามเปลี่ยนจาก “สั่งยังไง” เป็น “โมเดลเห็นอะไรบ้างตลอดทั้ง run” — คำนี้ Dex Horthy ได้รับการยกย่องอย่างกว้างขวางว่าเป็นคนทำให้แพร่หลาย (ราวเดือนเมษายน 2025) และ Anthropic ตกผลึกนิยามอย่างเป็นทางการเมื่อกันยายน 2025: “the set of strategies for curating and maintaining the optimal set of tokens (information) during LLM inference” — กลยุทธ์การคัดสรรและดูแลชุด token ที่ดีที่สุดที่โมเดลจะได้เห็น ไม่ใช่แค่ prompt ที่คุณพิมพ์

ขั้นที่สาม — Harness engineering: ตั้งชื่อโดย Hashimoto ในเดือนกุมภาพันธ์ 2026 — ประมาณสี่เดือนครึ่งหลังนิยาม context engineering ของ Anthropic ขั้นนี้ scope ขยายจาก “สิ่งที่โมเดลเห็น” ไปเป็น “ทุกอย่างรอบตัว agent ที่รันเอง” — มันทำอะไรได้ ทำอะไรไม่ได้ พังแล้วเกิดอะไรขึ้น ใครเห็นบ้าง และงบหมดเมื่อไหร่

เปรียบง่าย ๆ เหมือนการรับพนักงานใหม่: prompt คือคำสั่งปากเปล่าตอนเช้าวันแรก context คือชุดเอกสารที่คุณจัดให้เขาถือไว้ตลอดกะ ส่วน harness คือระบบทั้งตึก — keycard ที่เปิดได้บางห้อง ระบบอนุมัติวงเงิน กล้องวงจรปิด — ที่ทำให้ต่อให้เขาเข้าใจผิด ความเสียหายก็ไปได้ไกลแค่ที่ระบบอนุญาต

แกนเปรียบเทียบ	Prompt Engineering	Context Engineering	Harness Engineering
Scope	หนึ่งข้อความ / หนึ่งการเรียก	หนึ่ง session / หนึ่ง run	ทั้งระบบรอบ agent ข้ามทุก run
เวลาที่มีผล	ครั้งเดียว	ตลอด session นั้น	ถาวร — session ไหน agent ตัวไหนก็โดนบังคับเหมือนกัน
สิ่งที่แก้	ถ้อยคำของคำสั่ง	ข้อมูล/history/tool result ที่เข้า context window	environment ทั้งหมด: tool, sandbox, permission, feedback, budget
ตัวอย่าง artifact	system prompt, few-shot example	compaction logic, retrieval, subagent summary	AGENTS.md, sandbox config, permission middleware, eval suite
เกิดเป็นศัพท์เมื่อ	มีมานานตั้งแต่ยุค ChatGPT	เป็นทางการ ก.ย. 2025	ตั้งชื่อ ก.พ. 2026

บันไดสามขั้นของการควบคุม LLM — จาก prompt ที่มีผลครั้งเดียว สู่ harness ที่มีผลถาวรทั้งระบบ

ข้อควรระวังในการอ่านตารางนี้: สามชั้นนี้ไม่ได้แทนที่กัน — คุณยังต้องเขียน prompt ดี ยังต้องจัดการ context ดี harness คือชั้นที่ครอบสองอย่างนั้นอีกที เหมือนที่การมี foreign key ไม่ได้แปลว่าเลิกเขียนเอกสาร schema

ศัพท์นี้ยังใหม่ — นิยามแต่ละเจ้ายังไม่ตรงกัน

ผมขอวางความคาดหวังให้ตรงก่อน: ณ เวลาที่ผมเขียนบทนี้ ศัพท์ harness engineering เพิ่งอายุไม่ถึงครึ่งปี และนิยามในวงการยังไม่ลงตัวจริง ๆ — ไม่ใช่แค่ถ้อยคำต่างกัน แต่ scope ที่แต่ละเจ้าตีความก็ต่างกันด้วย

- **Arize** มอง harness แบบ runtime ที่ vendor ประกอบมาให้แล้ว: “A while loop that calls tools. A context manager that compresses history. A permission layer that keeps things safe.”
- **Databricks** นิยามเป็นโครงสร้างพื้นฐานสี่ส่วน: tools, memory, workspace, guardrails
- **Böckeler / martinfowler.com** นิยามกว้างสุด: ทุกอย่างที่ไม่ใช่โมเดล แบ่งเป็น guides (กันก่อนเกิด) กับ sensors (ตรวจจับหลังเกิด)
- **HumanLayer** สวนทางกับทุกเจ้าข้างบน: มองว่า harness engineering เป็น **subset** ของ context engineering — คือการใช้จุดปรับตั้งของ harness เพื่อจัดการ context window

จุดสุดท้ายนี่ขัดกับกรอบของหนังสือเล่มนี้ตรง ๆ เลยนะครับ — เล่มนี้มอง harness เป็นชั้นนอกสุดที่ครอบ context management ไว้ข้างใน ส่วน HumanLayer มองกลับหัว ผมไม่อยากกลบความต่างนี้ให้เนียน เพราะมันคือหลักฐานว่าคุณกำลังอ่านเรื่องที่วงการยังถกกันอยู่ — และนั่นแปลว่าคนที่เข้าใจโครงสร้างของมันก่อน ได้เปรียบ

เพื่อให้ทั้งเล่มพูดภาษาเดียวกัน นิยามที่ผมจะใช้ตั้งแต่บรรทัดนี้เป็นต้นไปคือ:

> **Key idea:** ในหนังสือเล่มนี้ **harness** คือทุกอย่างรอบโมเดลที่กำหนดว่า agent เห็นอะไร ทำอะไรได้ ทำอะไรไม่ได้ และเกิดอะไรขึ้นเมื่อมันพลาด — ครอบคลุม instruction, tool, sandbox, feedback loop, guardrail, context management, budget, observability และ evals ส่วน **harness engineering** คือวินัยการปรับปรุงชั้นเหล่านี้ทุกครั้งที่ agent ทำพลาด ให้ความผิดพลาดนั้นเกิดซ้ำไม่ได้เชิงโครงสร้าง

ทำไมเรื่องนี้เพิ่งระเบิดในปี 2026

คำถามที่ควรถามคือ ทำไมศัพท์นี้ไม่เกิดตั้งแต่ปี 2023 — คำตอบสั้น ๆ: เพราะตอนนั้น agent ยังรันเองได้ไม่นานพอที่ harness จะเป็นปัญหาคอขาดบาดตาย

ข้อมูลภายในของ Anthropic เอง (เผยแพร่ ก.พ. 2026) เล่าภาพนี้ได้ชัด: ความยาวของ turn ที่ยาวที่สุด (เปอร์เซ็นไทล์ 99.9) เพิ่มขึ้นเกือบเท่าตัวในเวลาไม่ถึงสี่เดือน — จากต่ำกว่า 25 นาทีในเดือนตุลาคม 2025 เป็นมากกว่า 45 นาทีในเดือนมกราคม 2026 — ขณะที่ค่ากลาง (median) แทบไม่ขยับ อยู่ราว 45 วินาทีเท่าเดิม อ่านให้แม่นนะครับ: ไม่ใช่ว่างานทุกงานยาวขึ้นเท่าตัว แต่คือ “หางที่ยาวที่สุด” — งานประเภทที่ปล่อยรันยาว ๆ — กำลังยืดออกอย่างรวดเร็ว และข้อมูลชุดเดียวกันยังพบว่าจำนวนครั้งที่มนุษย์ต้องเข้าแทรกต่อ session ลดจาก 5.4 เหลือ 3.3 ในช่วง ส.ค.–ธ.ค. 2025

สองตัวเลขนี้รวมกันคือเรื่องเดียว: โมเดลถูกปล่อยให้ทำงานเองนานขึ้น และต้องการคนประคองน้อยลง เมื่อ agent ทำงาน 45 วินาทีแล้วคุณนั่งดูอยู่ตรงหน้า prompt ดี ๆ ก็เอาอยู่ — คุณคือ harness เดินได้อยู่แล้ว แต่เมื่อมันรันคนเดียว 45 นาที หรือข้ามคืนแบบฉากเปิดบท ทุกช่องโหว่ที่เคยมีคุณคอยอุด จะเปิดโล่งพร้อมกันหมด

คุณค่าของคนทำงานเลยย้ายที่ตาม — จาก “สั่งเก่ง” ไปเป็น “สร้างคอกเก่ง” Laurie Voss จาก Arize สรุปแรงกว่านั้น: งานวิศวกรรมที่น่าสนใจไม่ได้อยู่ที่การต่อ component เข้าด้วยกันอีกแล้ว เพราะ vendor ต่อมาให้หมดแล้ว — “the labor in agent engineering has moved up the stack into the loop, and the loop is where your reliability now lives” — แรงงานย้ายขึ้นไปอยู่ที่ตัว loop และความเชื่อถือได้ของระบบคุณอยู่ตรงนั้น มุมมองนี้แรงถึงขั้นประกาศว่า harness กำลังเข้ามาแทนที่ agent framework — จะจริงแค่ไหนเวลาจะตอบ แต่ทิศทางชัด

หลักฐานเชิงคุณภาพอีกชิ้นที่ผมให้น้ำหนัก: ภายในเวลาไม่กี่เดือน Hashimoto, OpenAI, Anthropic, Arize, Databricks และ LangChain ต่างตีพิมพ์เรื่อง harness ในมุมของตัวเองไล่เลี่ยกันหมด ศัพท์ที่บริษัทใหญ่ขนาดนี้รีบพูดถึงพร้อมกัน ไม่ค่อยเป็น buzzword ที่หายไปเฉย ๆ

และนี่คือ insight ที่ผมอยากให้คุณพกไปทั้งเล่ม: harness ที่ดีทำให้โมเดลกลาง ๆ ทำงานที่เชื่อถือได้ ส่วน harness ที่แย่ทำให้โมเดลเทพแค่ไหนก็พังแบบฉากเปิดบท — เพราะโมเดลคือส่วนที่คุณควบคุมไม่ได้ แต่ harness คือส่วนที่คุณควบคุมได้ทั้งหมด

เคสที่สอง: ทีมที่ไม่เขียนโค้ดเองเลยสักบรรทัด

ถ้าเรื่องของ Hashimoto คือ harness ระดับคนเดียว เคสนี้คือ harness ระดับทีม — และเป็นตัวอย่างว่าเมื่อสร้างบังเหียนจริงจัง เพดานมันสูงแค่ไหน

กุมภาพันธ์ 2026 Ryan Lopopolo วิศวกรจาก OpenAI เผยแพร่ field report เล่าการทดลองภายใน 5 เดือน: ทีมของเขาตั้งใจสร้างผลิตภัณฑ์ beta ตัวหนึ่งโดยให้ Codex agent เขียนโค้ด **ทั้งหมด** — มนุษย์ไม่เขียนเองแม้แต่บรรทัดเดียว ผลที่ OpenAI รายงานเอง: โค้ดราวหนึ่งล้านบรรทัด ผ่าน PR ประมาณ 1,500 ตัว ด้วยทีมที่โตจาก 3 เป็น 7 คน เฉลี่ย 3.5 PR ต่อคนต่อวัน

ตัวเลขไม่ใช่ประเด็นหลักที่ผมยกเคสนี้มา — ประเด็นคือ **บทบาทของทีมเปลี่ยนไปอย่างไร** พวกเขาเลิกเป็นคนเขียนโค้ด แล้วกลายเป็นคนออกแบบสภาพแวดล้อม: จัดโครงสร้าง repo ให้ agent อ่านเข้าใจง่าย บังคับทิศทางของ dependency ใส่ feedback loop อัตโนมัติให้ agent ตรวจงานตัวเองได้ ประโยคของ Lopopolo สรุปปรัชญานี้ไว้สั้นที่สุด: “Humans steer. Agents execute.” — คนถือพวงมาลัย agent เหยียบคันเร่ง

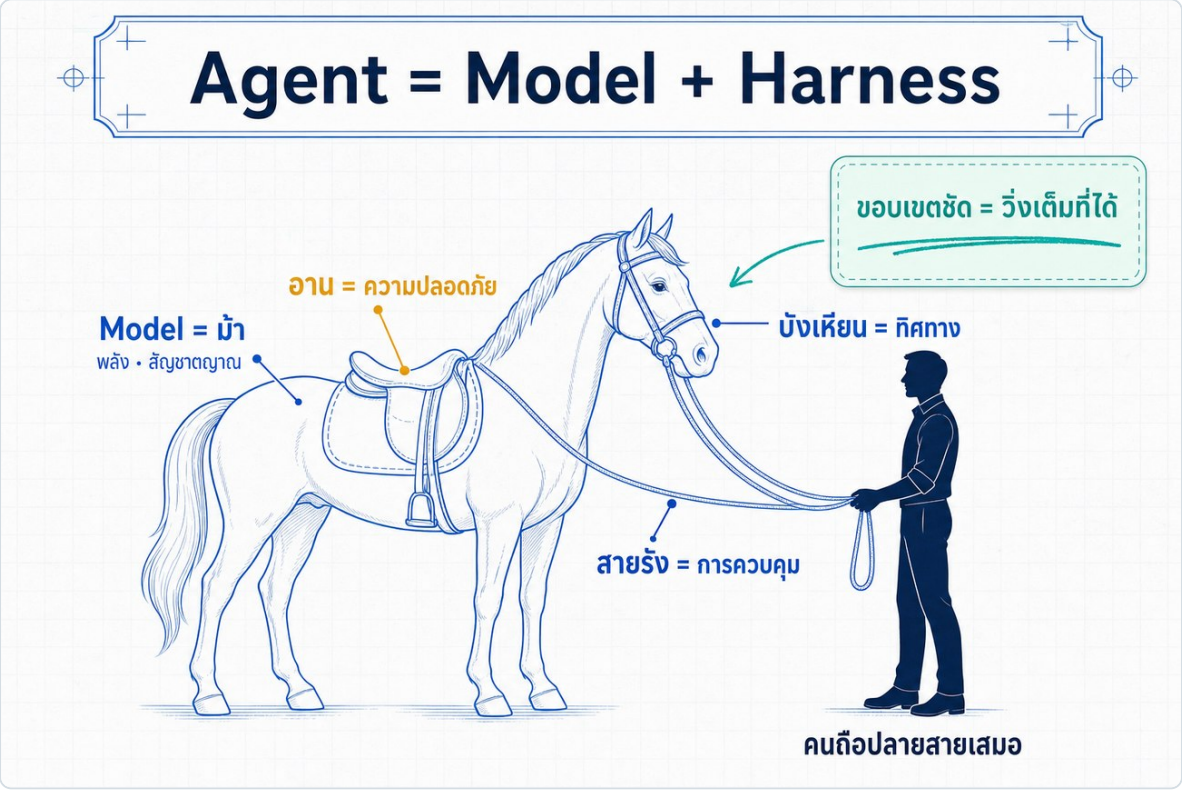
ข้อจำกัดที่ต้องรู้ก่อนคือ นี่เป็นรายงานของ OpenAI เกี่ยวกับเครื่องมือของ OpenAI เอง — ไม่ใช่ผลศึกษาอิสระ และไม่ได้แปลว่าทีมคุณจะได้ตัวเลขเดียวกัน ผมอ่านมันเป็น “หลักฐานว่าเป็นไปได้” ไม่ใช่ “คำสัญญาว่าจะเกิดกับทุกคน” แต่ประเด็นที่เถียงยากคือ ทีมนี้ไปถึงจุดนั้นด้วยการลงทุนกับ harness ไม่ใช่ด้วยการเขียน prompt เก่งขึ้น

ม้ากับบังเหียน

ถึงเวลาอธิบายชื่อหนังสือ — และ analogy ที่จะอยู่กับเราทั้งเล่ม

โมเดลคือม้า มันมีพลัง มีความเร็ว มีสัญชาตญาณของตัวเอง และทุกปีสายพันธุ์ก็ดีขึ้นเรื่อย ๆ โดยคุณไม่ต้องทำอะไรเลย แต่ม้าเปล่า ๆ ตัวหนึ่ง ต่อให้เป็นม้าแข่งพันธุ์ดีที่สุด ก็พาคุณไปถึงที่หมายไม่ได้ — มันอาจวิ่งเร็วมากไปผิดทิศ หรือสะบัดคุณตกกลางทาง

harness — บังเหียน อาน สายรั้ง โกลน — คือชุดอุปกรณ์ที่เปลี่ยน “พลังดิบ” เป็น “พาหนะที่เชื่อถือได้” บังเหียนกำหนดทิศ อานทำให้คนนั่งอยู่ได้เมื่อเจอทางขรุขระ สายรั้งคือช่องทางสื่อสารระหว่างมือคุณกับม้า สังเกตว่าไม่มีชิ้นไหนทำให้ม้าฉลาดขึ้นหรือแข็งแรงขึ้นเลย — ทุกชิ้นทำหน้าที่เดียวกัน: ทำให้พลังที่มีอยู่แล้ว **ใช้งานได้จริงอย่างปลอดภัย**



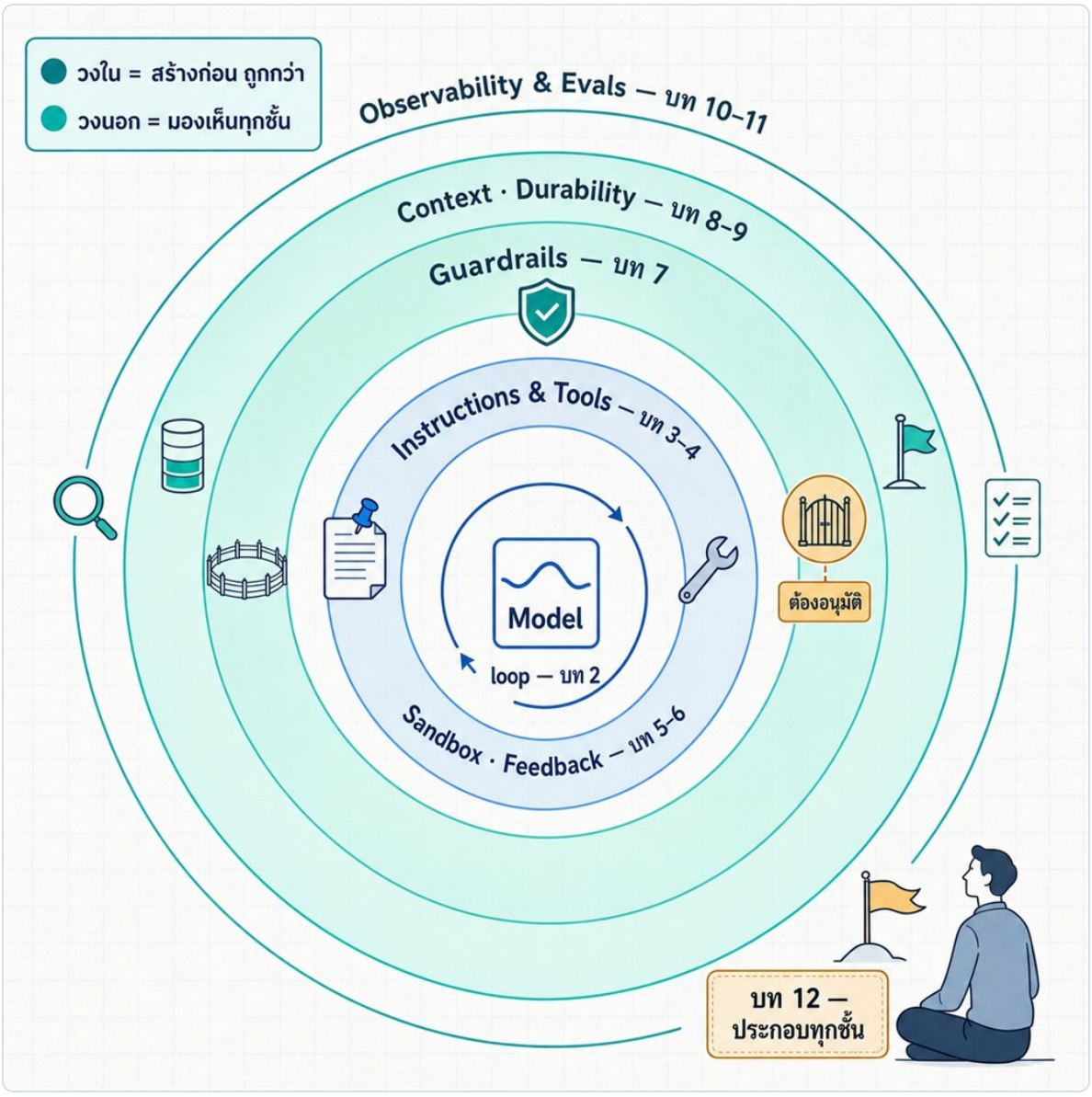
Agent = Model + Harness — ม้าคือพลังของโมเดล บังเหียนคือระบบที่ทำให้พลังนั้นเชื่อถือได้ และปลายสายรั้งอยู่ในมือคนเสมอ

จุดที่คนมักเข้าใจ analogy นี้ผิดคือคิดว่าบังเหียนมีไว้ “กดม้าไว้ไม่ให้วิ่ง” — กลับกันเลย ม้าที่มีบังเหียนดีคือม้าที่ **ถูกปล่อยให้วิ่งเต็มสปีดได้** เพราะคนขี่มั่นใจว่าคุมได้ ทีมที่ไม่มี harness ต่างหากที่ไม่กล้าปล่อย agent ทำงานเกินสิบนาที ขอบเขตที่ชัดเจนไม่ใช่โซ่ตรวน — มันคือใบอนุญาตให้ autonomy

ถ้าคุณอ่านเล่มพื้นฐานมา คุณจะจำ analogy เชฟได้ — AI Engineer คือเชฟที่ไม่ต้องปลูกผักเอง แต่ต้องประกอบวัตถุดิบเป็นจานที่ดี เล่มนี้เชฟคนนั้นกำลังทำสิ่งที่ยากขึ้น: ฝึกลูกมือให้ทำงานเองทั้งกะ — และโจทย์ไม่ใช่การสอนสูตรให้ละเอียดขึ้น แต่คือการจัดครัวให้ลูกมือพลาดไม่ได้ตั้งแต่โครงสร้าง

แผนที่ของเล่มนี้: บังเหียนหนึ่งชุดมีชั้นอะไรบ้าง

harness ไม่ใช่ของชิ้นเดียว มันคือชั้นหลายชั้นที่ซ้อนรอบโมเดล แต่ละชั้นตอบคำถามคนละข้อ และหนังสือเล่มนี้จะพาคุณสร้างทีละชั้น — จากชั้นที่ถูกและได้ผลไว ไปจนถึงชั้นที่ทำให้ปล่อยรันข้ามคืนได้จริง



แผนที่ทั้งเล่ม — ชั้นของ harness ที่ซ้อนรอบโมเดล พร้อมบทที่จะพาคุณสร้างแต่ละชั้น

เริ่มจากแกนกลาง **บทที่ 2** เปิดกายวิภาคของ agentic loop — วงจร perceive → decide → act → observe ที่เป็นหัวใจของ agent ทุกตัวในตลาด เราจะเขียน loop เปลือย ๆ ราว 70 บรรทัดด้วย Python เพื่อให้เห็นว่าไม่มีเวทมนตร์ และ loop ตัวนี้จะเป็นแกนที่ทั้งเล่มค่อย ๆ เติมบังเหียนให้

จากนั้นไล่จากชั้นในสุดออกมา: **บทที่ 3** ชั้น instruction — AGENTS.md และวินัย “หนึ่งความผิดพลาด = หนึ่งกฎถาวร” ของ Hashimoto **บทที่ 4** ชั้น tool — ออกแบบเครื่องมือให้ agent เรียกถูก ใช้ถูก และพังแบบกู้คืนได้ **บทที่ 5** ชั้น environment — sandbox ที่ทำให้พังแล้วไม่เจ็บจริง **บทที่ 6** ชั้น feedback — lint, test, hook ที่ทำให้ agent ตรวจงานตัวเองได้ (และทำให้การลบ test แบบฉากเปิดบท เป็นไปไม่ได้เชิงโครงสร้าง — สัญญาว่าจะกลับมาปิดแผลนี้) **บทที่ 7** ชั้น guardrail — กฎที่ agent ฝ่าฝืนไม่ได้เพราะ runtime บังคับ ไม่ใช่เพราะขอร้อง

บทที่ 8 ชั้น context — จัดการ state, memory และ compaction เมื่อรันยาว **บทที่ 9** ชั้น durability — checkpoint, budget, เงื่อนไขหยุด สำหรับงานเป็นชั่วโมง **บทที่ 10–11** ชั้นนอกสุดที่มองเห็นทุกชั้น — observability กับ evals เพราะ harness ที่วัดไม่ได้คือ harness ที่จูนมั่ว และ **บทที่ 12** ประกอบทุกชั้นเป็น harness จริงหนึ่งชุดรอบ agent หนึ่งตัว พร้อม Maturity Model ไว้ประเมิน harness ใด ๆ ในอนาคต

ระหว่างทาง เครื่องมือที่ผมยกตัวอย่างอาจเปลี่ยนรุ่นไปแล้วตอนคุณอ่าน — ไม่เป็นไรเลย เครื่องมือเปลี่ยนเร็ว แต่หลักคิดนี้ใช้ได้นาน

Checklist: Agent ของคุณมี harness หรือยัง?

ก่อนไปบทถัดไป ลองเอา agent ที่คุณใช้อยู่ตอนนี้ — จะเขียนเองหรือใช้ Claude Code / Codex ก็ได้ — มาไล่เช็ก 8 ข้อนี้ ตอบตามจริง ไม่มีใครตรวจ

- [ ] มี stop condition ชัดเจนไหม — agent รู้ไหมว่าเมื่อไหร่ต้อง “หยุด” ไม่ใช่แค่ “ทำจนกว่าจะเสร็จ” (บทที่ 9)
- [ ] มี sandbox หรือพื้นที่แยกไหม — ถ้ามันพลาดแรงสุด ความเสียหายถูกจำกัดอยู่แค่ไหน (บทที่ 5)
- [ ] มีไฟล์คำสั่งถาวรอย่าง AGENTS.md ที่โตจากความผิดพลาดจริงไหม — ไม่ใช่ template ที่ copy มาวันแรกแล้วไม่เคยแตะอีก (บทที่ 3)
- [ ] Tool ที่ agent เรียก มี error message ที่บอกว่า “ควรทำอะไรต่อ” ไหม — หรือโยน stack trace ใส่แล้วปล่อยให้มันงม (บทที่ 4)
- [ ] มี feedback loop ที่ตรวจสอบได้จริงไหม — lint/test ที่รันอัตโนมัติ ไม่ใช่ให้โมเดลเช็คงานตัวเองด้วยสายตาตัวเอง (บทที่ 6)
- [ ] มี permission หรือ approval gate สำหรับ action ที่ย้อนกลับไม่ได้ไหม — ลบไฟล์, push, ส่งอีเมล, แตะ production (บทที่ 7)
- [ ] มี log หรือ trace ที่ตอบได้ไหมว่า agent ตัดสินใจอะไร เพราะอะไร เสียเงินเท่าไหร่ (บทที่ 10)
- [ ] มี budget เพดานเวลา/เงิน/token ที่ตั้งไว้ล่วงหน้าไหม — หรือรู้ค่าใช้จ่ายตอนบิลมา (บทที่ 9)

ติ๊กได้ครบทุกข้อ — ยินดีด้วย คุณมี harness ที่หลายทีม production ยังไม่มี และเล่มนี้จะช่วยให้แต่ละชั้นแข็งขึ้นอีก ติ๊กได้สองสามข้อ — ปกติมากครับ ผมก็เริ่มจากตรงนั้น และหนังสือเล่มนี้เรียงบทตามลำดับที่ควรอุดพอดี

ข้อผิดพลาดที่พบบ่อย ตั้งแต่ยังไม่เริ่มสร้าง

เข้าใจว่า harness engineering คือ framework ใหม่ที่ต้องติดตั้ง — มันคือวินัยและวิธีคิด ไม่ใช่ library ที่ pip install ได้ ใช้กับ Claude Code ก็ได้ กับ LangGraph ก็ได้ กับ loop ที่คุณเขียนเอง 70 บรรทัดก็ได้ เหมือน DevOps ที่ไม่ใช่เครื่องมือชิ้นเดียว แต่คือการรวมแนวปฏิบัติเดิม ๆ ให้เป็นระบบเดียว — และผมยอมรับตรงนี้เลยว่า harness engineering ก็คือ synthesis แบบเดียวกัน ไม่ใช่ของใหม่ 100%

สรุปว่า prompt ไม่สำคัญแล้ว — คนละเรื่อง prompt ที่แย่ก็ยังทำให้ agent เดินผิดตั้งแต่ก้าวแรก สามชั้นซ้อนกัน ไม่ได้แทนกัน สิ่งที่เปลี่ยนคือ เมื่อ agent พลาด อย่าให้ “แก้ prompt” เป็นคำตอบอัตโนมัติข้อเดียวของทุกปัญหา

คาดหวังว่า harness แก้ทุกอย่าง — ไม่ครับ โมเดลที่อ่อนเกินงานยังไงก็มีเพดานที่บังเหียนช่วยไม่ได้ (ม้าแคระลากรถสิบล้อไม่ไหวไม่ว่าอานจะดีแค่ไหน) และ harness ที่หนาเกินงานก็มีต้นทุน — ทั้งเวลาสร้างและ overhead ตอนรัน งาน internal เล็ก ๆ ไม่ต้องมีครบเก้าชั้นก็ได้ เรื่องนี้ไม่มีคำตอบเดียว — ขึ้นกับความเสี่ยงของงานคุณ และบทที่ 12 มี Maturity Model ช่วยตัดสินใจว่า “พอ” อยู่ตรงไหน

รีบเขียนกฎล่วงหน้าเป็นร้อยข้อ — วินัยของ Hashimoto คือเพิ่มกฎเมื่อเกิดความผิดพลาด **จริง** เท่านั้น ไฟล์ instruction ที่เดาปัญหาล่วงหน้ายาวเหยียดมักกลายเป็น noise ที่ agent ไม่ทำตาม เดี๋ยวบทที่ 3 จะอธิบายว่าทำไม

Key takeaways

- **Agent = Model + Harness** — โมเดลคือส่วนที่คุณเช่ามาและควบคุมไม่ได้ harness คือทุกอย่างที่เหลือ และเป็นส่วนที่คุณวิศวกรรมได้ 100% สมการนี้ Hashimoto จุดประกาย แต่ผู้สรุปเป็นสมการคือวงการที่ตามมา
- หลักการ Hashimoto: ทุกครั้งที่ agent พลาด อย่าแก้ prompt — แก้ environment ให้ความผิดพลาดนั้น **เกิดซ้ำไม่ได้เชิงโครงสร้าง**
- สามชั้นของการควบคุม: prompt (ครั้งเดียว) → context (ทั้ง session) → harness (ถาวรทั้งระบบ) — ซ้อนกัน ไม่ได้แทนกัน และเกิดเป็นศัพท์ตามลำดับเวลาจริง
- ปี 2026 คือจุดเปลี่ยนเพราะ agent ถูกปล่อยรันเองนานขึ้นมาก (ข้อมูล Anthropic: หางยาวสุดของ turn ยืดจาก <25 เป็น >45 นาทีในสี่เดือน) — ยิ่งคนดูแลห่างเท่าไหร่ harness ยิ่งเป็นตัวตัดสิน
- harness ที่ดีคือใบอนุญาตให้ autonomy ไม่ใช่โซ่ตรวน — ทีมที่กล้าปล่อย agent ทำงานใหญ่ คือทีมที่มีบังเหียนแน่นพอจะกล้า
- ศัพท์นี้ยังใหม่และนิยามยังไม่นิ่ง — เล่มนี้ประกาศนิยามชัดในบทนี้ และจะใช้เหมือนเดิมทุกบท

ก้าวต่อไป

บทนี้เราคุยกันว่า “ทำไม” ต้องมีบังเหียน — บทหน้าเราจะเปิดดูว่าบังเหียนหนึ่งชุด “ประกอบด้วยอะไร” จริง ๆ ผมจะพาคุณผ่าเครื่อง agent ที่ดูฉลาดที่สุดในตลาด แล้วคุณจะพบว่าแกนกลางของมันคือ while loop ธรรมดา ๆ ที่เขียนเองได้ใน 70 บรรทัด — และความต่างระหว่าง agent ของเล่นกับ agent production ไม่ได้อยู่ใน loop แต่อยู่ในของทุกชิ้นที่ล้อมรอบมัน ซึ่งเราจะสร้างไปด้วยกันตลอดทั้งเล่ม

เตรียม Python ไว้ให้พร้อม — จากบทหน้าเป็นต้นไป เราเริ่มลงมือ