คู่มือสำหรับ Developer ที่ใช้ Git และ Terminal อยู่แล้ว

จาก Issue สู่ Change ที่พร้อม Review พร้อมหลักฐาน Verification

Agentic Coding ด้วย Claude Code

1. ใช้ Claude Code เป็นคู่หู

สร้างระบบอัตโนมัติและ
เวิร์กโฟลว์แบบ Agentic ได้ดั่งใจ

2. สร้างและต่อยอด แอป Next.js

พัฒนาแอป Next.js
ต่อยอดได้เรื่อยๆ
โดยมี AI ช่วยทุกขั้นตอน

3. สเกลงานไม่ลด คุณภาพโค้ด

ขยายการพัฒนาด้วย AI
พร้อมรักษาคุณภาพโค้ดไว้ครบ

Agentic Coding with Claude Code

คู่มือสร้าง workflow ที่ตรวจสอบได้สำหรับ developer ที่ใช้ Git และ terminal อยู่แล้ว

พัฒนาต่อยอดจาก *Agentic Coding with Claude Code* โดย Eden Marco และแหล่งเรียนรู้ด้าน Claude Code ระดับสากล

เรียบเรียงและจัดทำโดย **Agentic Stack**

หมายเหตุเกี่ยวกับฉบับเรียบเรียงภาษาไทย

eBook ฉบับนี้จัดทำโดย **Agentic Stack** ในลักษณะงานรวบรวม สังเคราะห์ ทดลอง และเรียบเรียงขึ้นใหม่จากแนวปฏิบัติของนักพัฒนา ผู้สร้างเครื่องมือ นักการศึกษา และนักวิจัยด้าน Agentic Coding ในต่างประเทศ รวมถึงแหล่งเผยแพร่สาธารณะของ Anthropic ตลอดจนผลงานและบทเรียนของ Eden Marco, Boris Cherny, Lydia Hallie, Elie Schoppik, Matt Pocock และ Dr. Jules White เนื้อหาถูกปรับให้เหมาะกับผู้อ่านไทย พร้อมคำอธิบาย ตัวอย่าง และบริบทการใช้งานที่ผู้จัดทำพัฒนาขึ้น มิใช่ฉบับแปลหรือฉบับจัดพิมพ์อย่างเป็นทางการของบุคคลหรือองค์กรใด

เนื่องจาก Claude Code พัฒนาอย่างรวดเร็ว ผู้จัดทำจะทบทวน แก้ไข และเสริมเนื้อหาให้สอดคล้องกับ feature, workflow และแนวปฏิบัติใหม่อย่างต่อเนื่อง โดยคงศัพท์เทคนิคและชื่อ feature ในรูป English แบบ canonical เพื่อรักษาความหมายทางเทคนิคและให้นำไปใช้งานจริงได้สะดวก

สิทธิ์อัปเดตตลอดชีพ (Lifetime Updates) ผู้ที่ได้รับ eBook อย่างถูกต้องจากช่องทางของผู้จัดทำมีสิทธิ์รับไฟล์ฉบับปรับปรุงของ eBook เล่มนี้โดยไม่มีค่าใช้จ่ายเพิ่มเติมตลอดอายุการให้บริการของผลิตภัณฑ์ โดยตรวจสอบฉบับล่าสุดได้จากเลข version และ release date ภายในไฟล์

ฉบับภาษาไทย

สารบัญ

Agentic Coding with Claude Code

คำนำ

generative AI และ coding agent กำลังเปลี่ยนแปลงวิธีสร้าง software อย่างรวดเร็ว tool อย่าง Claude Code, Cursor และ LangChain deep agents ไม่ได้จำกัดอยู่เพียง interaction แบบ prompt ง่าย ๆ อีกต่อไป tool เหล่านี้ทำงานครอบคลุมทั้ง codebase ผสานรวมกับ tool ภายนอก และเรียกใช้ workflow แบบ multi-step ที่ทำงานระยะยาว เมื่อระบบเหล่านี้มีความสามารถมากขึ้น การเข้าใจวิธีทำงานและวิธีควบคุมระบบจึงกลายเป็นสิ่งจำเป็น

หนังสือเล่มนี้มุ่งสร้างความเข้าใจที่แข็งแกร่งและนำไปใช้ได้จริงเกี่ยวกับ **Claude Code**, agentic workflow และ concept เบื้องหลังที่ทำให้ coding agent สมัยใหม่ทำงานได้อย่างมีประสิทธิภาพ theme หลักตลอดทั้งเล่มคือ Context Engineering ซึ่งอธิบายว่า agent สร้าง จัดการ และใช้ context อย่างไรในระหว่างที่ให้เหตุผลและลงมือทำ เมื่อ agent ซับซ้อนและขับเคลื่อนด้วย tool มากขึ้น การจัดการ context อย่างเหมาะสมจะส่งผลโดยตรงต่อ performance ต้นทุน reliability และความสม่ำเสมอ

ในหนังสือเล่มนี้ คุณจะได้เรียนรู้ว่า Claude Code โต้ตอบกับ codebase อย่างไร วิธีกำหนดค่าสำหรับการใช้งานประจำวัน และวิธีต่อขยายด้วย **Model Context Protocol** (**MCP**) คุณจะได้สำรวจ automation ด้วย GitHub Actions planning อย่างมีโครงสร้าง workflow แบบ multi-agent subagent Output style และ Agent Skills หนังสือยังพิจารณาหัวข้อเชิง architecture ที่ลึกยิ่งขึ้น รวมถึงวิธีทำงานของ deep agent และวิธีนำไปใช้ workflow แบบ long-horizon ในทางปฏิบัติ

ฉบับปรับปรุงนี้เพิ่มกล่อง **Expert Practice** ซึ่งสังเคราะห์แนวปฏิบัติจากผู้สร้างเครื่องมือ นักพัฒนา นักการศึกษา และนักวิจัยที่เผยแพร่ประสบการณ์เกี่ยวกับ Claude Code ต่อสาธารณะ เช่น Boris Cherny, Ray Amjad, Avthar Sewrathan, Nick Saraev, Lydia Hallie, Elie Schoppik, Matt Pocock, Dr. Jules White และ Paul Goldsmith-Pinkham กล่องเหล่านี้ถูกวางไว้ข้าง concept ที่เกี่ยวข้อง แยกข้อความที่ตรวจสอบจาก source ได้ออกจากข้อเสนอเชิงบรรณาธิการ และให้ link กลับไปยังแหล่งเรียนรู้ของเจ้าของเนื้อหา

แทนที่จะมุ่งเน้นเพียงการใช้งานระดับพื้นผิว หนังสือเล่มนี้มีเป้าหมายช่วยให้คุณเข้าใจว่าระบบเหล่านี้ได้รับ design อย่างไร และเหตุใดจึงมีพฤติกรรมเช่นนั้น ด้วยการผสานตัวอย่างเชิงปฏิบัติเข้ากับ insight ด้าน architecture ที่ลึกกว่า เป้าหมายคือมอบทั้ง skill สำหรับใช้งาน coding agent สมัยใหม่อย่างมีประสิทธิภาพ และความเข้าใจที่จำเป็นต่อการปรับตัวในขณะที่วงการนี้พัฒนาต่อไป

หนังสือเล่มนี้เหมาะกับใคร

หนังสือเล่มนี้เหมาะสำหรับ developer และ engineer ที่เขียน code ใช้ Git และทำงานใน terminal หรือ IDE ได้อยู่แล้ว และต้องการก้าวพ้น AI แบบ chat-based ไปสู่ agentic workflow ที่ตรวจสอบได้ด้วย Claude Code และ MCP หนังสือเขียนขึ้นสำหรับผู้ที่มีประสบการณ์ด้าน software engineering มาก่อน และต้องการผสาน AI agent แบบ context-aware เข้ากับ workflow จริง ทำให้งาน coding ที่ซับซ้อนเป็นระบบ และออกแบบ multi-agent workflow ที่ควบคุมขอบเขตได้

นอกจากนี้ หนังสือยังเป็นประโยชน์ต่อ AI engineer และผู้ปฏิบัติงานด้าน generative AI ที่ทำงานใกล้ชิดกับ development workflow สมัยใหม่

เพื่อให้ได้ประโยชน์สูงสุดจากหนังสือเล่มนี้ คุณควรมีประสบการณ์เขียนและแก้จุดบกพร่อง code ด้วย Python หรือ TypeScript คุ้นเคยกับ Git, branch, package manager และการรัน test ใน development environment ตลอดจนเข้าใจ concept หลักของ generative AI เช่น LLM, RAG และ agent หนังสือเล่มนี้ไม่ใช่คู่มือเริ่มเขียนโปรแกรมหรือเริ่มใช้ command line จากศูนย์

หากมีพื้นฐานเหล่านี้แล้วและต้องการเห็นผลก่อนอ่านเชิงลึก ให้เริ่มที่ *Quick Start: ส่ง change แรกให้ review ได้ใน 60 นาที* แล้วใช้ *Implementation Kit* สร้าง artifact จริงใน repository ของคุณ

เนื้อหาที่หนังสือเล่มนี้ครอบคลุม

บทที่ 1, *Context Engineering* แนะนำ Context Engineering และอธิบายว่าเหตุใดจึงมีความสำคัญต่อการสร้าง AI agent สมัยใหม่ บทนี้ครอบคลุมว่า context วิวัฒนาการมาจาก prompt engineering อย่างไร เติบโตขึ้นในระบบที่ซับซ้อนอย่างไร และการจัดการ context ที่ไม่มีประสิทธิภาพส่งผลต่อ performance และต้นทุนอย่างไร นอกจากนี้ ยังพิจารณา system prompt และการจัดการ context เชิงปฏิบัติด้วย Claude Code

บทที่ 2, *แก่นสำคัญของ Claude Code* นำเสนอข้อมูลเบื้องต้นเชิงปฏิบัติเกี่ยวกับ Claude Code และอธิบายว่า Claude Code โต้ตอบกับ codebase อย่างไร บทนี้ครอบคลุมการเริ่มต้น project file `CLAUDE.md` permission และการจัดการ context ทั้งยังแนะนำ MCP, design แบบ spec-driven และเริ่มสร้าง HookHub project ที่จะใช้ตลอดทั้งเล่ม

บทที่ 3, *เริ่มต้นใช้งาน Claude Code – สำรวจ command สำคัญ* มุ่งเน้นการกำหนดค่า Claude Code สำหรับใช้งานประจำวัน บทนี้ครอบคลุมราคาและ authentication command configuration ระดับ user และระดับ project ตลอดจน integration กับ Cursor นอกจากนี้ ยังสำรวจ hook checkpointing Skills และบทบาทของ Language Server Protocol

บทที่ 4, *ขยายขีดความสามารถของ Claude Code ด้วย MCP server และ plugin* อธิบาย Model Context Protocol และวิธีที่ protocol นี้สร้างมาตรฐานให้ integration ระหว่าง AI application กับ tool ภายนอก บทนี้ครอบคลุม architecture ของ MCP ซึ่งรวมถึง host client และ server พร้อมสาธิตวิธีกำหนดค่า MCP server ทั้งแบบ local และ remote นอกจากนี้ ยังกล่าวถึงการจัดการ context ข้อพิจารณาด้าน performance และการต่อขยายด้วย plugin

บทที่ 5, *ทำให้ development workflow เป็นอัตโนมัติด้วย Claude Code และ GitHub* สำรวจ integration ระหว่าง Claude Code กับ GitHub บทนี้ครอบคลุม configuration ของ repository automation สำหรับ pull request และ issue รวมถึงการใช้ GitHub Actions เพื่อเรียกใช้ workflow ทั้งยังอธิบายว่า YAML workflow file นิยาม automation แบบ event-driven อย่างไร

บทที่ 6, *planning ด้วย Claude Code และ workflow แบบ multi-agent* พิจารณา agentic workflow ที่มีโครงสร้างผ่าน planning และการทำงานแบบ multi-agent ที่ประสานกัน บทนี้อธิบายว่า spec-driven development ช่วยเพิ่มความสามารถในการคาดการณ์ได้อย่างไร และ agent หลายตัวสามารถร่วมงานกันภายใน codebase เดียวกันได้อย่างไร

บทที่ 7, *ทำงานกับ subagent ของ Claude Code* แนะนำ subagent ของ Claude Code และอธิบายว่า subagent ทำให้เกิด workflow ที่มีโครงสร้างและแยกจากกันได้อย่างไร บทนี้ครอบคลุม subagent configuration แบบ custom flow ของ context และการทำงานพร้อมกันด้วย pattern Infinite Agentic Loop

บทที่ 8, *สร้างและปรับแต่ง Output style* พิจารณาว่า Output style กำหนดรูปแบบ response ของ Claude Code อย่างไร บทนี้ครอบคลุมการสร้างและกำหนด scope ให้ custom style format ที่มีโครงสร้างอย่าง YAML และการทำให้พฤติกรรมภายในนิยามของ style เป็นอัตโนมัติ ทั้งยังอธิบายวิธีจัดการบทบาทและ configuration ของ session

บทที่ 9, *ทำความเข้าใจ Agent Skills* สำรวจ Agent Skills ในฐานะกลไกสำหรับต่อขยายความสามารถของ AI agent บทนี้ครอบคลุม concept พื้นฐาน การใช้งานจริงใน Claude Code และ LangChain DeepAgent รวมถึง flow ภายในของ context เมื่อมีการ invoke skill ก่อนปิดท้ายด้วยตัวอย่าง implementation จริง

บทที่ 10, *ใช้งาน Claude Code Desktop* อธิบายวิธีใช้ Claude Code ภายใน desktop application บทนี้ครอบคลุมการสลับระหว่าง local mode กับ cloud mode การเรียกใช้ agent แบบ parallel ด้วย Git worktree และ orchestration feature branch ข้าม environment

บทที่ 11, *ทำความเข้าใจ deep agent* พิจารณา deep agent และบทบาทในการเรียกใช้ task แบบ long-horizon บทนี้นิยาม characteristic ของ deep agent แยก model ออกจาก agent harness และวิเคราะห์ Harness Engineering ผ่าน context, action, persistence, execution control, governance และ observability ก่อนเชื่อมไปยัง runtime ที่ประกอบเองและ Claude Managed Agents

บทที่ 12, *Claude Code รุ่นปัจจุบัน: จาก autonomous workflow สู่ production-ready operations* เรียบเรียง capability ที่เพิ่มใน Claude Code ช่วง Week 13 ถึง Week 29 ปี 2026 และแยก patch หลัง Week 29 ถึง v2.1.215 ไว้ต่างหาก บทนี้เพิ่มแผนที่ **Agent Harness → Loop → Graph** เพื่อวินิจฉัย environment, feedback และ flow ก่อนเลือก architecture แล้วครอบคลุม Claude Sonnet 5, Agent view, background และ nested subagent, `/fork`, `/subtask`, `/goal`, Dynamic workflows, Graph Engineering สำหรับออกแบบ node, data edge, fresh-context verifier และ truth anchor, browser, Computer use, screen reader mode, MCP-backed Artifacts, Auto mode, permission rules, code review ตลอดจน diagnostic และ operation command รุ่นปัจจุบัน

เพื่อให้ได้ประโยชน์สูงสุดจากหนังสือเล่มนี้

หนังสือเล่มนี้ตั้งอยู่บนสมมติฐานว่าผู้อ่านมีประสบการณ์ด้าน software development และ concept ของ generative AI มาก่อน ก่อนเริ่มอ่าน คุณควรคุ้นเคยกับการเขียนและแก้จุดบกพร่อง code ด้วย Python หรือ TypeScript การเรียกใช้โปรแกรมจาก terminal และการทำงานกับ codebase หนังสือคาดหวังให้คุณมีความรู้พื้นฐานเกี่ยวกับ Git เช่น การ clone repository และการสร้าง commit การเปลี่ยนแปลง รวมทั้งควรเข้าใจ virtual environment และ environment variable ด้วย

คุณจำเป็นต้องคุ้นเคยกับ **large language model** (**LLM**) และ concept หลักอย่าง agent RAG และ ReAct คุณควรเคยโต้ตอบกับ LLM และสร้าง agent อย่างง่ายมาแล้วอย่างน้อยหนึ่งตัว หนังสือเล่มนี้ไม่ครอบคลุมการเขียนโปรแกรมระดับเริ่มต้นหรือหัวข้อเบื้องต้นเกี่ยวกับ generative AI

หากต้องการทำตามตัวอย่างแบบลงมือปฏิบัติ คุณจำเป็นต้องมี development environment ที่ใช้งานได้ พร้อม Node.js และตั้งค่า Next.js แบบง่ายตามที่แนะนำไว้ในบทแรก ๆ คุณยังต้องเข้าถึง Claude Code พร้อม configuration ด้าน authentication และราคาที่เหมาะสม บางบทจำเป็นต้องติดตั้งและกำหนดค่า GitHub CLI ทำงานกับ GitHub repository และใช้ GitHub Actions ส่วนท้าย ๆ จะเกี่ยวข้องกับการกำหนดค่า MCP server ทั้งแบบ local และ remote รวมถึงการเรียกใช้ Claude Code ภายใน Cursor และ Claude Desktop application

ขอแนะนำให้คุณทำตามตัวอย่างทีละขั้น และทดลองใช้ configuration เมื่อมีการแนะนำในแต่ละช่วง หนังสือสร้างเนื้อหาต่อยอดขึ้นตามลำดับ และหลายบทอาศัย project และการตั้งค่าที่จัดเตรียมไว้ก่อนหน้านี้ รวมถึง HookHub project

ตัวอย่างและ screenshot ในหนังสือเล่มนี้ได้รับการจัดลำดับและเรียบเรียงใหม่เพื่ออธิบาย workflow เชิงเทคนิคอย่างต่อเนื่อง โดยใช้ repository สำหรับการเรียนรู้เป็นกรณีศึกษา

รูปแบบที่ใช้ในหนังสือ

หนังสือเล่มนี้ใช้รูปแบบข้อความหลายประเภท

`CodeInText` : ใช้ระบุคำที่เป็น code ภายในข้อความ ชื่อตาราง database ชื่อ folder ชื่อ file นามสกุล file path URL จำลอง user input และ handle บน X/Twitter ตัวอย่างเช่น: “ทำได้โดยเริ่มต้น file `CLAUDE.md` เพื่อให้เพิ่มเข้าไปใน context ของ project ได้”

code block จะแสดงดังต่อไปนี้:

```
add_context() {
  local context_ref="$1"
  grep -qxF "$context_ref" "$CLAUDE_MD" || echo "$context_ref" >>
    "$CLAUDE_MD"
}
```

input หรือ output ของ command line จะแสดงดังต่อไปนี้:

```
npx create-next-app@latest
```

ตัวหนา: ใช้ระบุคำศัพท์ใหม่ คำสำคัญ หรือคำที่คุณเห็นบนหน้าจอ เช่น คำในเมนูหรือ dialog box จะแสดงในข้อความด้วยรูปแบบนี้ ตัวอย่างเช่น: “ตอนนี้ให้เลือก **Yes (ใช่)** และเลือกไม่ให้ระบบถามอีกใน session นี้”

> คำเตือนหรือหมายเหตุสำคัญจะแสดงในรูปแบบนี้

> เคล็ดลับและเทคนิคจะแสดงในรูปแบบนี้

> **Expert Practice** ใช้ระบุแนวปฏิบัติของผู้เชี่ยวชาญหรือ pattern ที่ผู้เรียบเรียงสังเคราะห์จากหลาย source โดยทุกกล่องจะบอกชื่อผู้เชี่ยวชาญ ขอบเขตของ attribution และแหล่งอ่านเพิ่มเติม

Quick Start: ส่ง change แรกให้ review ได้ใน 60 นาที

ส่วนนี้เป็นทางลัดสำหรับผู้อ่านที่เขียน code ใช้ Git และ terminal ได้อยู่แล้ว คุณไม่ต้องอ่านครบ 12 บทก่อนเริ่ม เป้าหมายของ timebox นี้คือทำ **change ขนาดเล็กหนึ่งรายการ** ใน repository ที่ไม่ใช่ production โดยให้ Claude Code ช่วยสำรวจ วางแผน แก้ code และรวบรวมหลักฐานจนพร้อมให้มนุษย์ review

คำว่า **60 นาที** เป็นกรอบฝึก ไม่ใช่คำรับประกันเวลา เลือกงานที่มี acceptance criteria เดียว กระทบไม่เกิน 1–3 file และมีคำสั่ง test หรือวิธีตรวจ behavior ที่รันได้ในเครื่อง หาก environment ยังติดตั้งไม่เสร็จหรือ task ต้องเปลี่ยน schema, authentication, payment หรือ production infrastructure ให้ใช้เส้นทาง 7 วันแทน

เส้นทาง 60 นาที

นาที 0–5: ตั้ง boundary ก่อนเปิด agent

1. เปิด repository ทดลองหรือสร้าง branch ใหม่จาก working tree ที่สะอาด
2. รันคำสั่ง baseline เช่น test, lint หรือ build อย่างน้อยหนึ่งรายการ แล้วบันทึกผล
3. ห้ามส่ง secret, token, customer data หรือไฟล์ production ให้ session
4. ระบุ operation ที่ห้ามทำ เช่น push, deploy, migration และ destructive Git command

```
git status --short
git branch --show-current
# เลือกคำสั่งที่ repository ใช้จริง เช่น
npm test
```

หาก baseline fail อยู่ก่อน ให้บันทึก failure เดิมไว้ อย่าให้ agent อ้างว่าเป็นผลจาก change ใหม่

นาที 5–15: ให้ context เท่าที่จำเป็น

คัดลอก **Repo Context Template** จาก Implementation Kit แล้วกรอกเฉพาะข้อเท็จจริงที่ตรวจได้: เป้าหมาย repository, directory สำคัญ, command ที่รันได้จริง, coding convention และข้อห้าม หลีกเลี่ยงข้อความกว้างอย่าง “แก้ให้ดีที่สุด”

ขอให้ Claude Code อ่าน `CLAUDE.md`, file ที่เกี่ยวข้อง และ test ใกล้เคียงก่อนเสนอวิธีแก้ โดยยังไม่อนุญาตให้ edit

นาที 15–25: ล็อก Task Contract

คัดลอก **Task Contract Template** แล้วกรอกให้ครบห้าช่อง:

- outcome ที่สังเกตได้หนึ่งข้อ
- non-goals ที่ห้ามขยาย

- file หรือ module ที่อนุญาตให้แตะ
- verification ที่ต้องผ่าน
- authority boundary เช่น “แก้ในเครื่องได้ แต่ห้าม commit, push หรือ deploy”

ให้ Claude สรุป contract กลับมา หากสรุปไม่ตรงให้แก้ contract ก่อนเริ่ม implementation

นาที 25–35: ขอแผนสั้นและตรวจ scope

ขอแผนไม่เกินห้าขั้น แต่ละขั้นต้องบอก file ที่คาดว่าจะแก้และหลักฐานที่จะใช้ตรวจ ตรวจว่าทุกขั้นนำไปสู่ outcome โดยตรง ไม่มี refactor หรือ dependency ใหม่ที่ไม่จำเป็น จากนั้นจึงอนุมัติให้ลงมือทำ

prompt ตั้งต้น:

```
อ่าน CLAUDE.md และ Task Contract ก่อน
สำรวจเฉพาะ file ที่เกี่ยวข้อง แล้วเสนอแผนไม่เกิน 5 ขั้น
แต่ละขั้นระบุ file, change และ verification
ยังไม่แก้ code จนกว่าฉันจะอนุมัติแผน
```

นาที 35–50: ทำ smallest viable change

ให้ agent แก้เฉพาะ scope ที่อนุมัติ ระหว่างทางให้หยุดเมื่อพบ requirement ขัดกัน ต้องแตะ file นอกขอบเขต หรือ baseline ใช้งานไม่ได้ อย่าเพิ่ม agent หลายตัวในรอบแรก เพราะ overhead ด้าน context และ review มักมากกว่าประโยชน์สำหรับงานเล็ก

นาที 50–60: Verify, review และส่งมอบ

1. รัน test/lint/build หรือ behavior check ที่ระบุใน contract
2. ตรวจ `git diff --check`, `git status --short` และ diff จริง
3. เทียบผลกับ baseline แยก failure เดิมออกจาก regression ใหม่
4. บันทึก command, exit status และข้อจำกัดที่ยังไม่ได้ตรวจ
5. ให้ session ใหม่หรือมนุษย์ review diff โดยไม่พึ่งคำสรุปของผู้ลงมือทำเพียงอย่างเดียว

ใช้ **Verification & Rollback Template** เพื่อรวม evidence ไว้ใน artifact เดียว

Definition of Done ของ first change

change พร้อมให้ review เมื่อครบทุกข้อ:

- outcome ใน Task Contract เกิดขึ้นจริง
- ไม่มี file นอก allowed scope เปลี่ยนโดยไม่อธิบาย

- verification command ที่กำหนดผ่าน หรือมีการบันทึก blocker ตามจริง
- diff ถูกตรวจโดยมนุษย์หรือ fresh-context reviewer
- ไม่มี secret, generated artifact หรือ unrelated change ปะปน
- มี rollback path ที่ไม่ต้องเดา
- ยังไม่มีการ push, merge หรือ deploy หากผู้ใช้ไม่ได้ให้อำนาจชัดเจน

เส้นทาง 7 วัน: Agentic Delivery Path

ใช้เส้นทางนี้เมื่อ repository จริงมี dependency, CI หรือ reviewer หลายคน เป้าหมายยังคงเป็น change หนึ่งรายการ ไม่ใช่การอ่านหนังสือให้ครบภายในเจ็ดวัน

วัน	ลงมือทำ	Artifact ที่ต้องได้
1	เลือก change เล็กและบันทึก baseline	`baseline.md`
2	สร้าง Repo Context และตรวจ command ทุกคำสั่ง	`CLAUDE.md` ฉบับสั้น
3	ล็อก outcome, non-goals, scope และ authority	`task-contract.md`
4	สำรวจ dependency และอนุมัติ plan	`plan.md` ไม่เกิน 5 ขั้น
5	Implement ใน branch/worktree แยก	diff ที่จำกัด scope
6	รัน verification และ fresh-context review	`verification.md`
7	แก้ feedback, ทำ rollback drill และสรุป handoff	review-ready change + retro 5 บรรทัด

หากวันใดไม่ผ่าน completion condition ให้แก้ artifact ของวันนั้นก่อน ไม่ต้องเร่งเปิด agent เพิ่มหรือขยาย autonomy

เลือกเส้นทางอ่านตามงาน

- **ต้องส่ง change แรก:** Quick Start → บทที่ 1 → บทที่ 2 → บทที่ 6 → บทที่ 12
- **ต้องเชื่อม tool:** บทที่ 1 → บทที่ 4 → บทที่ 12
- **ต้องสร้าง workflow ใช้ซ้ำ:** บทที่ 3 → บทที่ 7 → บทที่ 9
- **ต้องทำงาน parallel:** บทที่ 6 → บทที่ 7 → บทที่ 10 → บทที่ 12
- **ต้องออกแบบ runtime ระยะยาว:** บทที่ 1 → บทที่ 11 → บทที่ 12

ทุกบทจบด้วยกล่อง **ลงมือทำต่อ** ซึ่งระบุเวลา Artifact และเงื่อนไขว่าเสร็จเมื่อใด คุณสามารถใช้กล่องเหล่านั้นเป็น checkpoint โดยไม่ต้องทำแบบฝึกหัดทุกตัวอย่างในบท

Implementation Kit: Template พร้อมคัดลอก

Kit นี้แปลง concept หลักของหนังสือให้เป็น artifact ที่นำไปใช้กับ repository ของคุณได้ทันที ให้เก็บ file เหล่านี้ใน version control เฉพาะเมื่อไม่มี secret หรือข้อมูลเฉพาะบุคคล และปรับ command ทุกคำสั่งให้ตรงกับ project จริงก่อนให้ agent ใช้

ชุด Artifact พื้นฐาน

1. Repo Context Template

ใช้เป็นโครง `CLAUDE.md` ระดับ project เริ่มต้นให้สั้น แล้วเพิ่มเฉพาะกฎที่มีหลักฐานว่าจำเป็น

```
# Repository Context

## Product
- เป้าหมาย: [ระบบนี้ช่วยใครทำอะไร]
- ขอบเขต repository: [service/application/library]

## Architecture
- entry point: [path]
- module ที่เกี่ยวข้อง: [path + หน้าที่]
- source of truth: [schema/spec/config]

## Commands
- install: `[command ที่ทดสอบแล้ว]`
- test: `[command ที่ทดสอบแล้ว]`
- lint: `[command ที่ทดสอบแล้ว]`
- build: `[command ที่ทดสอบแล้ว]`

## Working rules
- อ่านก่อนแก้: [file/path]
- แก้เฉพาะ scope ที่ task ระบุ
- ห้าม commit, push, deploy หรือแก้ secret หากไม่ได้รับอนุญาต
- หยุดถามเมื่อ requirement ขัดกันหรือจำเป็นต้องแตะ protected path

## Definition of Done
- test ที่เกี่ยวข้องผ่าน
- ไม่มี unrelated diff
- สรุป file ที่เปลี่ยน command ที่รัน และข้อจำกัดที่ยังไม่ได้ตรวจ
```

ตรวจ file นี้เหมือน code: command ต้องรันได้จริง กฎที่หมดอายุต้องลบ และห้ามบันทึก token, credential หรือ customer data

2. Task Contract Template

```
# Task Contract

## Outcome
[พฤติกรรมที่ผู้ใช้หรือ test สังเกตได้หนึ่งข้อ]

## Non-goals
- [สิ่งที่ตั้งใจไม่ทำ]
- [refactor/dependency/migration ที่ไม่อยู่ในรอบนี้]
```

```
## Allowed scope
- อ่านได้: [path]
- แก้ได้: [path]
- ห้ามแตะ: [path หรือระบบ]

## Acceptance criteria
- [ ] [เงื่อนไขเชิง behavior]
- [ ] [เงื่อนไข regression]

## Verification
- baseline: `[command]`
- targeted check: `[command]`
- final check: `[command]`

## Authority
- ทำได้: inspect, plan, edit และ test ใน working tree นี้
- ต้องขออนุมัติก่อน: commit, push, network write, migration, deploy, delete

## Stop conditions
- requirement ขัดกัน
- ต้องแก้ file นอก allowed scope
- test infrastructure ใช้งานไม่ได้
- ล้มเหลวด้วยสาเหตุเดิมสองรอบ
```

3. Verification & Rollback Template

```
# Verification Record

## Baseline
- command: `[command]`
- result/exit status: [ผล]
- known failures: [ไม่มี/รายการ]

## Change evidence
- behavior ที่เปลี่ยน: [ก่อน → หลัง]
- files changed: [รายการ]
- diff reviewed by: [ชื่อ/session]

## Checks
| Check | Command/วิธีตรวจ | ผล | Evidence |
|---|---|---|---|
| targeted test | `[command]` | PASS/FAIL | [log/screenshot] |
| regression | `[command]` | PASS/FAIL | [log] |
| diff hygiene | `git diff --check` | PASS/FAIL | [ผล] |

## Not verified
- [platform, browser, integration หรือ edge case ที่ยังไม่ได้ตรวจ]

## Rollback
- หยุดใช้ change เมื่อ: [trigger]
- วิธีคืนค่า: [revert commit/disable flag/restore config]
- ผู้มีอำนาจตัดสินใจ: [owner]
```

Rollback ต้องอ้างอิง state ที่รู้จัก หลีกเลี่ยงการใช้ `git reset --hard` เป็นคำแนะนำทั่วไป เพราะอาจลบงานที่ยังไม่ได้ commit ให้ revert commit หรือคืน file ที่ระบุอย่างชัดเจนหลังตรวจ `git status` ก่อน

4. Subagent Brief Template

ใช้ brief นี้เมื่อ task ใหญ่พอที่จะได้ประโยชน์จาก fresh context ระบุ ownership และ output ให้จบในตัว ห้ามสมมติว่า subagent เห็นบทสนทนาหลักทั้งหมด

```
---
name: focused-reviewer
description: ตรวจ diff กับ Task Contract และส่งเฉพาะข้อค้นพบที่มีหลักฐาน
tools: Read, Glob, Grep
---

# Mission
ตรวจ [diff/path] เทียบกับ [task-contract.md]

# Context to read
- [CLAUDE.md]
- [task contract]
- [changed files]
- [tests ที่เกี่ยวข้อง]

# Ownership
- อ่านได้: [path]
- ห้ามแก้ file, commit, push หรือเรียก network write

# Required output
1. findings เรียงตามความรุนแรง พร้อม file/line/evidence
2. acceptance criteria ที่ตรวจแล้ว
3. สิ่งที่ตรวจไม่ได้และเหตุผล
4. verdict: ready / needs changes / blocked

# Stop condition
หยุดและรายงาน blocker หาก context ไม่พอหรือ contract ขัดกัน ห้ามเดา
```

5. CI Recovery Loop Template

```
# CI Recovery Loop
## Trigger
CI run ของ [branch/PR] จบด้วยสถานะ failed
## Read-only diagnosis
1. อ่าน failed job, exact command และ log ช่วงแรกที่เกิด error
2. แยก infrastructure/flaky failure ออกจาก regression ของ change
3. เทียบกับ baseline และ commit ล่าสุดที่ผ่าน
## Attempt contract
- แก้ได้เฉพาะ: [paths]
- retry ได้สูงสุด: 2 รอบ
- ห้ามเปลี่ยน test ให้ผ่านด้วยการ skip, ลด assertion หรือปิด quality gate
- ห้าม push/re-run billable workflow หากไม่ได้รับอนุญาต
## Verification
- รัน command ที่ fail ซ้ำใน environment ที่เหมาะสม
- รัน targeted test และ regression check
- ตรวจ diff กับ Task Contract
## Stop and escalate when
- failure เดิมเกิดซ้ำสองรอบ
- ต้องเปลี่ยน secret, permission, runner หรือ production service
- root cause อยู่นอก allowed scope
## Completion output
- root cause, evidence และ change ที่ทำ
- command, ผล, remaining risk และ recommended next action
```

Loop ที่ดีต้องมี Trigger, Verification, Stop Condition และ authority boundary ครบ หากขาดข้อใดข้อหนึ่งให้ใช้ human-guided workflow แทน automation

ใช้ Kit ให้พร้อมใน 15 นาที

1. เลือกเฉพาะ template ที่ตรงกับงานรอบนี้ ไม่ต้องคัดลอกทั้งชุด
2. แทนที่ข้อความในวงเล็บเหลี่ยมทุกจุดด้วย path, command และ owner ที่ตรวจสอบแล้ว
3. รัน baseline ก่อนแก้ เพื่อแยกปัญหาเดิมออกจาก regression ของ change
4. อ่าน authority boundary ซ้ำ แล้วตัดสิทธิ์ที่ task นี้ไม่จำเป็นต้องใช้
5. บันทึก Artifact ไว้ใน repository เฉพาะเมื่อไม่มี secret หรือข้อมูลส่วนบุคคล

Definition of Ready: ไม่มี placeholder ค้างอยู่, command หลักรันได้จริง, allowed scope ชัดเจน และมีคนหรือเงื่อนไขที่ตัดสินใจหยุดงานได้

FREE PREVIEW · ลงมือกับรากฐาน

บทที่ 1 — Context Engineering

จัด context ให้ Claude Code เห็นข้อมูลที่จำเป็น ถูกลำดับ และพร้อมใช้กับงานจริง โดยไม่ปล่อยให้ conversation เติบโตอย่างไร้ขอบเขต

- แยก context ที่ต้องจำออกจากข้อมูลเฉพาะ task
- ลด context pollution และเลือกข้อมูลเท่าที่จำเป็น
- ออกแบบ system prompt และ memory ให้ตรวจสอบได้

ตัวอย่างนี้คัดบางส่วนจากฉบับเต็มเพื่อให้เห็นแนวทางและรูปแบบจริง

ส่วนที่ 1

รากฐานของ Context Engineering และ Claude Code

ใน *ส่วนที่ 1* ของหนังสือเล่มนี้ คุณจะสร้างความเข้าใจที่ชัดเจนและนำไปใช้ได้จริงเกี่ยวกับ Context Engineering รวมถึงเหตุผลที่เรื่องนี้มีความสำคัญเมื่อทำงานกับ AI coding agent สมัยใหม่ เราจะพิจารณาว่าการเปลี่ยนผ่านจาก prompt engineering แบบเรียบง่ายไปสู่การจัดการ context อย่างมีโครงสร้าง ส่งผลต่อวิธีออกแบบและใช้งาน agent อย่างไร คุณจะได้สำรวจว่า context ถูกสร้างขึ้นอย่างไร เติบโตขึ้นตามเวลาอย่างไร และการจัดการที่ไม่มีประสิทธิภาพส่งผลต่อ performance ต้นทุน และ reliability อย่างไร

ควบคู่ไปกับรากฐานเชิง concept คุณจะเริ่มลงมือทำงานกับ Claude Code โดยตรง ผ่านตัวอย่างที่มีคำแนะนำเป็นลำดับ คุณจะได้เห็นว่า Claude Code ตีความ codebase สร้าง working context และนำ context นั้นไปใช้กับงาน development จริงอย่างไร เมื่อจบส่วนนี้ คุณจะมีความเข้าใจที่มั่นคงทั้งด้านทฤษฎีและการใช้งาน Claude Code ในแต่ละวัน

ส่วนนี้ของหนังสือประกอบด้วยบทต่อไปนี้:

- *บทที่ 1, Context Engineering*
- *บทที่ 2, แก่นสำคัญของ Claude Code*
- *บทที่ 3, เริ่มต้นใช้งาน Claude Code – สำรวจ command สำคัญ*

1

Context Engineering

บทนี้แนะนำ Context Engineering และอธิบายว่าเหตุใด concept นี้จึงมีความสำคัญอย่างยิ่งต่อการสร้างและใช้งาน AI agent สมัยใหม่ แม้ระบบ AI จำนวนมากมักถูกอธิบายว่าเป็น “เพียง prompt ที่ห่อหุ้ม LLM” แต่บทนี้จะแสดงให้เห็นว่าเหตุใดมุมมองดังกล่าวจึงใช้ไม่ได้เมื่อ agent ซับซ้อนขึ้น ทำงานยาวนานขึ้น และขับเคลื่อนด้วย tool คุณจะได้เรียนรู้ว่า context มาจากที่ใด เหตุใดจึงเพิ่มขึ้นอย่างต่อเนื่อง และการจัดการ context ที่ไม่ดีนำไปสู่ performance ที่ลดลง ค่าใช้จ่ายสูงขึ้น hallucination และพฤติกรรมที่ไม่สม่ำเสมอได้อย่างไร

เป้าหมายของบทนี้คือมอบ mental model ที่ชัดเจนเกี่ยวกับ Context Engineering และแสดงวิธีนำไปใช้ในทางปฏิบัติ เราจะเริ่มด้วยการอธิบายว่า Context Engineering วิวัฒน์มาจาก prompt engineering อย่างไร จากนั้นใช้ Claude Code เป็นตัวอย่างที่เป็นรูปธรรมว่า agent สมัยใหม่เขียน เลือก บีบอัด และแยก context อย่างไร ในช่วงท้าย เราจะพิจารณา system prompt เหตุใด system prompt ยังคงสำคัญ และวิธีออกแบบอย่างมีประสิทธิผล

บทนี้จะครอบคลุมหัวข้อต่อไปนี้:

- บทนำสู่ Context Engineering
- Claude Code และปรัชญาด้าน Context Engineering
- system prompt และเหตุผลที่สำคัญ

> การอภิปรายนี้อ้างอิงข้อมูลที่เปิดเผยต่อสาธารณะ รวมถึง blog post ด้าน engineering ของ Anthropic และการวิเคราะห์จาก community เนื้อหานี้ไม่ได้เป็นถ้อยแถลงอย่างเป็นทางการจาก Anthropic

บทนำสู่ Context Engineering

ในส่วนนี้ เราจะเจาะลึกว่า Context Engineering คืออะไร หากคุณเคยทำงานกับ AI agent และอาจรวมถึง coding agent เช่น Cursor และ Claude Code หรือบางทีคุณอาจเคยพัฒนา AI agent สำหรับบริษัทหรือใช้งานเอง คุณน่าจะทราบว่าโดยพื้นฐานแล้ว ทุกอย่างล้วนลงเอยที่การส่ง prompt ไปยัง LLM พร้อมกับงานด้าน engineering จำนวนมากที่อยู่รอบ prompt นั้น

คำกล่าวที่ว่า application อย่าง Cursor และ Claude Code เป็นเพียงตัวห่อหุ้ม LLMs นั้นมีส่วนจริงอยู่บ้าง อย่างไรก็ตาม การสร้างตัวห่อหุ้มที่ดีมากต้องอาศัยทั้งความรู้เชิงลึกและงาน engineering จำนวนมาก อีกคำหนึ่งที่มักใช้เรียกระบบแวดล้อมนี้คือ *Agentic harness* ซึ่งเป็น orchestration layer รอบ LLM โดยรับผิดชอบการจัดการ tool call ควบคุม agentic loop จัดการ error บังคับใช้ guardrail และที่สำคัญที่สุดคือตัดสินใจว่าจะส่ง context ใดไปยัง model ในแต่ละขั้นตอน

ในทางปฏิบัติ งาน engineering จริงส่วนใหญ่ไม่ได้อยู่ภายในตัว LLM call แต่อยู่รอบ ๆ การเรียก model มักไม่ซับซ้อน สิ่งที่กำหนดว่า agent จะทำงานได้อย่างน่าเชื่อถือหรือไม่ คือวิธีที่ระบบแวดล้อมจัดการ state, tool, memory และ context เพราะการเรียก LLMs มาพร้อม context เสมอ และ context นี้มาจากแหล่งกับ process ต่อเนื่องที่หลากหลาย

ตัวอย่างเช่น context อาจมาจากหลายแหล่ง:

- อาจมาจาก application developer
- อาจมาจาก user
- อาจมาจาก interaction ก่อนหน้าของ user tool call หรือ data ภายนอกอื่น ๆ

ทุกวันมีการเพิ่มแหล่ง context ใหม่ และปริมาณ context ก็เพิ่มขึ้นอย่างต่อเนื่อง การส่ง context ที่ถูกต้องและเกี่ยวข้องไปยัง LLM ไม่ได้เรียบง่ายอย่างที่เราเคยคิดในยุคแรก

จาก prompt engineering สู่ Context Engineering

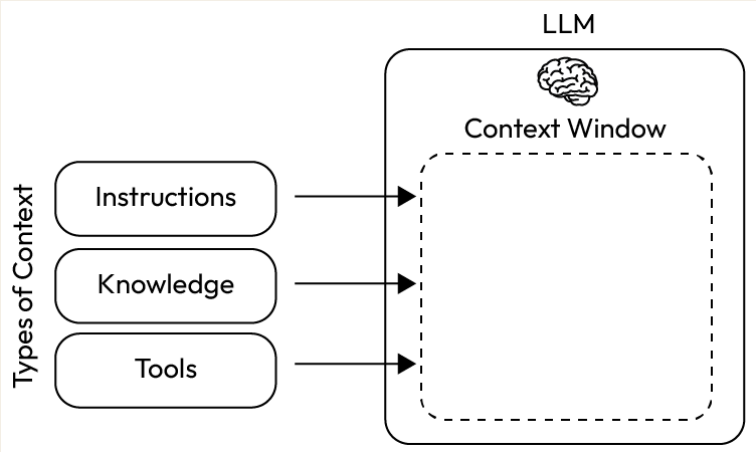
ในยุคแรก เราเชื่อว่า prompt engineering ก็เพียงพอแล้ว เราคิดว่าการเขียน prompt สวยหรูจะช่วยแก้ปัญหาและให้สิ่งที่เราต้องการได้ แต่ปัญหาคือ prompt เป็นแบบ static ขณะที่ context มีความ dynamic อย่างยิ่ง

หาก context เป็นแบบ dynamic การสร้าง context ที่ถูกต้องก็ต้องอาศัยระบบ dynamic เช่นกัน เรื่องนี้ไม่ได้มีเพียงการเขียน static prompt อีกต่อไป นี่คือเหตุผลที่เรากำลังก้าวเข้าสู่อาณาจักรของ **Context Engineering** ซึ่งเป็นวิวัฒนาการตามธรรมชาติของ prompt engineering แต่เป็น concept ที่ลึกซึ้งกว่ามาก

เหตุผลที่ context สำคัญต่อ agent

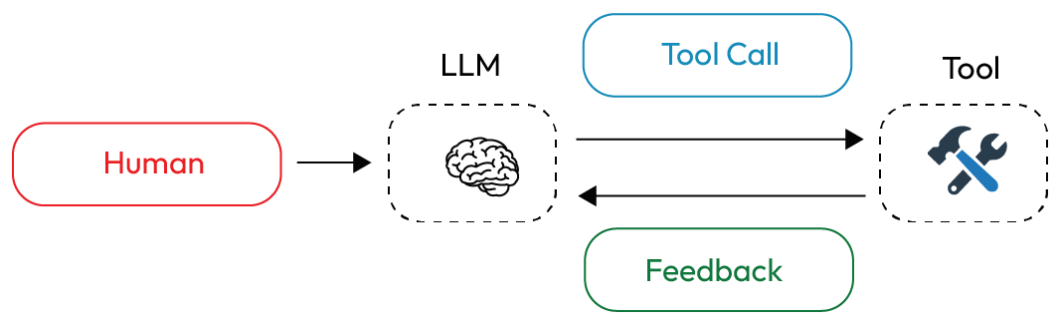
เราทุกคนรู้จักคำกล่าวที่ว่า *“ใส่ขยะเข้าไป ก็ได้ขยะออกมา”* นี่เป็นหนึ่งในสาเหตุที่พบบ่อยที่สุดซึ่งทำให้ระบบ agentic ทำงานไม่ได้ตามที่ควร เพราะระบบไม่ได้รับ context ที่ถูกต้อง

LLMs ไม่สามารถอ่านใจเราได้ เราต้องให้ข้อมูลที่ถูกต้อง และที่จริงแล้ว สิ่งที่ต้องให้ไม่ใช่เพียงข้อมูลหรือ data เสมอไป บางครั้งเราต้องมอบ tool ที่ถูกต้อง เพื่อให้ model ดึงข้อมูลอื่น ดำเนินการ และทำงานแทนเราได้



รูปที่ 1.1 – context window ของ LLM (ดัดแปลงจาก concept ที่ LangChain กล่าวถึง, www.langchain.com)

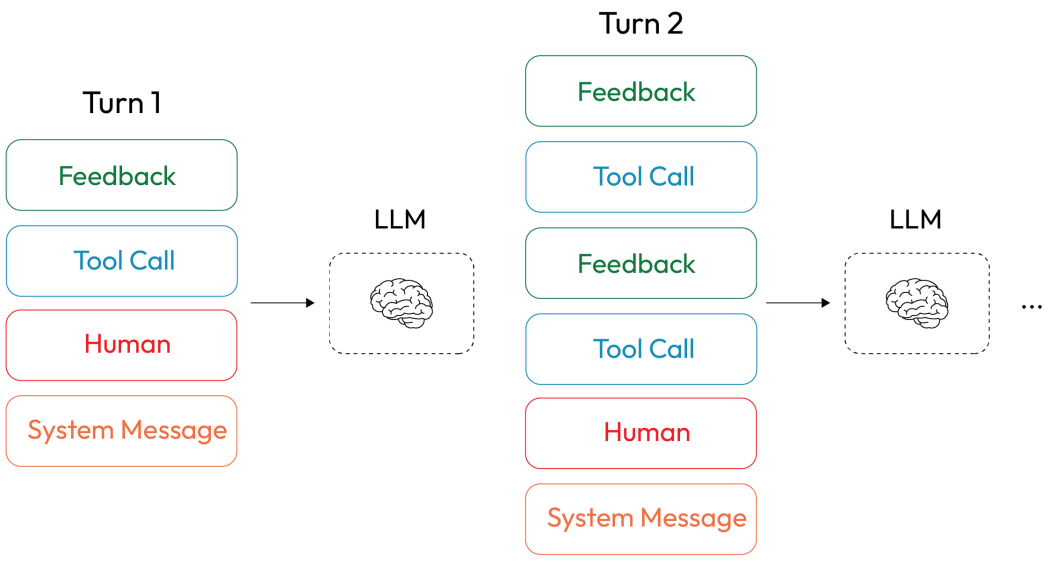
LLMs มีความสามารถใน reasoning ดีขึ้นเรื่อย ๆ เรามี tool calling และสามารถสร้าง AI agent ที่เรียกใช้ tool รับ output และวน loop จนกว่างานจะเสร็จสมบูรณ์



รูปที่ 1.2 – tool-calling loop ของ LLM (ดัดแปลงจาก concept ที่ LangChain กล่าวถึง, www.langchain.com)

อย่างไรก็ตาม เมื่อเป็นงานซับซ้อนและทำงานยาวนาน เรามักสะสม feedback จาก tool call

สิ่งนี้ทำให้ context window ขยายขึ้นอย่างต่อเนื่อง และเต็มไปด้วย output จาก tool call กับ output ระหว่างทาง



รูปที่ 1.3 – การเติบโตของ context ในหลาย turn (ดัดแปลงจาก concept ที่ LangChain กล่าวถึง, www.langchain.com)

สิ่งนี้นำไปสู่ปัญหาหลายประการ:

- context window อาจเกินขีดจำกัดขนาด
- ค่าใช้จ่ายและ latency เพิ่มขึ้น
- performance ของ agent เริ่มลดลง

หากไม่ดำเนินการใด ๆ ความเสื่อมถอยนี้จะหลีกเลี่ยงไม่ได้ การอภิปรายล่าสุดได้สำรวจว่าเหตุใด context ยาวจึงล้มเหลวในทางปฏิบัติ และความเสื่อมถอยค่อย ๆ ปรากฏขึ้นอย่างไรเมื่อข้อมูลที่ไม่เกี่ยวข้องหรือขัดแย้งกันสะสมเพิ่มขึ้น หากต้องการอ่านการอภิปรายเชิงลึก โปรดดู blog post ยอดเยี่ยมชื่อ *“เหตุใด context ยาว จึง ล้มเหลว”* โดย Dan Breunig

(https://www.dbreunig.com/2025/06/22/how-contexts-fail-and-how-to-fix-them.html) หากปล่อยให้ context เติบโตโดยปราศจากโครงสร้าง การคัดเลือก หรือการควบคุม ปัญหาหลายอย่างจะเริ่มปรากฏในระบบ agentic เช่นรายการต่อไปนี้:

- **context poisoning**: เมื่อ hallucination จาก tool call หรือ LLM call เข้าสู่ context และเริ่มส่งผลต่อ output ในอนาคต
- **context confusion**: เมื่อ context ที่ไม่จำเป็นมีอิทธิพลต่อคำตอบ แม้จะไม่เกี่ยวข้องกับงาน
- **context clash**: เมื่อส่วนต่าง ๆ ของ context ขัดแย้งกัน

ในส่วนถัดไป เราจะอภิปรายเทคนิคสำหรับทำ Context Engineering ให้ดีขึ้น เทคนิคบางส่วนนำไปใช้โดย application developer เช่น ใน tool อย่าง Claude Code ส่วนเทคนิคอื่นอยู่ฝั่ง user

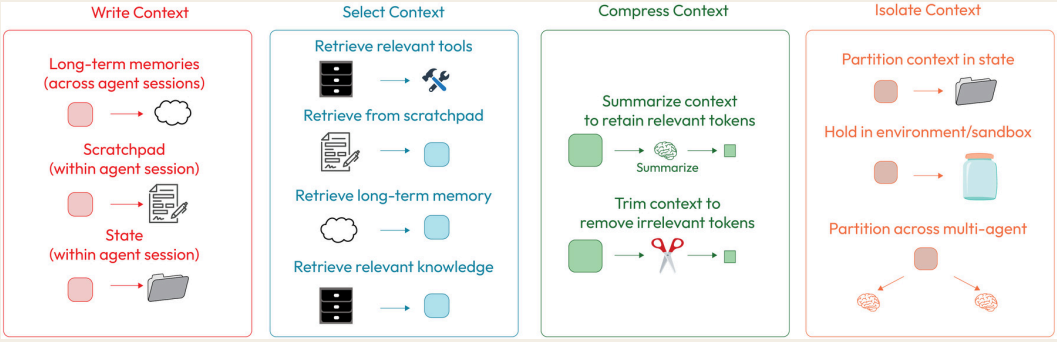
ในฐานะ user Claude Code เรามีอิทธิพลอย่างมากต่อ context ที่จะถูกส่งไปยัง LLM ในท้ายที่สุด รวมถึงคำตอบที่เราได้รับ ซึ่งหมายความว่าแม้แต่ผู้ที่ไม่ใช่ developer ก็ต้องเข้าใจ Context Engineering หากต้องการคำตอบที่ดีขึ้นจากระบบ AI และ AI agent

coding agent อย่าง Claude Code เป็นตัวอย่างที่ยอดเยี่ยมว่าเทคนิค Context Engineering ถูกนำไปใช้อย่างไรทั้งฝั่ง developer และฝั่ง user นี่คือสิ่งที่เราจะสำรวจในส่วนถัดไป รวมถึงวิธีออกแบบ context ของเราให้ดียิ่งขึ้น

Claude Code และกลยุทธ์ด้าน Context Engineering

ในส่วนนี้ เราจะกล่าวถึง Claude Code ปรัชญาด้าน Context Engineering และวิธีที่ Claude Code รับมือความท้าทายเหล่านี้ Claude Code นำกลยุทธ์ด้าน Context Engineering มาใช้ด้วยการนำไปใช้ทั้งสี่แนวทางต่อไปนี้:

- **การเขียน context**: ระบบ memory แบบคงอยู่
- **การเลือก context**: context retrieval และการแทรกอย่างชาญฉลาด
- **Context Compression**: การนำเสนอและสรุป context อย่างมีประสิทธิภาพ
- **การแยก context**: การจัดการ context แบบ multi-agent และแบบกำหนด scope



รูปที่ 1.4 – หมวดหมู่ของ Context Engineering (ดัดแปลงจาก concept ที่ LangChain กล่าวถึง, www.langchain.com)

มาเริ่มจากแนวทางแรกกัน

กลยุทธ์ที่ 1: การเขียน context และ memory แบบคงอยู่

กลยุทธ์แรกคือ **การเขียน context และ architecture memory แบบคงอยู่** Claude มีระบบ memory หลาย layer และ Claude Code นำ memory หลาย scope มาประกอบกัน เพื่อคง context ข้าม session ระหว่างเขียน code ด้วย Claude Code

- อันดับแรก เรามี **memory ของ project** (`./CLAUDE.md`) นี่คือ context ที่แชร์กันในทีมสำหรับ architecture มาตรฐาน หรือสิ่งใดก็ตามที่เกี่ยวข้องกับ project หนึ่ง ๆ memory ประเภทนี้ควบคุมด้วย version และสมาชิกทุกคนในทีมเข้าถึงได้

```
# context ของ project

## ภาพรวม architecture
นี่คือ application microservice ที่ใช้:
- Node.js พร้อม Express สำหรับบริการ API
- React พร้อม TypeScript สำหรับ frontend
- PostgreSQL พร้อม Prisma ORM
- Redis สำหรับ caching

## มาตรฐานการเขียน code
- ใช้ functional component พร้อม hook
- นำไปใช้ error boundary สำหรับ route component ทั้งหมด
- ปฏิบัติตามข้อตกลง RESTful API
- เขียน unit test สำหรับ business logic ทั้งหมด
```

- จากนั้นเรามี **memory ของ user** (`~/.claude/CLAUDE.md`) ซึ่งจัดเก็บไว้ใน home directory ของ user ภายใน directory `claude` ใน file `CLAUDE.md` โดยมีค่ากำหนดและ shortcut ส่วนบุคคลสำหรับทุก project ของ user นั้น ข้อมูลนี้จะไม่ถูก commit ไปยัง GitHub และเป็นข้อมูลเฉพาะ user แต่ละคนจะมีค่าต่างกัน และข้อมูลจะคงอยู่ในทุก Claude Code session

```
# ค่ากำหนดส่วนบุคคลสำหรับ development

## code style
- ใช้ return type ที่ชัดเจนใน TypeScript เสมอ
- เลือกใช้ const assertion แทน type annotation
- ใช้ชื่อตัวแปรที่สื่อความหมาย และหลีกเลี่ยงตัวย่อ

## shortcut workflow
- เมื่อเขียน test ให้ใช้ Jest ร่วมกับ React Testing Library
- เรียกใช้ `npm run lint` ก่อน commit เสมอ
- เลือกใช้ composition แทน inheritance
```

FREE PREVIEW · เชื่อม TOOL อย่างมีขอบเขต

บทที่ 4 — ขยาย Claude Code ด้วย MCP

เชื่อม tool และ data ภายนอกผ่าน MCP โดยกำหนด scope, permission และ context cost ก่อนนำ integration ไปใช้กับ workflow จริง

- เข้าใจ host, client และ server boundary
- เลือก scope ของ configuration ให้เหมาะกับงาน
- ตรวจ context cost, credential และผลลัพธ์ก่อนเชื่อถือ

ตัวอย่างนี้คัดบางส่วนจากฉบับเต็มเพื่อให้เห็นแนวทางและรูปแบบจริง

4

การขยายความสามารถของ Claude Code ด้วย MCP server และ plugin

Model Context Protocol (**MCP**) มอบวิธีมาตรฐานให้ AI application เชื่อมต่อกับ tool แหล่ง data และบริการต่าง ๆ ผ่าน interface ร่วม แทนที่จะสร้าง integration แยกสำหรับ application host แต่ละตัว developer สามารถนำไปใช้ functionality เพียงครั้งเดียวใน MCP server แล้วนำกลับมาใช้กับ client ใด ๆ ที่รองรับ MCP ได้ MCP ถูกนำเสนอขึ้นเพื่อแก้โจทย์ด้าน integration นี้ abstraction layer ดังกล่าวช่วยให้ระบบต่าง ๆ ทำงานร่วมกันได้ มี portability และเอื้อต่อการเติบโตของ ecosystem ภายใน context ของ Claude Code นั้น MCP มีความสำคัญเป็นพิเศษ เพราะช่วยให้ developer ขยายความสามารถของ coding agent ด้วย tool ความสามารถ และ integration เพิ่มเติมได้อย่างเป็นระบบและนำกลับมาใช้ใหม่ได้

ในบทนี้ คุณจะได้เรียนรู้ว่า MCP คืออะไร เหตุใดจึงถูกสร้างขึ้น และทำงานอย่างไรในทางปฏิบัติ คุณจะได้สำรวจปัญหาด้าน integration ที่ MCP เข้ามาแก้ไข และพิจารณา component หลักทาง architecture ได้แก่ host client และ server ผ่านการสาธิตเชิงปฏิบัติ คุณจะได้เชื่อมต่อ MCP server เข้ากับ AI application เช่น Claude และ Cursor กำหนดค่าทั้ง MCP server แบบ local และ remote ตลอดจนทำความเข้าใจว่า tool ถูกเปิดให้ใช้งานและเรียกผ่าน protocol อย่างไร

คุณจะได้พิจารณาประเด็นสำคัญด้านการปฏิบัติงานด้วย แม้ MCP จะทรงพลัง แต่ก็นำมาซึ่งความท้าทายเกี่ยวกับการจัดการ context ประสิทธิภาพ และประสิทธิผลในการทำงาน บทนี้สำรวจกลยุทธ์ด้าน Context Engineering เพื่อลด context pollution กำหนด scope ของ MCP configuration ให้เหมาะสม และจัดการ server แบบ dynamic ภายใน session นอกจากนี้ คุณจะได้เรียนรู้ว่า Claude Code plugin รวม MCP server และ component ที่เกี่ยวข้องไว้เป็นหน่วยที่นำกลับมาใช้และแชร์ต่อได้อย่างไร ซึ่งช่วยสร้างมาตรฐานร่วมกันทั้งทีม

เป้าหมายโดยรวมของบทนี้คือช่วยให้คุณสร้างพื้นฐานเชิง concept และเชิงปฏิบัติที่แข็งแกร่งเกี่ยวกับ MCP เมื่ออ่านจบ คุณจะไม่เพียงเข้าใจว่า MCP ช่วยให้ผสานรวม tool ข้าม platform ได้อย่างไร แต่ยังเข้าใจวิธีกำหนดค่าและจัดการ MCP อย่างรับผิดชอบในระบบ AI ที่ใช้งานจริงด้วย

ความรู้นี้สำคัญเพราะ MCP กำลังกลายเป็น layer พื้นฐานใน architecture ที่ใช้ agent อย่างรวดเร็ว เมื่อ AI development เปลี่ยนไปสู่ระบบที่เสริมความสามารถด้วย tool และ automation ความเข้าใจเกี่ยวกับ architecture ประโยชน์ และข้อแลกเปลี่ยนของ MCP จะช่วยให้คุณออกแบบ AI application ที่ขยายขนาดได้ มีประสิทธิภาพ และทำงานร่วมกับระบบอื่นได้

ในบทนี้ เราจะครอบคลุมหัวข้อต่อไปนี้:

- ทำไมต้อง MCP?
- architecture หลักของ MCP
- MCP ในการใช้งานจริง

- การเชื่อมต่อ Claude Code กับ MCP server
- ชีตสรุป scope ของ MCP
- Context Engineering ใน MCP ecosystem
- MCP configuration ระดับ project
- ข้อจำกัดในปัจจุบันของ MCP
- การจัดการ configuration ด้วย Claude Code plugin

> แม้หนังสือเล่มนี้จะเน้น Claude Code เป็นหลัก แต่ MCP เป็น protocol ข้าม platform ที่ tool พัฒนา AI หลายตัวรองรับ เพื่อจุดประสงค์ในการสาธิต บทนี้จึงใช้ Cursor ในบางตัวอย่าง concept พื้นฐานยังคงเหมือนเดิมและสามารถนำไปใช้เมื่อใช้งาน Claude Code ได้

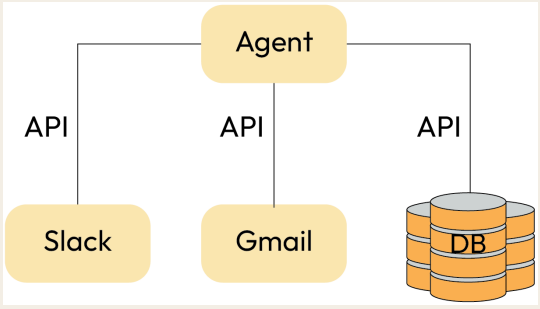
ทำไมต้อง MCP?

MCP ได้รับการยอมรับอย่างรวดเร็ว มีการพูดถึงอย่างแพร่หลายและมี implementation MCP server จำนวนมาก AI application หลายตัว รวมถึง tool อย่าง Claude Code และ Cursor ต่างผสานรวม MCP ไว้ภายในระบบของตน

ก่อนจะนิยาม MCP อย่างละเอียด สิ่งสำคัญคือต้องเข้าใจความจำเป็นที่ MCP เข้ามาตอบโจทย์

ปัญหาด้าน integration

ลองพิจารณา AI agent ที่ออกแบบมาให้ดำเนินการต่าง ๆ เช่น ส่งข้อความ Slack อ่านและส่งอีเมล หรือสืบค้น database เพื่อเปิดใช้ความสามารถเหล่านี้ ต้องนำไปใช้ functionality ที่จำเป็นโดยตรง ซึ่งเกี่ยวข้องกับการทำงานกับ API เช่น Slack หรือ Gmail และการเขียน code เฉพาะเพื่อห่อหุ้ม integration เหล่านี้เป็น tool ที่ agent เข้าถึงได้



รูปที่ 4.1 – agent ที่ผสานรวมกับ Slack, Gmail และ database ผ่าน API

ในหลายกรณี จำเป็นต้องมี implementation แบบเฉพาะ ตัวอย่างเช่น integration อีเมลอาจตั้งใจไม่ให้เข้าถึง endpoint สำหรับลบใน Gmail API เพื่อป้องกันการสูญหายของ data โดยไม่ตั้งใจ ด้วยเหตุนี้ logic ของ integration จึงถูกปรับให้ตรงกับ requirement เฉพาะและนำไปใช้ด้วยตนเอง

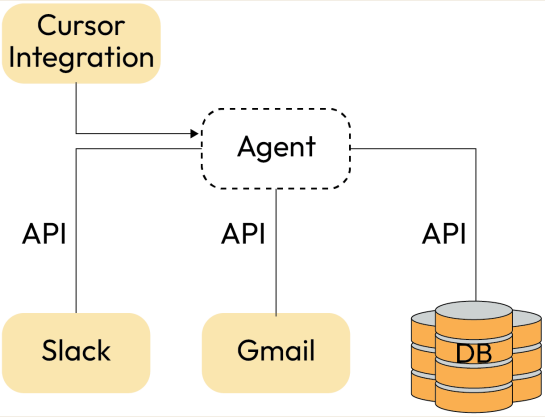
framework อาจมี tool ในตัว ตัวอย่างเช่น framework หนึ่งอาจมี integration Gmail ที่นำไปใช้ไว้ล่วงหน้าและครอบคลุมพื้นผิว API ทั้งหมด tool ดังกล่าวสามารถใช้งานได้ทันทีสำหรับความต้องการทั่วไป อย่างไรก็ตาม ไม่ว่าจะใช้ tool ในตัวหรือ tool เฉพาะ ก็ยังต้องผสานรวม functionality นั้นเข้ากับ environment ของ agent ที่ต้องการ

เมื่อนำไปใช้แล้ว agent จะสามารถส่งอีเมล โพสต์ข้อความ Slack หรือเรียกใช้ query database ได้ integration นี้ทำงานภายใน agent ตัวนั้นโดยเฉพาะ

ความท้าทายด้าน portability

สมมติว่า agent ได้รับการใช้งานอย่างแพร่หลาย และทีมอื่น ๆ ต้องการใช้ functionality ของ agent ใน environment ที่ต่างออกไป

ตัวอย่างเช่น ลองพิจารณา agent ที่เดิมสร้างมาให้ทำงานภายใน environment Cursor ในตั้งค่านี้ agent ผสานรวมกับบริการภายนอก เช่น Slack, Gmail และ database ผ่าน API ของแต่ละบริการ diagram ต่อไปนี้แสดง architecture ดังกล่าว

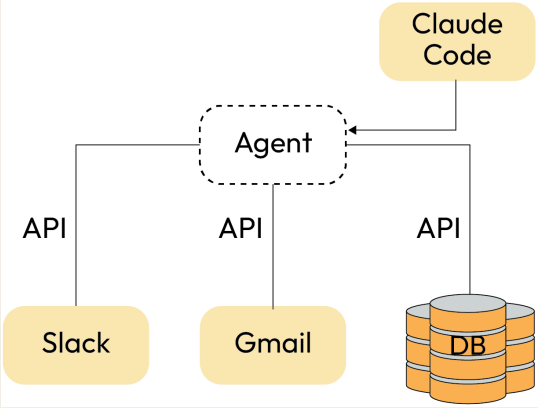


รูปที่ 4.2 – agent ที่ผูกติดกับ environment Cursor และ integration API ของ environment นั้น

ในกรณีนี้ agent ผูกติดอย่างแน่นหนากับ environment Cursor integration กับ Slack, Gmail และ database ถูกนำไปใช้ขึ้นสำหรับตั้งค่าดังกล่าวโดยเฉพาะ

ทีนี้สมมติว่าเราต้องการผสานรวมความสามารถของ agent ชุดเดียวกันเข้ากับ Claude Code แม้ functionality จะเหมือนกันในเชิง concept แต่ไม่สามารถนำ integration กลับมาใช้ซ้ำได้โดยตรง เนื่องจาก implementation ถูกเขียนขึ้นสำหรับ agent ของเราโดยเฉพาะ จึงต้องสร้าง integration สำหรับ Claude Code ขึ้นใหม่

ด้วยเหตุนี้ ความสามารถที่คล้ายกัน เช่น การส่งข้อความ Slack interaction กับ Gmail หรือการเรียกใช้ query database จึงต้องนำไปใช้ใหม่สำหรับ environment ใหม่ ลองนึกต่อไปว่าเราต้องการผสานรวมความสามารถชุดเดียวกันเข้ากับอีก environment หนึ่ง จะเป็นอย่างไรหากเราต้องการผสานรวม agent ตัวเดียวกันเข้ากับ Cursor?



รูปที่ 4.3 – การนำไปใช้ integration ของ agent ใหม่สำหรับ environment Claude Code

การขยายการรองรับไปยัง environment อื่น เช่น Cursor, Bolt, GitHub Copilot หรือผู้ช่วยเขียน code AI ตัวอื่น จำเป็นต้องทำ process นี้ซ้ำ integration ใหม่แต่ละรายการทำให้มีงาน implementation เพิ่มขึ้น ต้องปรับ functionality เดียวกันแยกกันสำหรับทุกระบบเป้าหมาย

MCP ในฐานะ abstraction layer

ความท้าทายด้าน portability ก่อให้เกิดปัญหา integration ที่เกิดขึ้นซ้ำ ๆ ทุกครั้งที่ต้องใช้ functionality เดียวกันใน environment ที่ต่างออกไป developer ต้องสร้าง integration ใหม่ตั้งแต่ต้น

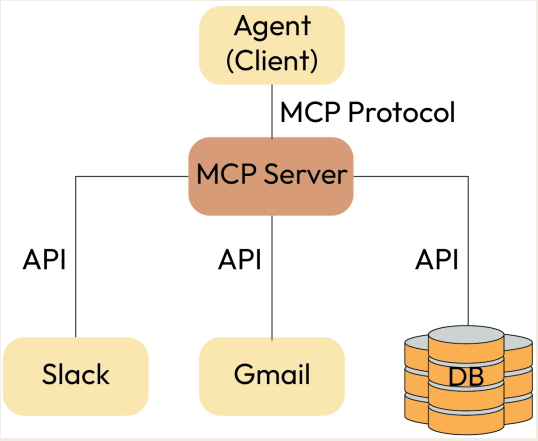
MCP แก้ปัญหานี้ด้วยการเพิ่ม abstraction layer อีกหนึ่งชั้น หลักการที่รู้จักกันดีใน software engineering กล่าวไว้ว่า:

“เราสามารถแก้ปัญหาใด ๆ ได้ด้วยการเพิ่มระดับของการอ้างอ้อมเข้าไปอีกหนึ่งระดับ”

— Andrew Koenig

concept นี้เชื่อมโยงอย่างใกล้ชิดกับทฤษฎีบทพื้นฐานของ software engineering ซึ่งระบุว่าปัญหาซับซ้อนจำนวนมากในระบบสามารถทำให้ง่ายลงได้ด้วยการเพิ่ม abstraction layer MCP server ทำหน้าที่นี้

แทนที่จะผสานรวม functionality แยกกันใน agent แต่ละตัว เราจะผสานรวม implementation เพียงครั้งเดียวใน MCP server จากนั้น agent ที่รองรับ MCP ก็สามารถเชื่อมต่อกับ MCP server นั้นได้โดยตรง



รูปที่ 4.4 – MCP server ที่ทำหน้าที่เป็น integration layer แบบรวมศูนย์สำหรับบริการภายนอก

แนวทางนี้ทำให้นำไปใช้ความเข้ากันได้เพียงครั้งเดียว เมื่อเปิดความสามารถผ่าน MCP server แล้ว ก็สามารถใช้ความสามารถเหล่านั้นกับ platform ใด ๆ ที่รองรับ protocol ได้ ความสามารถที่เดิมนำไปใช้สำหรับ Cursor จะเข้ากันได้กับ Claude Code และ environment อื่นที่รองรับ MCP โดยไม่ต้องใช้ logic integration เพิ่มเติม

developer ไม่จำเป็นต้องเขียนหรือทำสำเนา code integration ใหม่สำหรับแต่ละ platform ระบบใดก็ตามที่รองรับ MCP สามารถเชื่อมต่อและใช้ functionality ได้อย่างราบรื่น

network effects และการเติบโตของ ecosystem

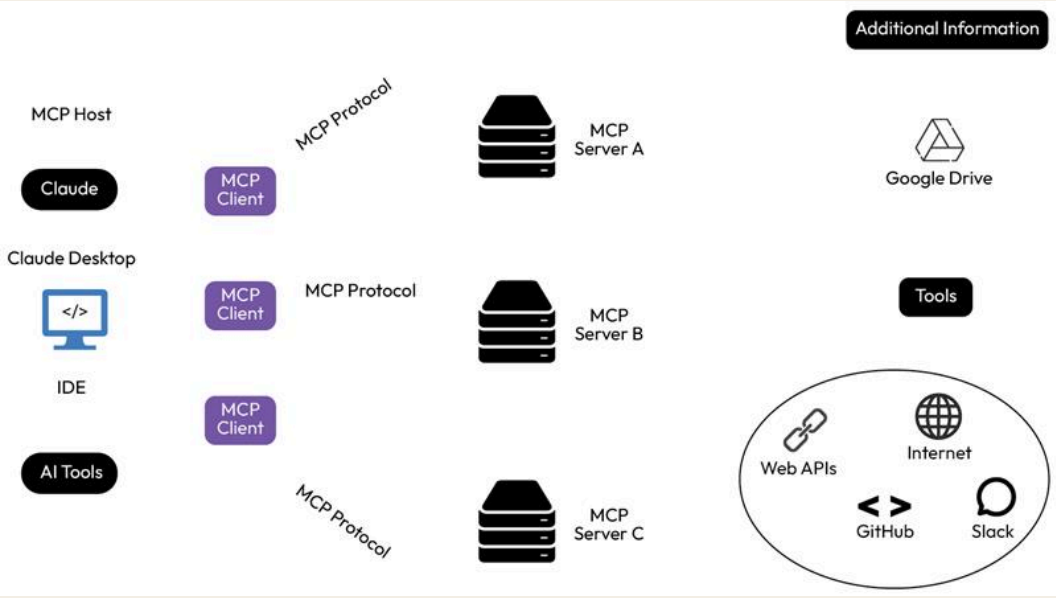
การเติบโตของ MCP สะท้อน network effects ที่คล้ายกับโซเชียล platform application โซเชียลที่มีฐาน user ขนาดเล็กให้คุณค่าได้จำกัด เมื่อมีการใช้งานเพิ่มขึ้นและ user ร่วมสร้าง content ระบบก็มีคุณค่ามากขึ้น จนเกิดเป็นวงจรที่ส่งเสริมการเติบโตอย่างต่อเนื่อง

MCP กำลังเกิดพลวัตในลักษณะคล้ายกัน เมื่อมีการใช้งานเพิ่มขึ้นและมี MCP server development มากขึ้น ecosystem ก็ขยายตัว จำนวน implementation ที่เพิ่มขึ้นช่วยขยายขอบเขตของ integration และความสามารถที่พร้อมใช้งาน ทำให้ protocol โดยรวมมีประโยชน์มากขึ้น

เมื่อวางพื้นฐานนี้แล้ว ขั้นต่อไปคือพิจารณา architecture หลักของ MCP และดูว่ามันทำงานอย่างไรในทางปฏิบัติ

architecture หลักของ MCP

MCP สร้างมาตรฐานให้กับวิธีที่ application ส่งมอบ context แก่ **large language model** (**LLMs**) context มีได้หลายรูปแบบ อาจเป็นข้อมูลเพิ่มเติมที่ผนวกเข้ากับ prompt requirement ของ tool ที่จะเรียกใช้ หรือแม้แต่ตัว prompt เอง



รูปที่ 4.5 – architecture หลักของ MCP ที่เชื่อมต่อ host client และ MCP server

ในแง่นี้ context ครอบคลุมทั้ง data สนับสนุนและคำสั่งด้านการทำงานที่กำหนดแนวทาง response ของ LLM

เมื่อมาตรฐานดังกล่าวได้รับการยอมรับอย่างแพร่หลาย ก็จะเอื้อให้เกิด integration development ที่ทรงพลังและขยายต่อได้ใน AI application หลากหลายประเภท

ข้อดีของ MCP

เอกสาร MCP (https://modelcontextprotocol.io/) เน้นย้ำข้อดีหลายประการของการนำ MCP มาใช้:

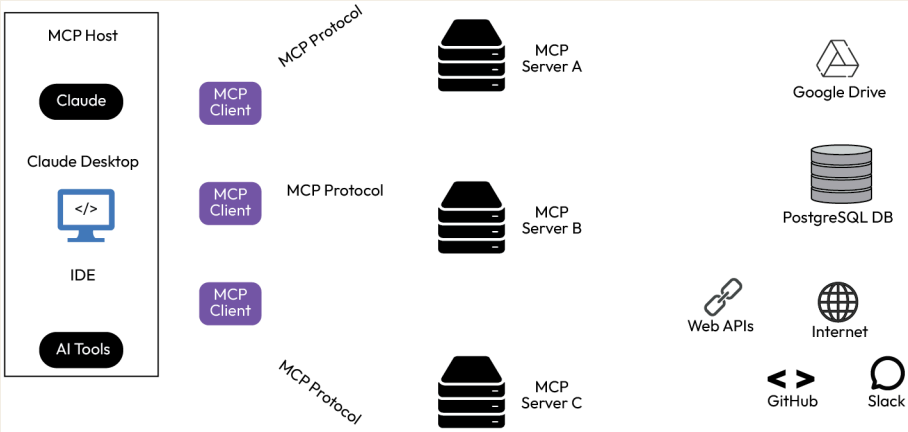
- ประการแรก MCP รองรับ ecosystem ขนาดใหญ่ของ integration tool และแหล่ง data ซึ่งสามารถเชื่อมต่อกับ application ที่ขับเคลื่อนด้วย LLM ได้ในรูปแบบ modular ตัวอย่างเช่น Claude Code ในฐานะ coding agent ที่ขับเคลื่อนด้วย AI ใช้ MCP เพื่อขยายความสามารถผ่าน integration tool อย่างเป็นระบบ
- ประการที่สอง MCP ลดการผูกติดกับผู้ให้บริการ LLM หรือผู้ให้บริการ AI application รายใดรายหนึ่ง developer สามารถนำไปใช้ functionality หลักเป็น tool และเปิดให้ใช้งานผ่าน MCP server จากนั้น tool เหล่านี้สามารถนำกลับมาใช้กับผู้ให้บริการและ application host ที่ต่างกันได้โดยไม่ต้องแก้ไข
- framework open source อย่าง LangChain แก้โจทย์บางส่วนที่ทับซ้อนกันในงาน AI engineering อย่างไรก็ตาม MCP และ LangChain ใช้แนวทางที่ต่างกันในการแก้ปัญหาเหล่านี้ และ solution ของทั้งคู่ไม่ได้ทับซ้อนกันโดยตรง ตัวอย่างเช่น ทั้งสองอาจรองรับการขยายความสามารถ แต่กลไกและผลกระทบทาง architecture แตกต่างกัน

component หลักของ MCP

architecture MCP ประกอบด้วย component หลักสามส่วน ได้แก่ MCP host, MCP server และ MCP client

MCP host

MCP host คือ AI application ที่รองรับ MCP ตัวอย่างได้แก่ Claude Desktop, development environment อย่าง Claude Code หรือ Cursor และ AI agent เฉพาะทาง application เหล่านี้ได้รับการเสริมความสามารถผ่าน MCP ด้วย connection เข้ากับ tool แหล่ง data หรือผลิตภัณฑ์เพิ่มเติมที่ไม่มีมาให้โดยกำเนิด



รูปที่ 4.6 – MCP host ที่เชื่อมต่อกับ MCP server ผ่าน MCP client

FREE PREVIEW · เลือกชั้นแก้ปัญหาให้ถูก

บทที่ 12 — Harness, Loop และ Graph

วินิจฉัยระบบ Agent ด้วยสามชั้น Environment > Feedback > Flow เพื่อเลือกแก้ runtime, วงรอบหลักฐาน หรือโครงสร้าง workflow ให้ตรงกับ failure ก่อนเพิ่ม Agent

- Harness กำหนด tool, state, permission และขอบเขต runtime
- Loop เปลี่ยนผลลัพธ์แต่ละรอบให้เป็น evidence ที่ตรวจซ้ำได้
- Graph กำหนด ownership, dependency, branch, join และ stop condition

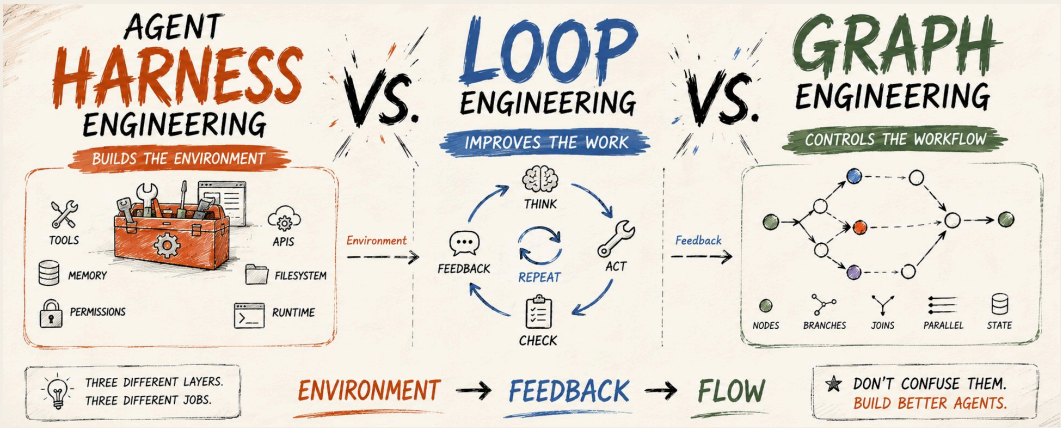
ตัวอย่างนี้คัดบางส่วนจากฉบับเต็มเพื่อให้เห็นแนวทางและรูปแบบจริง

Agent Harness, Loop และ Graph: Environment → Feedback → Flow

เมื่อระบบ agent ทำงานผิด ทีมมักแก้ด้วยการเพิ่ม prompt, tool, retry หรือ agent พร้อมกันจนไม่รู้ว่าอะไรแก้ปัญหาจริง กรอบจากบทความของ **beamnxw (`@beamnxw`)** ช่วยแยก architecture เป็นสามคำถาม:

1. **Agent Harness Engineering — Environment:** model ทำอะไรได้ ภายใต้ context, state, tool, permission และ observability แบบใด
2. **Loop Engineering — Feedback:** รอบถัดไปเริ่มเมื่อใด ใช้ evidence อะไรแก้งาน และหยุดด้วย proof, budget หรือ escalation แบบใด
3. **Graph Engineering — Flow:** component ใดมีสิทธิ์ทำงานต่อ output ใดไหลไป node ไหน และ branch, join, approval หรือ recovery เกิดตรงใด

คำย่อ **Environment → Feedback → Flow** เป็น mental model เชิงปฏิบัติของผู้เขียนบทความ ไม่ใช่ taxonomy ทางการของ Anthropic หรือมาตรฐานสากล การแยกสามชั้นมีประโยชน์เพราะ failure แต่ละชนิดต้องแก้คนละตำแหน่ง

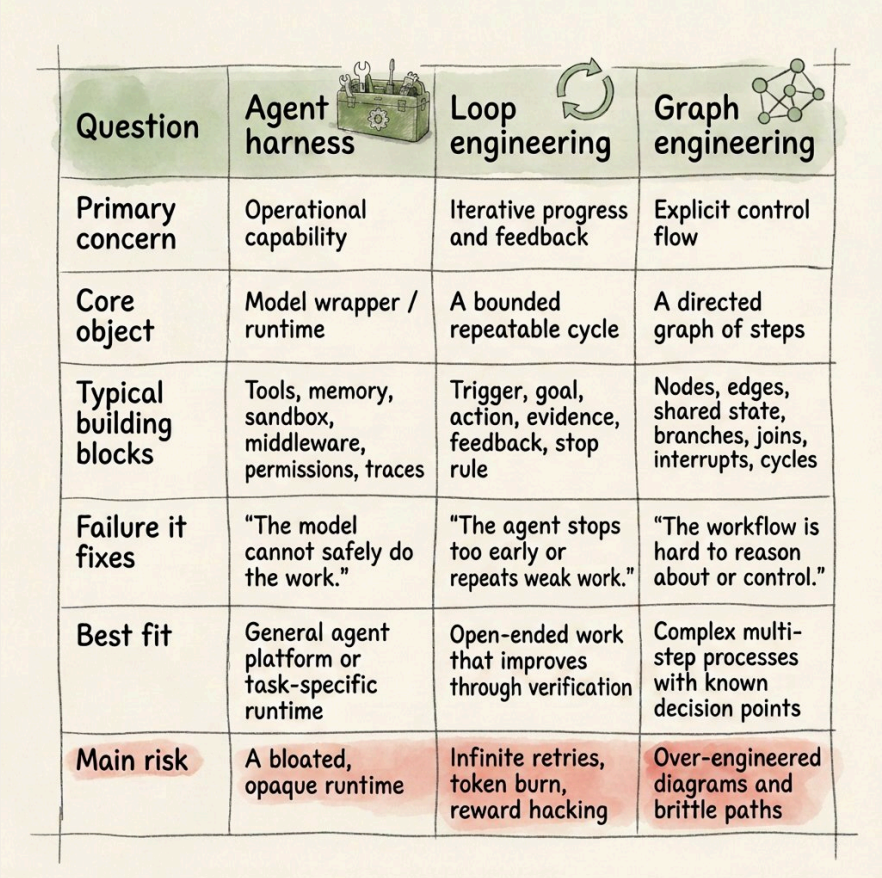


รูปที่ 12.3 – แผนที่สามชั้น: harness จัด environment, loop จัด feedback และ graph จัด flow ภาพจากบทความของ beamnxw / @beamnxw

ภาพและกรอบคิดต้นฉบับ: beamnxw, Agent Harness Engineering vs. Loop Engineering vs. Graph Engineering — วันที่เข้าถึง 26 กรกฎาคม 2026 เนื้อหาในเล่มสังเคราะห์ใหม่และตรวจเทียบกับแหล่งปฐมภูมิ ไม่ได้แปลเรียงประโยค ภาพใช้พร้อมเครดิตโดยไม่อ้าง permission หรือ endorsement

มิติ	Agent Harness Engineering	Loop Engineering	Graph Engineering
Primary concern	operational capability: model ทำงานจริงได้ภายใต้ environment แบบใด	iterative progress และ feedback: งานดีขึ้นจากหลักฐานอย่างไร	explicit control flow: component ใดมีสิทธิ์ทำงานต่อ
Core object	model wrapper หรือ runtime	วงรอบที่ทำซ้ำได้และมีขอบเขต	directed graph ของขั้นตอน
Typical building blocks	tool, memory, sandbox, middleware, permission และ trace	trigger, goal, action, evidence, feedback และ stop rule	node, edge, shared state, branch, join, interrupt และ controlled cycle
Failure ที่แก้	“model ทำงานนี้อย่างปลอดภัยไม่ได้”	“agent หยุดเร็วเกินไปหรือทำงานอ่อนซ้ำ”	“workflow ทำความเข้าใจหรือควบคุมได้ยาก”

มิติ	Agent Harness Engineering	Loop Engineering	Graph Engineering
Best fit	agent platform ทั่วไปหรือ runtime เฉพาะ task	งานปลายเปิดที่ดีขึ้นได้ผ่าน verification	กระบวนการหลายขั้นที่มีจุดตัดสินใจชัด
Main risk	runtime ใหญ่และทึบจน behavior คาดเดายาก	retry ไม่สิ้นสุด, token burn และ reward hacking	diagram เกินจำเป็นและ path เปราะ



รูปที่ 12.4 – ภาพเปรียบเทียบ concern, building blocks, failure signature, best fit และ risk ของทั้งสามชั้น ภาพจากบทความของ beamnxw / @beamnxw; ตารางภาษาไทยด้านบนเป็น text alternative ที่ค้นหาและอ่านด้วย screen reader ได้

ภาพต้นฉบับ: beamnxw, Agent Harness, Loop and Graph Engineering — วันที่เข้าถึง 26 กรกฎาคม 2026 ความหมายของ harness ตรวจเทียบกับ LangChain, The Anatomy of an Agent Harness; หลัก loop ตรวจเทียบกับ The Art of Loop Engineering; หลักเริ่มจาก architecture ที่เรียบง่ายตรวจเทียบกับ Anthropic, Building Effective AI Agents

วินิจฉัยจาก failure ก่อนเลือก architecture

สิ่งที่สังเกตเห็น	เริ่มแก้ที่ชั้น	การทดลองที่เล็กที่สุด
agent อ่านข้อมูลผิดชุดหรือเข้าถึง tool สำคัญไม่ได้	Harness	ลด tool contract, ตรวจ permission/sandbox และ log context ที่ส่งจริง
progress หายเมื่อ context compact หรือเริ่ม session ใหม่	Harness	เขียน checkpoint/progress file และ Git commit แล้วทดสอบ clean-context handoff

สิ่งที่สังเกตเห็น	เริ่มแก้ที่ชั้น	การทดลองที่เล็กที่สุด
first draft เกือบถูกแต่ผลไม่สม่ำเสมอ	**Loop**	เพิ่ม deterministic test หรือ grader ที่แยกจากผู้สร้าง แล้ว retry แบบจำกัดรอบ
agent หยุดโดยไม่มี proof หรือทำต่อหลังผ่านเกณฑ์แล้ว	**Loop**	นิยาม terminal state จาก evidence พร้อม max attempts, timeout และ escalation
specialist ต้องทำงานตามลำดับ, branch หรือ join ที่ควบคุมได้	**Graph**	วาด node/edge เฉพาะ dependency จริงและระบุ state ที่ข้าม edge
workflow หลายขั้นล้มเหลวแต่หาเจ้าของไม่ได้	**Graph + Harness**	ผูก trace กับ node, state transition และ tool event ให้ replay ได้
workflow เปลี่ยนแทบทุกสัปดาห์จน graph ต้องวาดใหม่ตลอด	**Harness ที่เรียบง่ายกว่า**	เก็บ execution trace จาก model-directed planning ก่อน formalize graph

ตารางนี้ป้องกันความผิดพลาดที่แพงมาก เช่น เพิ่ม agent เพราะ tool permission ผิด หรือสร้าง visual graph ทั้งระบบทั้งที่ปัญหาจริงคือ verifier ไม่มีหลักฐาน การใช้ model ที่เก่งขึ้นอาจช่วย reasoning แต่ไม่ชดเชย state ที่หาย, tool contract ที่กว้างเกินไป หรือ stop rule ที่ไม่มีอยู่

ทั้งสามชั้นประกอบกันได้หลายรูปแบบ ไม่มีกฎสากลว่า graph ต้องอยู่ “ภายใน” harness และ loop ต้องอยู่ “ภายใน” graph เสมอ runtime ของ graph อาจเป็นส่วนหนึ่งของ harness ส่วน loop อาจถูกเขียนเป็น cycle ใน graph หรือทำงานภายใน node เดียว ให้ใช้คำสามคำนี้เพื่อหา **เจ้าของของ failure** มากกว่าจะบังคับ diagram ทุกระบบให้เหมือนกัน

ใน Claude Code mapping ที่ใช้งานได้คือ:

- `CLAUDE.md`, Skills, MCP, tool, permission, sandbox, hook, filesystem และ Git เป็นส่วนของ **harness surface**
- `/goal`, `/loop`, Monitor, Routine, test, verifier, budget และ escalation เป็น **loop control**
- Dynamic workflow, agent team, dependency, fan-out/barrier/synthesis และ approval route เป็น **graph/control-flow concern**

หากต้องการรายละเอียด harness ให้ย้อนดู Harness Engineering ในบทที่ 11 ส่วนถัดไปจะเจาะ Loop Engineering และย้ำหลักสำคัญว่า **วนบน evidence ไม่ใช่ confidence** จากนั้นหัวข้อ Graph Engineering จะขยาย flow สำหรับ branch, parallelism, verification และ recovery

Loops in Claude Code: ออกแบบ Trigger, Verification และ Stop Condition

หัวข้อก่อนหน้าอธิบาย primitive แต่ละตัวแยกกันแล้ว ส่วนนี้จะนำ `/goal`, `/loop`, Monitor, Skills และ Routines มาวางในกรอบเดียว เพื่อให้คุณเลือก automation ได้จาก **Trigger** และ **Stop Condition** แทนการเลือกจากชื่อ command เพียงอย่างเดียว กรอบคิด 4 แบบต่อไปนี้เรียบเรียงใหม่จากบทความ Mastering Claude Code Loops แล้วตรวจสอบ behavior ของ command เทียบกับเอกสาร Claude Code ทางการ ณ วันที่ 11 กรกฎาคม 2026

กรอบ 4 แบบนี้เป็น taxonomy สำหรับออกแบบ workflow ไม่ใช่หมวด feature อย่างเป็นทางการของ Anthropic และคำว่า Loop ในส่วนนี้หมายถึง **orchestration/control loop** ที่ควบคุมการทำงานข้าม turn หรือข้ามเวลา ไม่ใช่ ReAct loop ภายใน model ที่อธิบายไว้ในบทที่ 11 และไม่ใช่ Infinite Agentic Loop สำหรับกระจาย subagent ในบทที่ 7

Expert Practice: Ralph-style loop ของ Ray เทียบกับ primitive รุ่นปัจจุบัน

Ray Amjad ใช้แนวคิด **Ralph-style loop** ในการสอน workflow ที่ agent ลงมือ ตรวจผล เรียนรู้จากรอบก่อน และทำซ้ำจนถึง outcome ดูภาพรวมแนวทางได้ที่ Master Claude Code คำว่า Ralph-style ในที่นี้เป็นชื่อ pattern ในชุมชน **ไม่ใช่ command ทางการของ Claude Code** เมื่อนำมาเทียบกับ primitive ที่บทนี้อธิบาย จะได้ mental model ดังนี้:

ความต้องการของ loop	primitive ที่ใกล้ที่สุด	สิ่งที่ต้องส่งมอบ
วนแก้ทันทีจน test/metric ผ่าน	`/goal`	completion condition ที่ evaluator ตรวจได้
ตรวจ state ตามเวลาใน session	`/loop` หรือ Monitor	trigger, wait state และ stop/escalation rule
ทำงานต่อเมื่อเครื่องปิดหรือรับ event	Routine	prompt, policy, environment และ review boundary
แยกงานจำนวนมากที่เป็นอิสระ	Dynamic workflow / subagent	task graph, owner, merge contract และ budget

หัวใจของ Ralph-style loop ไม่ใช่การสั่ง “ทำต่อไปเรื่อย ๆ” แต่คือ feedback ที่มีหลักฐานในแต่ละรอบ หาก iteration ไม่ได้เพิ่ม test evidence, state change หรือข้อมูล diagnosis ใหม่ ให้ตอบ `WAIT` หรือ `ESCALATE` แทนการแก้ code เพิ่มอย่างไร้ทิศทาง

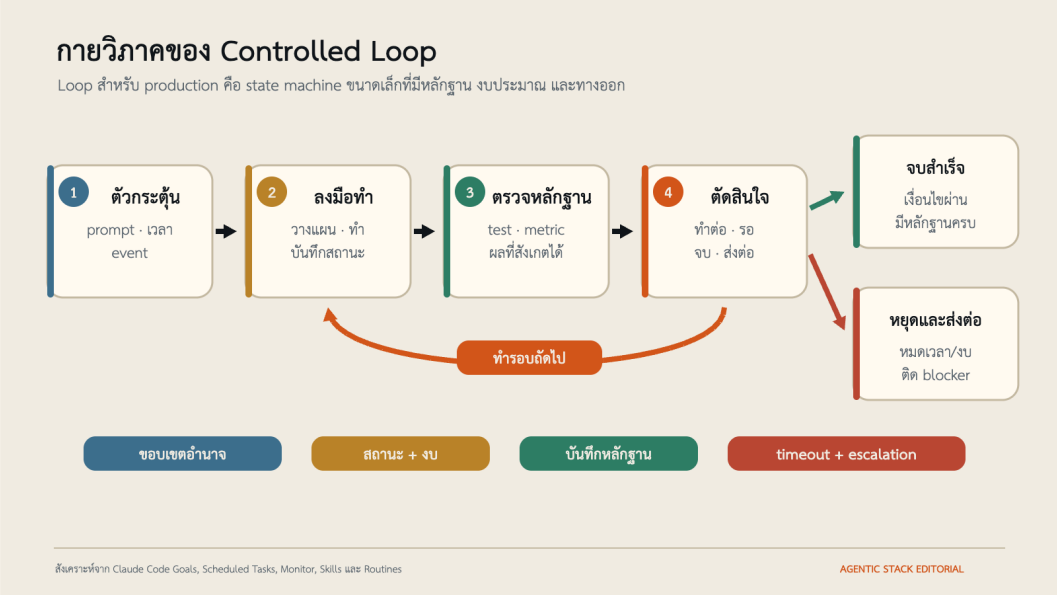
แหล่งอ่านเพิ่มเติม: Ray Amjad, Master Claude Code; Anthropic, Goals guide และ Routines guide — วันที่เข้าถึง 2026-07-13

Anatomy ของ Loop ที่ควบคุมได้

Loop ที่พร้อมใช้จริงควรนิยามองค์ประกอบอย่างน้อย 7 ส่วน:

1. **Trigger** — อะไรทำให้เริ่ม iteration ใหม่ เช่น user prompt, turn ก่อนหน้าจบ, interval ครบ, CI event หรือ schedule
2. **Work unit** — ในหนึ่ง iteration Claude อ่าน context และลงมือได้มากเพียงใด
3. **Verification** — หลักฐานใดใช้ตัดสินผล เช่น test exit code, metric, diff, queue length หรือ screenshot
4. **Stop Condition** — state ใดคือ `DONE` และ state ใดต้อง `WAIT`, `RETRY` หรือ `ESCALATE`
5. **Repeat policy** — วนทันที, รอ interval คงที่, ให้ Claude เลือก interval หรือรอ event
6. **Authority boundary** — operation ใดอนุญาตอัตโนมัติ และ operation ใดต้องผ่าน human approval
7. **Observability** — จะดู progress, token usage, error และเหตุผลที่หยุดจากที่ใด

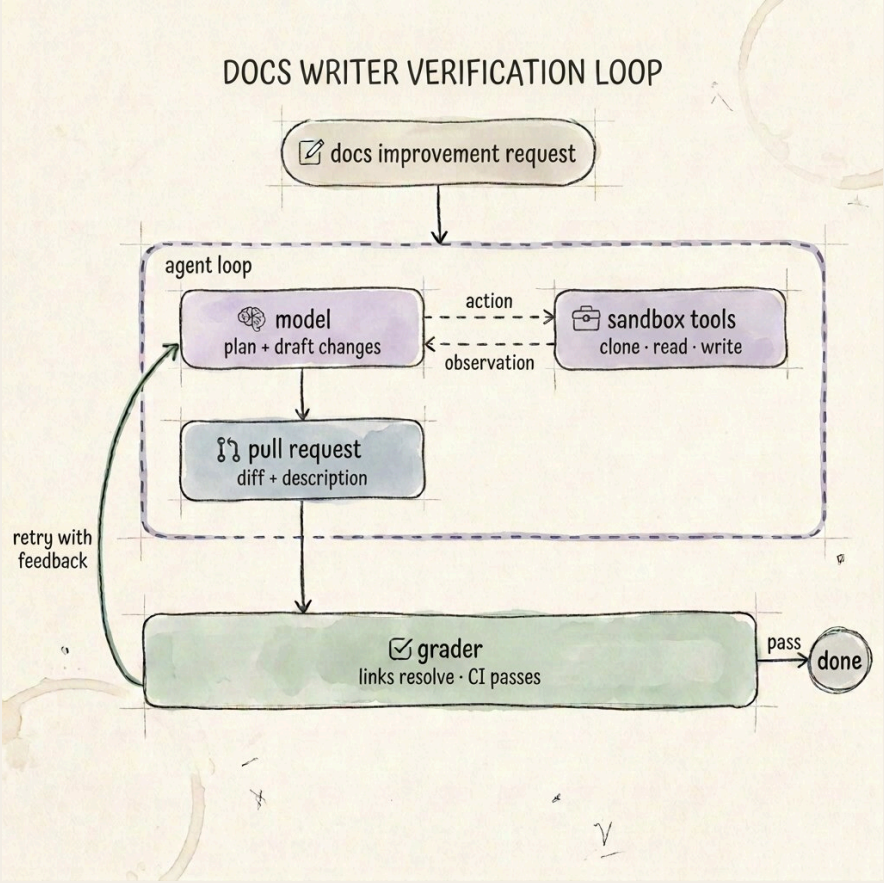
หากขาด Verification, Loop จะวนจากความรู้สึกว่า “น่าจะดีขึ้น” หากขาด Stop Condition จะไม่มีใครรู้ว่าควรหยุดเมื่อใด และหากขาด authority boundary ความผิดพลาดหนึ่งครั้งอาจกลายเป็นการเปลี่ยนแปลงที่ย้อนกลับยาก ดังนั้น Loop ที่ดีจึงไม่ใช่เพียง prompt ที่เรียกซ้ำ แต่เป็น state machine ขนาดเล็กที่มีหลักฐานและทางออกชัดเจน



รูปที่ 12.5 – anatomy ของ Loop ที่ควบคุมได้: ทุก iteration ต้องมี Trigger, Work, Verification, Decision และ Stop พร้อม authority boundary, state/budget, evidence log และ timeout/escalation แผนภาพต้นฉบับโดยกองบรรณาธิการ

อ่านกรณีศึกษาและตรวจ behavior กับแหล่งต้นฉบับ: Claude Developers, Loops automation guide; Anthropic, Goals, Scheduled tasks และ Routines — วันที่เข้าถึง 19 กรกฎาคม 2026

หลักใช้งานที่สั้นที่สุดคือ **loop on evidence, not confidence** อย่าให้ node ที่สร้างงานเป็นผู้ให้คะแนนตนเองจากคำว่า “น่าจะถูก” เพียงอย่างเดียว ส่ง output ให้ deterministic check หรือ grader ที่มี rubric ชัด แล้วส่ง failure evidence กลับไปยังผู้แก้ การวนต้องมีเพดาน attempts, token/cost, timeout และผู้รับ escalation เสมอ



รูปที่ 12.6 – ภาพแสดง writer ใช้ sandbox tool สร้าง pull request แล้วแยก grader ตรวจ links และ CI: ผ่านจึงจบ ไม่ผ่านจึง retry พร้อม feedback ส่วนเพดาน budget และทางออก ESCALATE เป็น guardrail ที่ต้องเพิ่มตามคำอธิบายก่อนภาพ ภาพจากบทความของ beamnxw / @beamnxw

ภาพและตัวอย่างต้นฉบับ: beamnxw, Agent Harness, Loop and Graph Engineering — วันที่เข้าถึง 26 กรกฎาคม 2026 โครง loop และต้นทุนการ retry ตรวจเทียบกับ LangChain, The Art of Loop Engineering

Loop type	Trigger	สิ่งที่ส่งมอบให้ Claude	Stop Condition	primitive ที่เหมาะ
Turn-based	user prompt	verification check ของงานหนึ่งชิ้น	งานเสร็จหรือขาด context สำคัญ	normal session และ Verification Skill
Goal-based	turn ก่อนหน้าจบ	completion condition	evaluator ยืนยันว่า condition เป็นจริง หรือถึง limit ที่เขียนไว้	`/goal` หรือ Stop hook
Time-based	interval หรือ background signal	trigger และกติกาของแต่ละ iteration	self-paced Loop หยุดเมื่องานเสร็จ; fixed Loop หยุดโดยผู้ใช้หรือเมื่อหมดอายุ	`/loop`, Monitor หรือ scheduled task
Proactive	schedule, API หรือ GitHub event	self-contained prompt, policy และ environment	แต่ละ run จบเมื่อ outcome สำเร็จ; automation คงอยู่จน pause/disable	Routine, Skills และ Dynamic workflow

Turn-based Loop: ส่งมอบ Verification Check

ทุก prompt เริ่ม agentic loop แบบพื้นฐาน: Claude รวบรวม context, ลงมือ, ตรวจผล และวนซ้ำภายใน turn จนเชื่อว่างานเสร็จ จากนั้นผู้ใช้ตรวจอีกครั้งและส่ง prompt ถัดไป Turn-based Loop จึงเหมาะกับงาน ad hoc, การสำรวจ solution space หรือ task ที่ยังไม่มี Stop Condition เชิงตัวเลข

สิ่งที่ควรส่งมอบไม่ใช่เพียงคำสั่งว่าให้ “แก้ bug” แต่คือ verification check ที่ปิดช่องว่างระหว่าง code edit กับ behavior จริง ตัวอย่างเช่น:

```
Fix the failing authentication test. Before reporting completion:
1. reproduce the failure with the focused test,
2. apply the smallest safe patch,
3. rerun the focused test and the auth regression suite,
4. inspect the final diff for unrelated changes, and
5. report the exact commands and exit codes.
```

รูปที่ 12.7 – Turn-based Loop ปิดหนึ่งรอบด้วย Verification Check ใน response เดียว: รวบรวม context ลงมือ ตรวจหลักฐาน แล้วรายงานสิ่งที่เปลี่ยน สิ่งที่ตรวจ และสิ่งที่ยังไม่แน่นอน แผนภาพต้นฉบับโดยกองบรรณาธิการ

อ่าน workflow ต้นฉบับที่ใช้เทียบเคียง: Claude Developers, Loops automation guide — วันที่เข้าถึง 19 กรกฎาคม 2026

หากทีมทำ verification แบบเดิมซ้ำบ่อย ให้ย้ายขั้นตอนนั้นไปไว้ใน project Skill ตามบทที่ 9 เพื่อให้ทุก Turn-based Loop ใช้มาตรฐานเดียวกัน การเพิ่ม Skill มีประโยชน์กว่าการทำ prompt ให้ยาวขึ้นเรื่อย ๆ เพราะ Skill เป็น reusable contract ที่แก้ไขและ review ผ่าน version control ได้

Goal-based Loop: ส่งมอบ Completion Condition

Goal-based Loop เหมาะเมื่อคุณรู้ว่า “เสร็จ” หน้าตาเป็นอย่างไร แต่หนึ่ง turn อาจไม่พอ `/goal` จะเริ่ม turn ถัดไปทันทีหลัง evaluator ตรวจว่ายังไม่ผ่าน จึงเหมาะกับ local migration, test repair, lint cleanup หรือ backlog ที่มีจำนวนรายการตรวจได้ รายละเอียด command อยู่ในหัวข้อ ใช้ /goal กับ completion condition ที่ตรวจสอบได้

อ่านต่อในฉบับเต็ม

ครบทั้ง 12 บท · 377 หน้า

รับ E-book + Implementation Kit พร้อม 60-Minute Quick Start, 7-Day Agentic Delivery Path และ Template แบบ copy-ready รวม Agent Harness, Evidence Loop, Graph Engineering, Dynamic workflows, กรณีศึกษา Bun และภาพอธิบายพร้อมเครดิตครบชุด

ดูรายละเอียดฉบับเต็ม

https://www.contextagentict.com