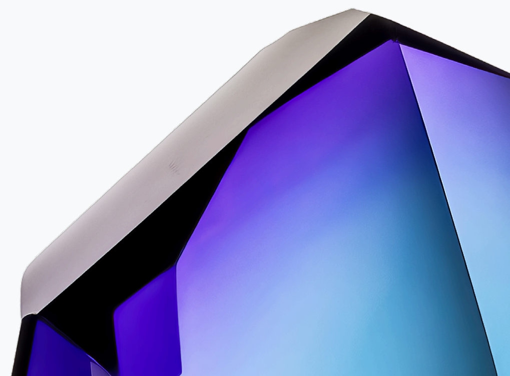
Vòng đời phát triển phần mềm thời Vibe Coding

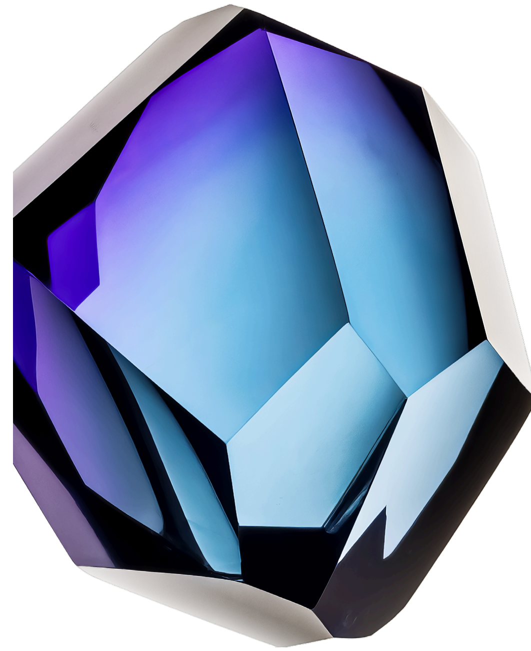
Từ prompt tùy hứng đến Agentic Engineering

Tác giả: Addy Osmani, Shubham Saboo,
và Sokratis Kartakis



Phiên bản tiếng Việt biên dịch và Việt hóa bởi Phong Hồ.

Lời cảm ơn



Đóng góp nội dung

Elia Secchi

Julia Wiesinger

Anant Nawalgaria

Tuyển chọn và biên tập

Anant Nawalgaria

Thiết kế

Michael Lanning

Mục lục

Giới thiệu ... 6

Vì sao có tài liệu này, và vì sao là lúc này ... 9

Tài liệu này dành cho ai ... 9

Bước chuyển từ cú pháp sang ý định ... 10

Agent AI: ôn lại nhanh ... 10

Vibe coding là gì? ... 11

Dải quang phổ: từ vibe coding đến agentic engineering ... 12

Context engineering: kỹ năng thực thụ ... 15

Vòng đời phát triển phần mềm kiểu mới ... 19

SDLC truyền thống dưới áp lực ... 19

AI biến đổi từng giai đoạn ra sao ... 21

Thu thập yêu cầu và lập kế hoạch ... 21

Thiết kế và kiến trúc ... 21

Lập trình ... 22

Kiểm thử và đảm bảo chất lượng ... 22

Mục lục

Review code và triển khai 23

Bảo trì và tiến hóa 24

Mô hình nhà máy: xây hệ thống tạo ra phần mềm 24

Kỹ thuật Harness: thứ bao quanh model 26

Trong harness có gì 28

Harness trong SDLC 29

1. Yêu cầu, lập kế hoạch và kiến trúc (Cấu hình harness) 29

2. Triển khai (Vận hành harness) 29

3. Kiểm thử và QA (Vòng phản hồi) 30

4. Review code, triển khai và bảo trì (Quan sát harness) 30

Vai trò đang đổi của lập trình viên: nhạc trưởng và người điều phối 31

Nhạc trưởng: trực tiếp, chỉ huy theo thời gian thực 32

Người điều phối: bất đồng bộ, giao việc cho nhiều agent 33

Bài toán 80% 34

Coding agent trong thực tế 35

Mục lục

Coding agent xuất hiện ở đâu trong ngày làm việc **35**

Vibe coding để dựng agent sẵn sàng chạy thật 36

Bài toán kinh tế của phát triển phần mềm bằng AI **39**

Món nợ ẩn của vibe coding (CapEx thấp, OpEx cao) 40

Khoản đầu tư của agentic engineering (CapEx cao, OpEx thấp) 41

Context engineering như một đòn bẩy tài chính 41

Mở rộng hiệu quả nhờ ngữ cảnh động và Skill 42

Định tuyến model thông minh 42

Bắt đầu từ đâu **43**

Với lập trình viên cá nhân 43

Với lãnh đạo kỹ thuật 44

Với tổ chức 45

Kết luận: Ý định là giao diện mới **47**

Ghi chú nguồn **49**

> Bước chuyển sâu sắc nhất trong ngành kỹ thuật phần mềm không phải là một ngôn ngữ, framework hay dịch vụ đám mây mới. Đó là sự dịch chuyển từ viết code sang biểu đạt ý định, và tin tưởng các hệ thống thông minh dịch ý định ấy thành phần mềm chạy được.

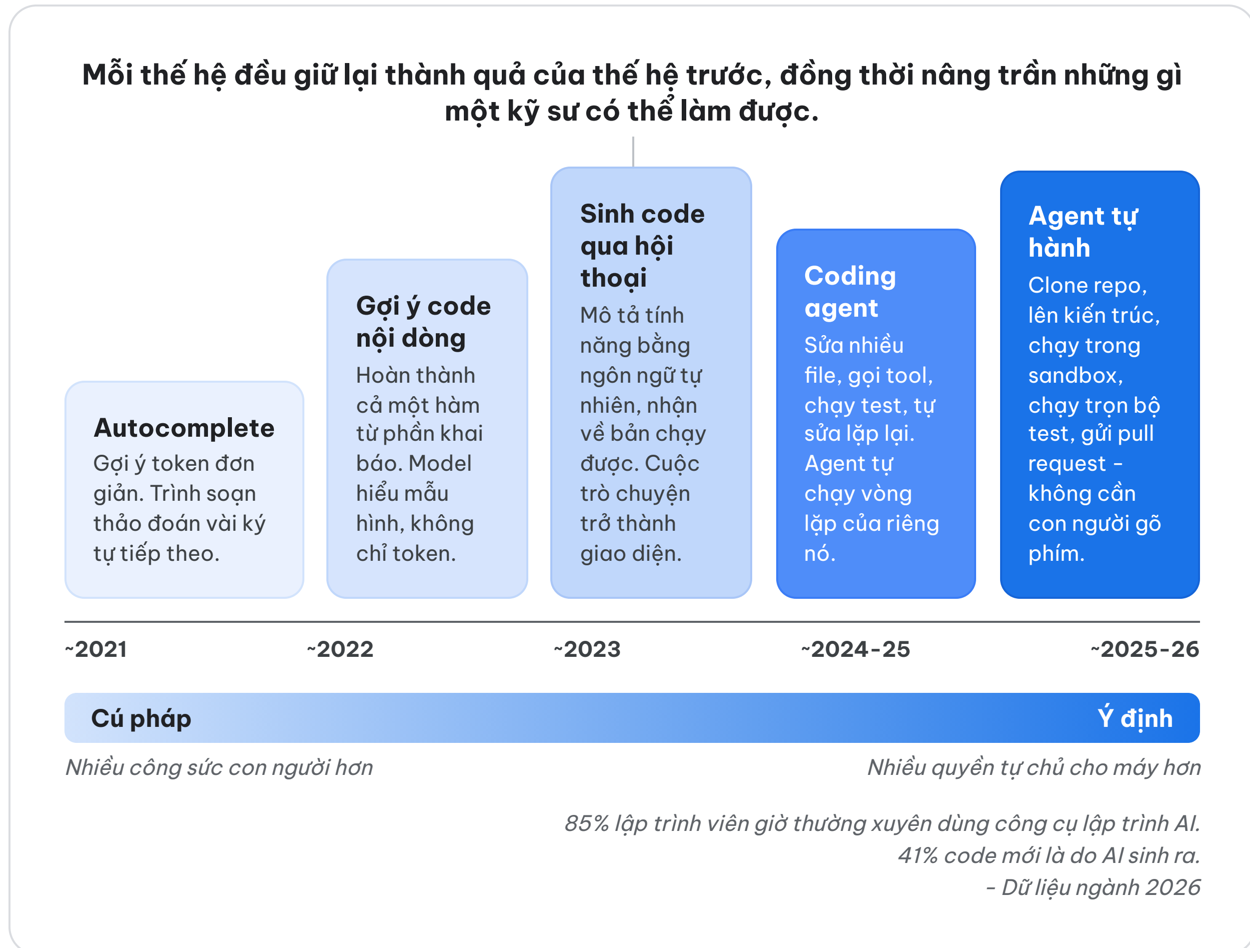
Giới thiệu

Trong gần như suốt lịch sử điện toán, lập trình là một hành vi phiên dịch: hiểu vấn đề theo ngôn ngữ con người, thiết kế lời giải ở mức trừu tượng, rồi diễn đạt nó bằng cú pháp mà máy có thể thực thi. Mỗi bước đều sinh ra ma sát. Thứ ma sát ấy giờ đang xẹp dần. Ngành kỹ thuật phần mềm đang trải qua cuộc chuyển đổi lớn nhất kể từ khi

các ngôn ngữ lập trình bậc cao ra đời. Suốt nhiều thập kỷ, giao diện chính giữa lập trình viên và máy là cú pháp: dấu ngoặc nhọn, dấu chấm phẩy, khai báo kiểu, và thứ ngữ pháp chính xác của các ngôn ngữ lập trình. Thời ấy đang khép lại.

Một mô thức mới đã đến, nơi lập trình viên biểu đạt thứ họ muốn xây thay vì cách xây nó. Máy lo phần triển khai. Con người đưa ra ý định, kiến trúc và phán đoán. Đây không phải tương lai xa - nó là thực tế hằng ngày của một lượng lập trình viên chuyên nghiệp đang tăng nhanh. Tính đến đầu năm 2026, 85% lập trình viên chuyên nghiệp thường xuyên dùng coding agent AI, 51% dùng hằng ngày, và ước tính 41% toàn bộ code mới là do AI sinh ra.[1]

Bước dịch chuyển này không diễn ra sau một đêm. Nó bắt đầu từ autocomplete - dự đoán token đơn giản ngay trong trình soạn thảo. Rồi đến gợi ý code nội dòng có thể hoàn thành trọn một hàm. Tiếp theo, các giao diện hội thoại cho phép lập trình viên mô tả tính năng bằng ngôn ngữ tự nhiên và nhận về bản triển khai chạy được. Giờ đây, những agent hoàn toàn tự hành có thể clone repo, lên kế hoạch thay đổi nhiều file, thực thi trong môi trường sandbox, chạy test và gửi pull request - tất cả mà không cần con người gõ một dòng code nào.



Hình 1: Từ Autocomplete đến Tự hành.

Hệ quả với vòng đời phát triển phần mềm (SDLC) là rất sâu rộng. Mọi giai đoạn - từ thu thập yêu cầu, triển khai cho tới bảo trì - đều đang được năng lực AI định hình lại. Nhưng cuộc chuyển đổi này không đồng đều hay đơn giản. Dải quang phổ trải từ "vibe coding" tùy hứng, nơi lập trình viên ra lệnh cho AI rồi nhận gì lấy nấy, đến "agentic engineering" bài bản, nơi AI đóng vai một cỗ máy triển khai mạnh mẽ bên trong những hệ thống ràng buộc, test và vòng phản hồi được thiết kế kỹ, còn con người vẫn nắm quyền giám sát kiến trúc, tính đúng đắn và chất lượng.

Sự phân biệt này quan trọng. Nói với một CTO rằng đội của bạn đang vibe coding hệ thống xử lý thanh toán thì sẽ - và nên - gióng chuông báo động. Nói với chính CTO đó rằng đội của bạn thực hành agentic engineering, với AI lo phần triển khai dưới những ràng buộc do con người thiết kế trong khi độ phủ test đảm bảo tính đúng đắn, lại là một cuộc trò chuyện hoàn toàn khác.

Tài liệu này đặt nền cho cuộc trò chuyện đó. Chúng tôi lần theo dải quang phổ từ vibe coding tùy hứng đến agentic engineering bài bản, xem xét vai trò của lập trình viên đang dịch chuyển ra sao - từ viết code sang vận dụng phán đoán, từ nhạc trưởng sang người điều phối - và phác ra những gì cần có để áp dụng các công cụ này theo cách tạo ra phần mềm mà bạn thật sự có thể dựa vào.

Vì sao có tài liệu này, và vì sao là lúc này

Công cụ, năng lực và mô thức mới xuất hiện mỗi tuần. Các đội kỹ thuật cần một khung tư duy để hiểu bức tranh này - không phải một bức ảnh chụp nhanh sẽ lỗi thời sau vài tháng, mà một bộ nguyên tắc và mô hình tư duy vẫn còn hữu ích khi các công cụ cụ thể thay đổi.

Tài liệu này dành cho ai

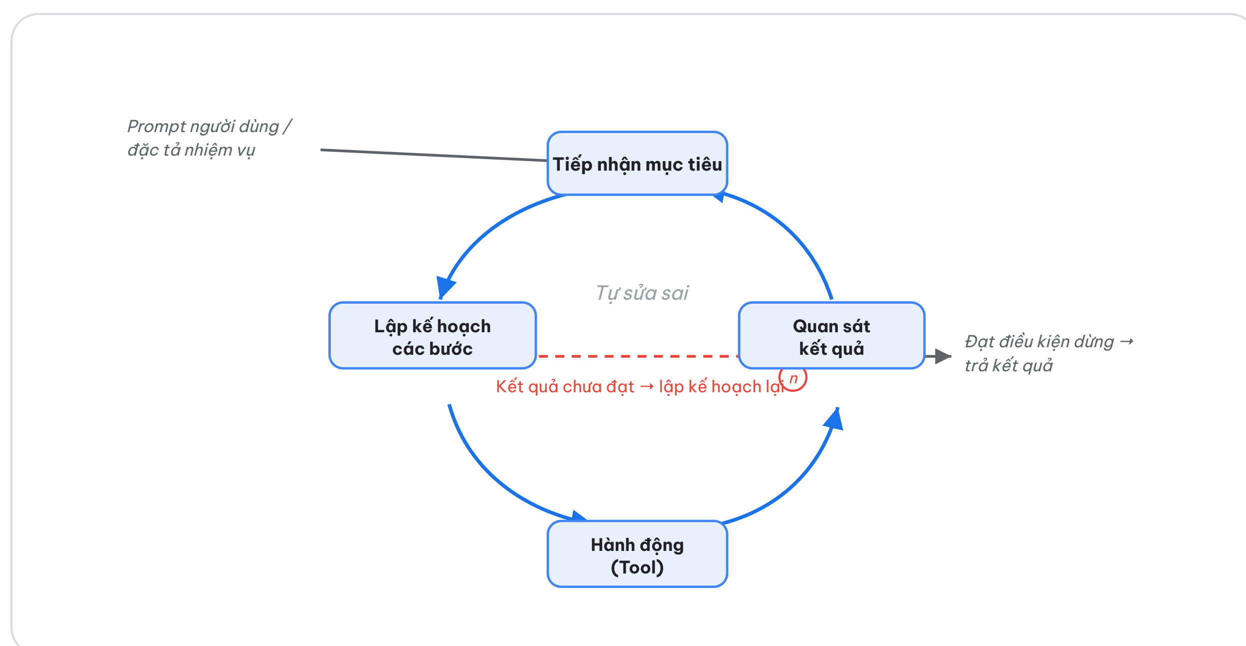
Tài liệu dành cho kỹ sư phần mềm, quản lý kỹ thuật, kiến trúc sư và lãnh đạo công nghệ muốn hiểu AI đang định hình lại SDLC ra sao, và áp dụng những năng lực mới này mà không đánh mất tính kỷ luật mà phần mềm chạy thật đòi hỏi. Chúng tôi giả định bạn đã quen với cách phát triển phần mềm hiện đại, nhưng không cần rành những chi tiết riêng của AI hay máy học.

Bước chuyển từ cú pháp sang ý định

Trước khi đi xa hơn, ta cần một bức tranh chung về agent là gì và vibe coding thực sự nghĩa là gì. Cả hai thuật ngữ đã tích tụ đủ nhiều tầng nghĩa đến mức cần được mổ xẻ cẩn thận.

Agent AI: ôn lại nhanh

Một agent AI là một hệ thống phần mềm cảm nhận một mục tiêu, lập kế hoạch các bước để đạt nó, hành động thông qua các tool, quan sát kết quả, rồi lặp lại cho đến khi đạt mục tiêu hoặc chạm một điều kiện dừng. Trong khi một chatbot tạo ra một câu trả lời rồi chờ prompt tiếp theo, một agent tự chạy vòng lặp của riêng nó. Bạn giao cho nó một mục tiêu ở trên cùng, rồi ở mỗi bước nó tự quyết định việc cần làm tiếp theo.



Hình 2: Vòng lặp của agent – Tiếp nhận, lập kế hoạch, hành động, quan sát, lặp lại.

Mọi agent, dù đơn giản hay tinh vi, đều được dựng từ năm thành phần. Tài liệu *Introduction to Agents* (tháng 11 năm 2025) bàn sâu về từng phần.[11] Ở đây, bản tóm gọn:

- **Model** là cỗ máy suy luận. Nó đọc ngữ cảnh hiện tại, quyết định điều gì nên xảy ra tiếp theo, rồi tạo ra suy nghĩ kế tiếp, lệnh gọi tool kế tiếp, hoặc tin nhắn kế tiếp.
- **Tool** kết nối model với thế giới. Chúng gồm các API mà agent có thể gọi, code nó có thể thực thi, cơ sở dữ liệu nó có thể truy vấn, và những agent khác nó có thể giao việc.
- **Bộ nhớ (Memory)** là trạng thái. Nó cho phép agent nhớ lại các tương tác trước, lấy ra những quy tắc riêng của dự án, và giữ ngữ cảnh xuyên suốt các phiên để không bao giờ phải bắt đầu từ con số không.
- **Điều phối (Orchestration)** là phần code vận hành vòng lặp. Nó tập hợp ngữ cảnh cho mỗi lần gọi model, gửi đi các lệnh gọi tool, thu lại kết quả, và quyết định có nên tiếp tục hay không.
- **Triển khai (Deployment)** là thứ biến nguyên mẫu thành một dịch vụ: hosting, định danh, khả năng quan sát, và hạ tầng chạy thật mà agent vận hành trên đó.

Những thành phần này phối hợp với nhau trong một vòng lặp liên tục: nhận nhiệm vụ, quét bối cảnh, suy nghĩ thấu đáo, ra tay hành động, quan sát rồi lặp lại. Vòng lặp chính là trái tim đập của mọi agent. Mọi thứ khác trong tài liệu này, và mọi thứ trong phần còn lại của khóa học, đều là một biến thể của vòng lặp này.

Vibe coding là gì?

Tháng 2 năm 2025, Andrej Karpathy đăng một mô tả về một cách lập trình mới, gây tiếng vang rộng khắp cộng đồng kỹ thuật phần mềm. Ông mô tả một lối làm mà ở đó bạn "buông mình hoàn toàn theo cảm hứng, ôm lấy đà tăng theo cấp số nhân, và quên rằng code thậm chí

còn tồn tại." Ở chế độ này, lập trình viên mô tả thứ mình muốn bằng ngôn ngữ tự nhiên, chấp nhận đầu ra của AI, và khi có gì đó hỏng thì copy thông báo lỗi dán ngược lại vào prompt rồi nhờ AI sửa.[2]

Thuật ngữ này lan truyền chóng mặt vì nó nắm bắt được một điều có thật: nhiều lập trình viên vốn đã làm theo cách này nhưng chưa có từ để gọi tên. Chỉ trong vài tháng, "vibe coding" trở thành cách gọi chung cho bất kỳ quy trình phát triển nào có AI hỗ trợ, và điều đó gây nhầm lẫn. Một kỹ sư cấp cao dùng trợ lý AI để triển khai một tính năng đã được đặc tả kỹ có phải là "vibe coding" không? Một đội dùng agent AI để hiện thực một kiến trúc đã lên kế hoạch cẩn thận thì sao? Thuật ngữ bị dùng rộng đến mức bắt đầu mất nghĩa.

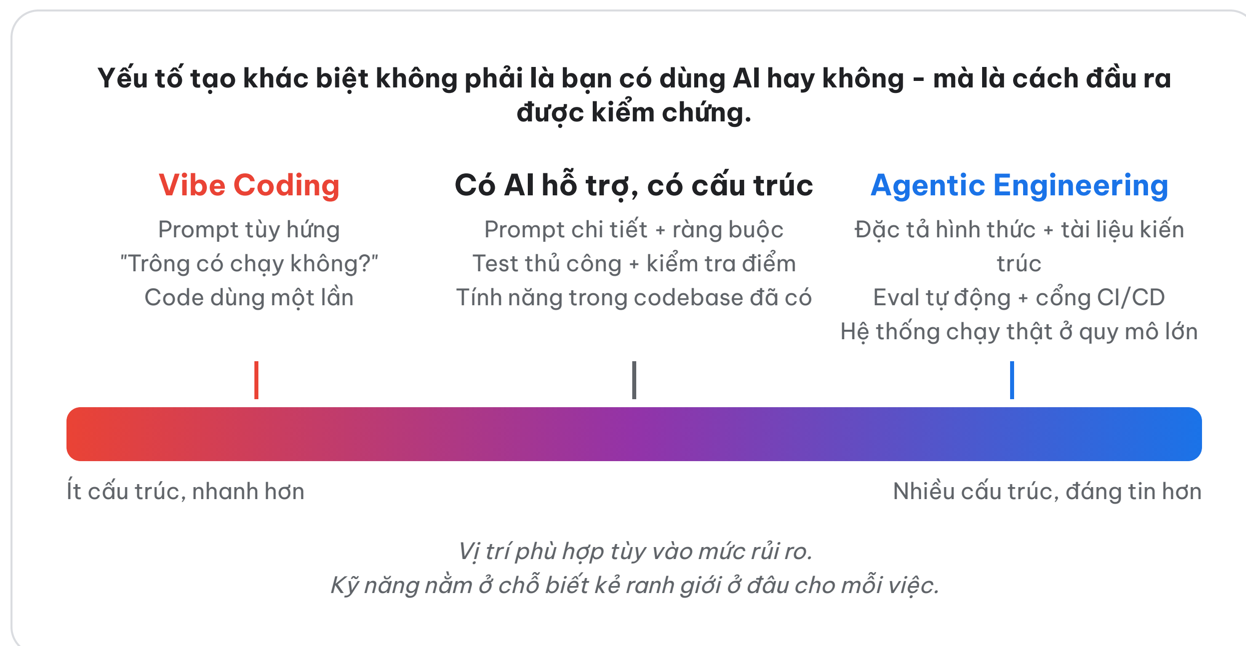
Đến đầu năm 2026, chính Karpathy thừa nhận cách định khung ban đầu là quá hẹp, và đưa ra thuật ngữ "agentic engineering" để chỉ đầu bài bản hơn của dải quang phổ.[4]

Dải quang phổ: từ vibe coding đến agentic engineering

Thay vì xem vibe coding và agentic engineering như hai thái cực nhị phân, chúng tôi thấy hữu ích hơn khi coi chúng là hai đầu của một dải quang phổ. Điểm khác biệt mấu chốt không nằm ở chuyện bạn có dùng AI hay không. Nó nằm ở lượng cấu trúc, sự kiểm chứng và phán đoán của con người bao quanh đầu ra của AI.

Tiêu chí	Vibe Coding	Lập trình có AI hỗ trợ, có cấu trúc	Agentic Engineering
Cách đặc tả ý định	Prompt ngôn ngữ tự nhiên, tùy hứng	Prompt chi tiết kèm ví dụ và ràng buộc	Đặc tả hình thức, tài liệu kiến trúc, file bộ nhớ
Kiểm chứng	"Trông có vẻ chạy được?"	Test thủ công, kiểm tra điểm	Bộ test tự động, cổng CI/CD, LM chấm điểm
Hiểu codebase	Tối thiểu; lập trình viên có thể không đọc code được sinh ra	Review chọn lọc các đường đi quan trọng	Review toàn diện kiến trúc; AI lo chi tiết triển khai
Xử lý lỗi	Copy-paste thông báo lỗi ngược lại cho AI	Lập trình viên chẩn đoán căn nguyên, AI hiện thực bản sửa	Agent tự chẩn đoán trong giới hạn đã định; con người xử lý vấn đề kiến trúc
Phạm vi phù hợp	Nguyên mẫu, script, dự án cá nhân, hackathon	Tính năng trong codebase đã có sẵn	Hệ thống chạy thật, phát triển ở quy mô đội
Mức rủi ro	Cao; chấp nhận được với code dùng một lần	Vừa phải; con người phán đoán tại các chốt quan trọng	Thấp; kiểm chứng hệ thống ở mọi giai đoạn

Bảng 1: Dải quang phổ từ Vibe Coding đến Agentic Engineering.



Hình 3: Dải quang phổ từ Vibe Coding đến Agentic Engineering.

💡 **Mẹo áp dụng:**

Vị trí phù hợp trên dải quang phổ này tùy vào mức rủi ro. Một nguyên mẫu làm cuối tuần có thể vibe coding thuần túy. Một API chạy thật xử lý giao dịch tài chính thì đòi hỏi agentic engineering. Phần lớn công việc thực tế nằm đâu đó ở giữa, và kỹ năng chính là biết kẻ ranh giới ở đâu cho từng việc.

Khác biệt lớn nhất giữa hai đầu là cách đầu ra được kiểm chứng. Trong vibe coding, kiểm chứng là tùy chọn; lập trình viên chạy code và xem nó có vẻ ổn không. Trong agentic engineering, hai cơ chế cùng làm việc. Test kiểm chứng những phần tất định của hệ thống: một hàm với đầu vào này sẽ cho ra đầu ra kia. Đánh giá, hay eval, thì kiểm chứng những

những phần không tất định: liệu agent có đi đúng lộ trình các bước, chọn đúng tool, và tạo ra câu trả lời cuối cùng đạt chuẩn chất lượng hay không. Test thì được kiểm bằng code; eval thì được kiểm bằng tập dữ liệu đã gán nhãn, các thang chấm điểm, và LM chấm điểm. Thiếu một trong hai, thì dù prompt có tinh vi đến đâu, việc đang làm vẫn luôn là vibe coding.

Context engineering: kỹ năng thực thụ

Khi lĩnh vực này trưởng thành, một nhận thức then chốt đã xuất hiện: chất lượng code do AI sinh ra phụ thuộc ít vào độ khôn khéo của prompt mà nhiều vào chất lượng của ngữ cảnh được cung cấp. Nhận thức này khai sinh khái niệm context engineering - việc cung cấp cho agent AI thông tin phong phú, có cấu trúc về codebase, kiến trúc, quy ước và ý định của bạn.[5]

Lập trình viên cần cân nhắc sáu loại ngữ cảnh chính:

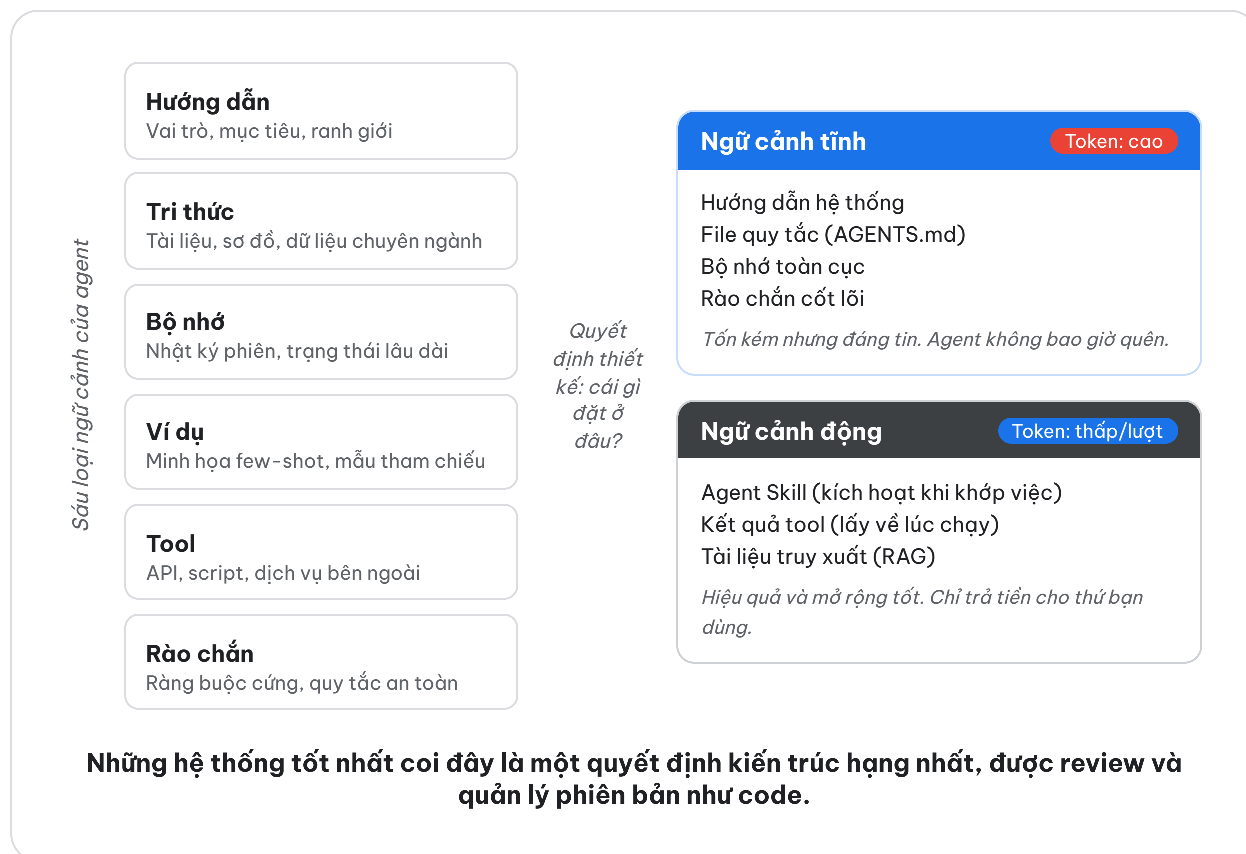
- **Hướng dẫn (Instructions):** Vai trò cốt lõi, mục tiêu và ranh giới hoạt động của agent.
- **Tri thức (Knowledge):** Tài liệu được truy xuất, sơ đồ kiến trúc, và dữ liệu chuyên ngành.
- **Bộ nhớ (Memory):** Nhật ký phiên ngắn hạn (việc vừa xảy ra) và trạng thái lưu giữ dài hạn (dự án là gì).
- **Ví dụ (Examples):** Các minh họa hành vi kiểu few-shot và mẫu hình tham chiếu trong codebase.
- **Tool:** Định nghĩa chính xác về các API, script và dịch vụ bên ngoài mà agent có thể gọi.
- **Rào chắn (Guardrails):** Các ràng buộc cứng, quy tắc định dạng, và kiểm tra an toàn.

Trong việc sinh code bằng AI, context engineering là sự cân bằng cẩn thận: trong sáu yếu tố này, cái nào agent nắm sẵn từ đầu, cái nào nó có thể truy xuất khi cần. Điều này tạo ra một sự tách biệt then chốt giữa ngữ cảnh tĩnh và ngữ cảnh động.

Ngữ cảnh tĩnh luôn được nạp: hướng dẫn hệ thống, file quy tắc (AGENTS.md, CLAUDE.md, GEMINI.md), bộ nhớ toàn cục, và định nghĩa persona. Nó xác định agent là ai và hành xử ra sao. Ngữ cảnh tĩnh tốn kém vì mọi token đều hiện diện trong mọi tương tác, bất kể có liên quan hay không.

Ngữ cảnh động được nạp khi cần: hướng dẫn của skill kích hoạt khi khớp nhiệm vụ, kết quả tool lấy về trong lúc thực thi, tài liệu truy xuất từ pipeline RAG, và lịch sử phiên theo cửa sổ. Ngữ cảnh động hiệu quả vì agent chỉ trả chi phí token khi thật sự cần đến thông tin đó.

Quyết định thiết kế về việc cái gì thuộc ngữ cảnh tĩnh, cái gì thuộc ngữ cảnh động là một sự đánh đổi kỹ thuật thực thụ. Quá nhiều ngữ cảnh tĩnh thì phí token và làm loãng tín hiệu. Quá ít thì agent quên mất các quy tắc quan trọng. Những hệ thống tốt nhất coi ranh giới này như một quyết định kiến trúc hạng nhất, được review và quản lý phiên bản như mọi cấu hình khác.



Hình 4: Context Engineering - Tĩnh và Động.

Mẫu hình mạnh nhất để quản lý ngữ cảnh động là **Agent Skill**: những gói tri thức quy trình có cấu trúc, dùng được đa nền tảng, mà agent chỉ nạp khi nhiệm vụ cần đến.

Thay vì nhồi mọi mảnh tri thức chuyên biệt vào system prompt của agent, các skill cho phép agent giữ vai một người đa năng gọn nhẹ, linh hoạt khoác lên vai trò chuyên gia khi cần thông qua cơ chế bộc lộ dần (progressive disclosure). Lúc khởi động, agent chỉ thấy phần metadata gọn nhẹ; nạp đầy đủ hướng dẫn khi một nhiệm vụ khớp; và chỉ kéo về tài liệu tham khảo chuyên sâu khi thật sự cần. Kết quả là một agent có thể mang theo hàng chục năng lực chuyên biệt mà chỉ trả chi phí token cho đúng năng lực đang dùng.

Agent Skill được đón nhận nhanh chóng trên khắp các coding agent lớn và nền tảng doanh nghiệp vì chúng giải quyết bốn vấn đề vốn đeo bám việc phát triển agent AI:

- Ngữ cảnh bị "mục ruỗng" do prompt quá tải (context rot)
- Thiếu bộ nhớ quy trình cho các LLM
- Chi phí vận hành của các kiến trúc nhiều agent
- Nhu cầu dùng được đa nền tảng, đa công cụ và nhà cung cấp

Phần này đã giới thiệu các nguyên lý cốt lõi của context engineering: sáu loại ngữ cảnh mà mọi agent cần, sự đánh đổi giữa ngữ cảnh tĩnh và động, và Agent Skill như mẫu hình then chốt để quản lý sự đánh đổi đó ở quy mô lớn.

Tài liệu Ngày 3 đi kèm trong loạt bài này, *Context Engineering: Sessions, Skills & Memory*, đào sâu từng ý tưởng trên: bàn về cách thiết kế và quản lý phiên, viết và đánh giá skill, xây bộ nhớ bền vững xuyên suốt các tương tác, và tối ưu bài toán kinh tế token cho hệ thống chạy thật.

Bước chuyển từ "prompt engineering" sang "context engineering" phản ánh một sự thật sâu hơn về việc làm với AI. Model không cần những hướng dẫn được diễn đạt khôn khéo cho bằng cần đúng thứ ngữ cảnh mà một lập trình viên giỏi cũng cần để làm tốt việc. Câu hỏi không phải là "làm sao để dụ AI viết code tốt?" mà là "một thành viên mới cần biết gì để đóng góp hiệu quả, và làm sao mã hóa tri thức đó thành dạng mà AI dùng được?"

Context engineering là chiếc cầu nối giữa vibe coding và agentic engineering. Nó cũng là cầu nối giữa phần này và phần kế tiếp, nơi ta xem xét cấu trúc bao quanh mọi model và biến model thành thứ hữu dụng.

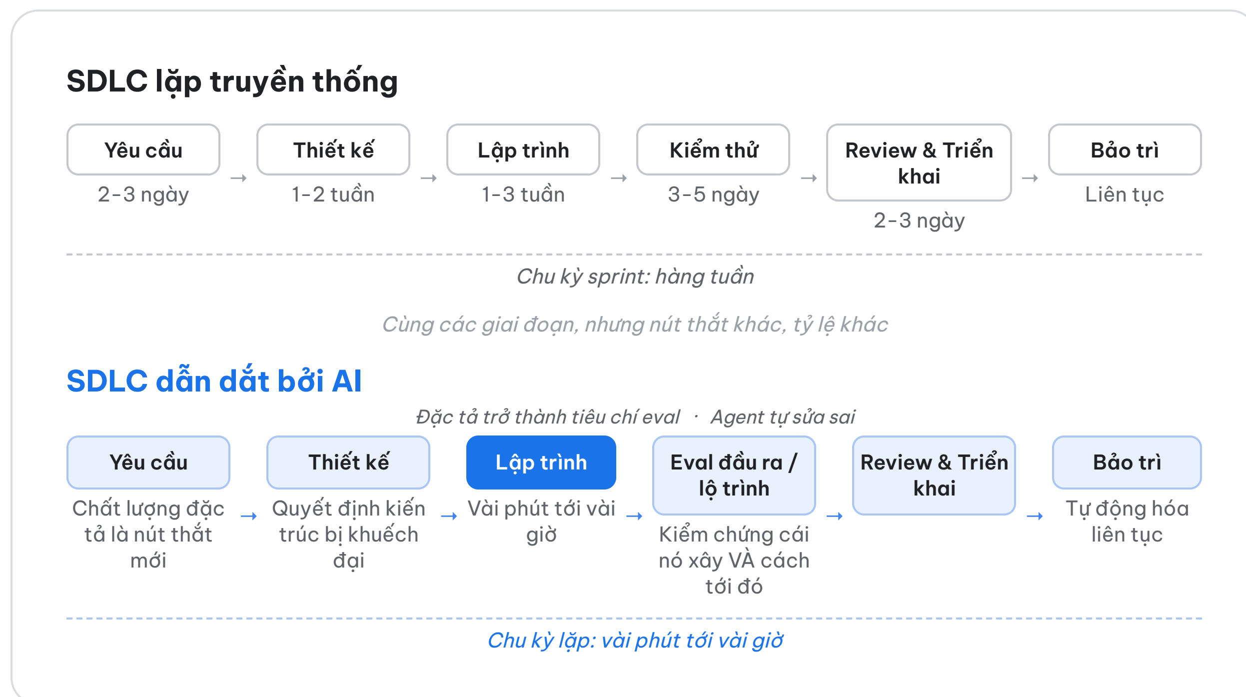
Bằng cách dời trọng tâm từ viết cú pháp sang kiến tạo ngữ cảnh này, các nút thắt trong việc tạo ra phần mềm thay đổi tận gốc. Ta không còn chờ đôi tay con người gõ những đoạn code khuôn mẫu; ta chờ trí óc con người định ra ranh giới. Điều này đòi hỏi hình dung lại hoàn toàn Vòng đời phát triển phần mềm (SDLC) truyền thống, bởi giờ đây chính các hệ thống ta dùng để xây phần mềm quyết định tốc độ giao ra phần mềm.

Vòng đời phát triển phần mềm kiểu mới

SDLC truyền thống dưới áp lực

Vòng đời phát triển phần mềm đã trải qua một cuộc chuyển đổi lớn rồi. Suốt hai thập kỷ qua, phần lớn doanh nghiệp chuyển từ quy trình thác nước tuần tự sang các mô hình lặp: sprint Agile, tích hợp liên tục, pipeline DevOps, và các chu kỳ phát hành nhanh. Bước chuyển ấy rút ngắn vòng phản hồi, kéo kiểm thử lại gần khâu phát triển, và biến triển khai thành một quá trình liên tục thay vì một sự kiện mỗi quý.

AI nén chu kỳ này lại rất mạnh, nhưng không đồng đều: phần triển khai từng mất hàng tuần nay có thể xong trong vài giờ, trong khi thu thập yêu cầu, kiến trúc và kiểm chứng vẫn cứ bám nhịp con người. Kết quả không phải là một phiên bản nhanh hơn của SDLC cũ. Đó là một quy trình khác hẳn, nơi ranh giới giữa các giai đoạn mờ đi, chu kỳ lặp rút từ hàng tuần xuống vài phút, và vai trò của lập trình viên dịch chuyển từ người triển khai chính sang người thiết kế hệ thống và người phân định chất lượng.



Hình 5: SDLC truyền thống so với SDLC dẫn dắt bởi AI.

Một lưu ý về tốc độ thay đổi: Bức tranh từng-giai-đoạn mô tả ở trên phản ánh trạng thái của SDLC dẫn dắt bởi AI tính đến giữa năm 2026. Nó đang thay đổi nhanh. Những dấu hiệu ban đầu cho thấy sự nén lại sẽ lan ra ngoài khâu triển khai: nhiều đội đã thử nghiệm các quy trình mà lập trình viên đi thẳng từ đặc tả đến review, còn agent AI lo phần triển khai, kiểm thử và đưa lên chạy ở hậu trường. Những ranh giới vẽ ra trong phần này có thể trông khác đi sau 12 tháng. Thứ không đổi sẽ là phán đoán, gu thẩm mỹ của con người, và kỹ năng kiểm chứng đầu ra của AI khi máy móc đảm nhận ngày càng nhiều phần triển khai.

AI biến đổi từng giai đoạn ra sao

Thu thập yêu cầu và lập kế hoạch

Thu thập yêu cầu là giai đoạn mà khoảng cách giữa ý định và triển khai xưa nay rộng nhất. Việc dịch nhu cầu kinh doanh thành đặc tả kỹ thuật vốn là một quá trình thủ công, dễ sai, tạo ra khoảng cách dai dẳng giữa thứ các bên liên quan muốn và thứ kỹ sư xây ra.

Các công cụ AI hiện đại có thể tham gia trực tiếp vào việc tinh chỉnh yêu cầu: sinh ra user story từ bản tóm tắt sản phẩm, phát hiện những trường hợp biên mà con người bỏ sót, tạo ra schema API từ mô tả bằng ngôn ngữ tự nhiên, và sinh ra nguyên mẫu tương tác được từ tài liệu đặc tả. Các môi trường phát triển dạng agentic cho phép lập trình viên đi từ một mô tả đến một nguyên mẫu chạy được chỉ trong vài phút, kéo vòng phản hồi từ yêu cầu đến nguyên mẫu xuống gần như bằng không.

Yêu cầu không còn là một tài liệu chuyển tay giữa các đội. Chúng trở thành một cuộc trò chuyện giữa con người và AI, tạo ra đồng thời cả đặc tả lẫn bản triển khai ban đầu.

Thiết kế và kiến trúc

Kiến trúc vẫn là giai đoạn lấy-con-người-làm-trung-tâm bền bỉ nhất của SDLC, và có lý do chính đáng. Các quyết định kiến trúc về bản chất là chuyện đánh đổi: nhất quán hay sẵn sàng, phức tạp hay linh hoạt, tự xây hay đi mua. Những đánh đổi này phụ thuộc vào bối cảnh kinh doanh, ràng buộc tổ chức, và những cân nhắc chiến lược dài hạn mà AI chưa thể nắm bắt trọn vẹn.

AI rất giỏi hiện thực các quyết định kiến trúc một khi chúng đã được chốt. Cho một tài liệu kiến trúc rõ ràng, agent AI có thể dựng khung cho cả ứng dụng, sinh ra các mẫu hình nhất quán xuyên suốt các module, và đảm bảo code mới tuân theo quy ước đã định. Vai trò của lập trình viên dịch chuyển từ viết code khuôn mẫu sang đưa ra và ghi lại những quyết định cấu trúc mà code khuôn mẫu ấy hiện thực.

Lập trình

Coding agent hiện đại có thể sinh ra cả một tính năng từ mô tả bằng ngôn ngữ tự nhiên, hiện thực các thuật toán phức tạp, và tạo ra những thay đổi nhiều file ăn khớp với nhau. Lợi ích về năng suất là có thật: các khảo sát ngành báo cáo mức tăng năng suất 25 đến 39%, một số tác vụ còn tăng mạnh hơn.[7]

Bức tranh tinh tế hơn những con số giật tít. Một nghiên cứu của METR phát hiện rằng các lập trình viên giàu kinh nghiệm khi dùng trợ lý AI thực ra mất thêm 19% thời gian ở một số tác vụ, phần lớn vì thời gian bỏ ra để kiểm chứng, gỡ lỗi và chỉnh sửa đầu ra của AI.[8] AI không xóa bỏ công việc lập trình cho bằng biến đổi nó từ viết thành review, dẫn dắt và kiểm chứng.

Kiểm thử và đảm bảo chất lượng

Kiểm thử code do AI sinh ra đòi hỏi đánh giá không chỉ thứ agent tạo ra, mà cả cách nó đi tới đó. Đánh giá đầu ra (output evaluation) xem xét sản phẩm cuối: code có biên dịch được không, test có pass không? Đánh giá lộ trình (trajectory evaluation) xem xét toàn bộ chuỗi lệnh gọi tool và suy luận trung gian. Cả hai đều cần, vì một đầu ra trôi chảy nhưng bỏ qua các bước kiểm chứng là dạng lỗi nguy hiểm hơn cả một đầu ra có lỗi lộ rõ.

AI cũng biến đổi chính việc sinh test. Agent có thể tạo ra các ca kiểm thử, gồm cả trường hợp biên và test dựa trên thuộc tính (property-based test), mà con người có thể không nghĩ ra. Quan trọng hơn, test và eval trở thành cơ chế chính để truyền đạt ý định cho agent AI: một bộ eval viết tốt cho AI biết "đúng" nghĩa là gì và đưa ra cách tự động để kiểm chứng điều đó.

Những cách làm này hiệu quả nhất khi được nối vào một bánh đà chất lượng liên tục: đánh giá trên một bộ benchmark, chẩn đoán các thất bại bằng cách gom nhóm căn nguyên, tối ưu những prompt hay tool đã gây ra lỗi, kiểm chứng bản sửa trên một bộ test hồi quy, và giám sát lưu lượng chạy thật để bắt các dạng lỗi mới. Mỗi vòng lặp lại bồi đắp thêm.

Review code và triển khai

Bản thân quá trình review cũng đang được tăng cường, với AI đóng vai người review vòng đầu, có thể phát hiện lỗi tiềm ẩn, vi phạm phong cách, lỗ hổng bảo mật và vấn đề hiệu năng trước khi người review nhìn vào code. Việc này không thay thế review của con người, vì những quyết định phụ thuộc bối cảnh về thiết kế, khả năng bảo trì và sự ăn khớp chiến lược vẫn cần con người phán đoán, nhưng nó giảm đáng kể gánh nặng tư duy cho người review.

Pipeline triển khai cũng đang dần "hiểu AI". Agent AI có thể giám sát sức khỏe của bản triển khai, tự động rollback những bản phát hành có vấn đề, và dự đoán rủi ro triển khai dựa trên bản chất và phạm vi thay đổi. Các nền tảng triển khai hiện đại ngày càng tích hợp với khả năng quan sát chạy bằng AI để tạo vòng phản hồi giữa hành vi lúc chạy thật và các quyết định phát triển.

Ngày 5 trong loạt bài này bàn về những gì thay đổi với người review khi lượng pull request tăng theo đầu ra của agent - các bản tóm tắt gộp, LGTM có điều kiện, và các skill review code do agent dẫn dắt.

Bảo trì và tiến hóa

Có lẽ cuộc biến đổi bị xem nhẹ nhất nằm ở khâu bảo trì. Những codebase cũ kỹ từng là bất khả xâm phạm với thành viên mới nay có thể được dò đường, hiểu và sửa đổi nhờ AI hỗ trợ. Một agent AI có thể đọc một codebase, hiểu các mẫu hình của nó, xác định những file liên quan cho một thay đổi, và hiện thực các sửa đổi trong khi vẫn tôn trọng kiến trúc sẵn có.

Điều này có ý nghĩa lớn với nợ kỹ thuật. Code từng bị coi là "quá rủi ro để đụng vào" vì chỉ tác giả gốc mới hiểu nay có thể được tái cấu trúc, hiện đại hóa và mở rộng một cách an toàn. Agent AI có thể di trú codebase giữa các framework một cách hệ thống, cập nhật các API đã lỗi thời, và hiện đại hóa các bộ test - những việc trước đây nhàm chán và rủi ro đến mức gần như chẳng bao giờ được làm.

Mô hình nhà máy: xây hệ thống tạo ra phần mềm

Mô hình tư duy gắn kết các biến đổi này lại với nhau là thứ chúng tôi gọi là mô hình nhà máy. Trong mô hình này, đầu ra chính của lập trình viên không phải là code - mà là hệ thống tạo ra code. Hệ thống này gồm:[8]

- Đặc tả và ngữ cảnh định ra cái cần xây
- Agent dịch đặc tả thành bản triển khai
- Test và cổng chất lượng kiểm chứng tính đúng đắn
- Vòng phản hồi đưa các thất bại quay lại agent để sửa
- Rào chắn giới hạn agent trong hành vi an toàn, dự đoán được

Một quản đốc nhà máy không tự tay lắp ráp từng món hàng. Họ thiết kế dây chuyền và đảm bảo kiểm soát chất lượng. Lập trình viên hiện đại thiết kế hệ thống phát triển và đảm bảo đầu ra của nó đạt chuẩn yêu cầu. Thành công đến từ việc trao cho agent các tiêu chí thành công thay vì hướng dẫn từng bước, rồi để chúng tự lặp.

Hình 6: Mô hình nhà máy - Lập trình viên thiết kế hệ thống → agent tạo ra code → test kiểm chứng đầu ra.

Điều này dẫn tới câu hỏi chi phối phần còn lại của tài liệu: cỗ máy trung tâm trong nhà máy là gì? Bản thân agent - thứ làm việc bên trong dây chuyền - thực sự trông như thế nào?

Nếu lập trình viên là quản đốc nhà máy, thì model AI chỉ là cỗ động cơ thô trên sàn nhà máy. Một động cơ đơn độc không thể sản xuất ra một chiếc xe; nó cần dây curoa, bánh răng, cảm biến an toàn và một dây chuyền lắp ráp. Trong bối cảnh phát triển có AI hỗ trợ, bộ máy bao quanh ấy được gọi là Harness.

Kỹ thuật Harness: thứ bao quanh model

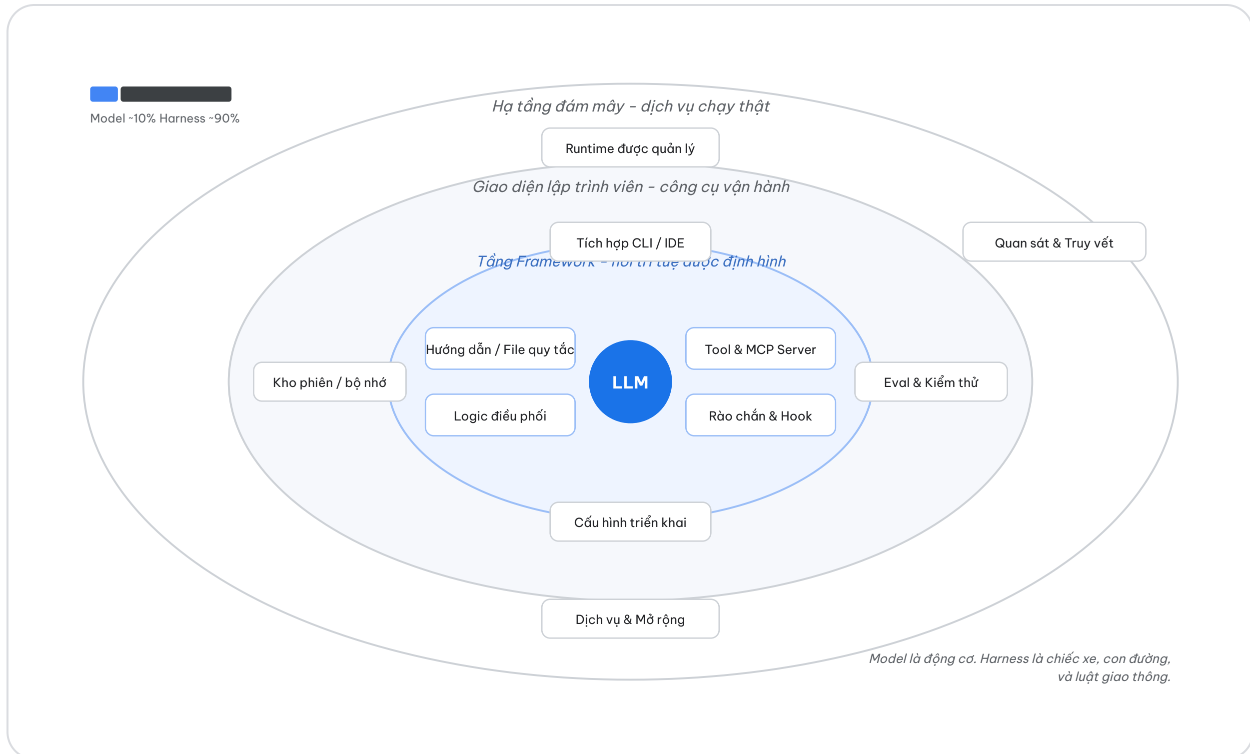
Có một cám dỗ, khi người ta bắt đầu làm việc với agent AI, là coi model chính là cả hệ thống. Model mới ra, agent thông minh hơn. Model cũ hơn thì agent kém đi. Model trở thành lời giải thích cho mọi điều tốt lẫn xấu.

Trực giác ấy sai, và nó dẫn tới những khoản đầu tư sai chỗ. Model chỉ là một đầu vào cho một agent đang chạy. Mọi thứ còn lại - prompt, tool, chính sách ngữ cảnh, hook, sandbox, sub-agent, khả năng quan sát - mới là harness: bộ khung dựng quanh model để nó thực sự hoàn thành được việc gì đó.[11]

Một phương trình hữu ích:

Agent = Model + Harness

Một model thô không phải là một agent. Nó trở thành agent một khi một harness trao cho nó trạng thái, khả năng thực thi tool, các vòng phản hồi, và những ràng buộc cưỡng chế được. Hành vi mà lập trình viên cảm nhận khi làm việc với Claude Code, Cursor, Codex, Antigravity, Aider hay Cline phần lớn do harness quyết định, chứ không chỉ do model bên dưới là cái nào.



Hình 7: Giải phẫu Harness | Agent = Model + Harness.

Trong harness có gì

Cụ thể, một harness gồm:

- **Hướng dẫn và file quy tắc:** Phần văn bản định ra agent là ai, nó quan tâm điều gì, và nó bị cấm làm gì. Bao gồm AGENTS.md, CLAUDE.md, GEMINI.md, các file skill, và prompt của sub-agent.
- **Tool:** Các hàm, MCP server và API mà agent có thể gọi, cùng phần mô tả xung quanh chúng để cho model biết khi nào và gọi chúng ra sao.
- **Sandbox và môi trường thực thi:** Nơi code của agent thực sự chạy, nó được truy cập gì, và không với tới được gì.
- **Logic điều phối:** Việc sinh sub-agent, định tuyến model, bàn giao giữa các chuyên gia, và các quy tắc chi phối khi nào mỗi thứ được kích hoạt.
- **Rào chắn hoặc Hook:** Code tất định chạy tại những điểm cụ thể trong vòng đời: trước một lệnh gọi tool, sau một lần sửa file, trước một commit. Hook là nơi dành cho những thứ agent đáng lẽ không bao giờ được quên nhưng lại hay quên.
- **Khả năng quan sát (Observability):** Log, trace, eval, đo lường chi phí và độ trễ. Thiếu khả năng quan sát thì không có cách nào biết agent đang làm tốt hay đang âm thầm trôi dạt.

Nếu nghe có vẻ nhiều thứ phải lo, thì đúng là vậy. Và đó là phần việc của đội bạn, không phải của nhà cung cấp model.

Harness trong SDLC

Trong khi bản thân model quyết định cách hoàn thành một nhiệm vụ, thì harness là bộ khung cung cấp quyền truy cập tới các tool, sandbox và sự điều phối cần thiết để thực hiện nó. Vì vậy, harness này phải hiện diện trong mọi giai đoạn nơi một agent AI hoạt động.

Đây là cách harness chủ động vận hành xuyên suốt các giai đoạn khác nhau của SDLC kiểu mới:

1. Yêu cầu, lập kế hoạch và kiến trúc (Cấu hình harness)

Đây là nơi harness được cấu hình và hiệu chỉnh. Trước khi AI viết bất kỳ code chạy thật nào, lập trình viên phải thiết lập môi trường cho agent.

- **Cấu hình harness:** Cung cấp các file hướng dẫn và quy tắc (vd tạo AGENTS.md và định ra các ràng buộc kiến trúc) mà harness sẽ nạp và cho model dùng.
- **Hành động:** Lập trình viên định ra những tool mà agent được truy cập (như các API hay schema cơ sở dữ liệu cụ thể) và đặt ra những quy tắc nền tảng mà agent không được phá vỡ.

2. Triển khai (Vận hành harness)

Trong lúc lập trình tích cực, harness đóng vai ranh giới giữ cho model AI luôn tập trung, an toàn và làm việc hiệu quả.

- **Thành phần harness được dùng:** Sandbox, môi trường thực thi, và tool.

- **Hành động:** Khi model sinh code, nó thực thi code đó bên trong sandbox cô lập của harness. Nếu model cần đọc một file hay tìm trên web, nó dùng những tool do harness cung cấp.

3. Kiểm thử và QA (Vòng phản hồi)

Kiểm thử trong quy trình agentic dựa rất nhiều vào harness để tạo điều kiện cho việc tự sửa sai một cách tự chủ.

- **Thành phần harness được dùng:** Logic điều phối và rào chắn.
- **Hành động:** Khi agent viết một hàm, harness cung cấp môi trường thực thi (chẳng hạn một terminal trong sandbox) cho phép chạy các test tự động. Nếu một test thất bại, logic điều phối thu lấy thông báo lỗi từ môi trường đó và đẩy ngược về model, yêu cầu nó thử lại. Chính harness tạo ra vòng lặp tự động "suy nghĩ → hành động → quan sát" này.

4. Review code, triển khai và bảo trì (Quan sát harness)

Ngay cả sau khi code đã viết xong, harness vẫn đảm bảo agent hành xử an toàn trong môi trường chạy thật hoặc gần như thật.

- **Thành phần harness được dùng:** Hook và khả năng quan sát.
- **Hành động:** Harness chạy các hook tất định (vd chặn một commit nếu agent định đẩy lên một mật khẩu hard-code). Hơn nữa, tầng quan sát theo dõi chi phí token, độ trễ và độ trôi dạt của agent, cho phép kỹ sư con người truy xét chính xác vì sao một agent đưa ra một quyết định triển khai cụ thể.

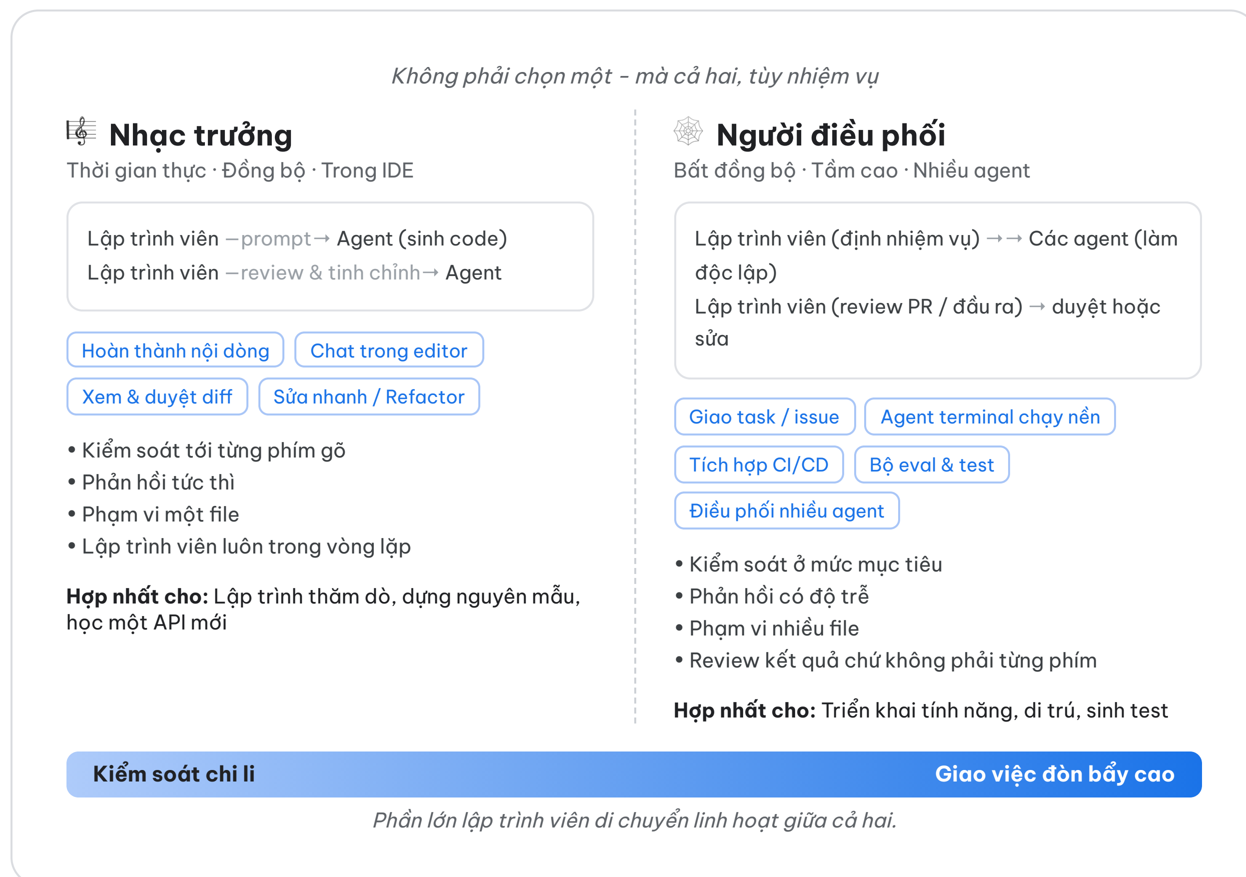
Bước chuyển từ "vibe coding" sang "agentic engineering" không đơn thuần là chuyện công cụ bạn dùng – một lập trình viên có thể vibe code hoặc áp dụng agentic engineering bằng đúng cùng một agent. Thay vào đó, nó được định nghĩa bởi mức độ chủ đích khi bạn cấu hình và vận dụng harness. Vibe coding dựa vào bộ khung tối thiểu hoặc ngầm định, chỉ nhắm tới triển khai thật nhanh. Agentic engineering dựa vào những lớp trừu tượng harness rõ ràng, bài bản, dẫn dắt AI từ tài liệu lập kế hoạch đầu tiên cho tới tận khâu giám sát lúc chạy thật.

Tác động của việc cấu hình có chủ đích này đo lường được rõ ràng. Các benchmark công khai cho thấy quy mô của hiệu ứng harness một cách cụ thể. Trên Terminal Bench 2.0, một đội đã đưa một coding agent từ ngoài Top 30 lên Top 5 chỉ bằng cách thay đổi harness, hoàn toàn không đổi model. Một nghiên cứu riêng tại LangChain nâng điểm của một coding agent trên cùng benchmark đó thêm 13,7 điểm chỉ bằng cách tinh chỉnh system prompt, tool và middleware quanh một model cố định.

Phiên bản đời thường của quan sát này rất quan trọng với các đội đang đưa AI vào khắp SDLC: khi một agent làm sai điều gì đó, phản xạ đầu tiên là đổ lỗi cho model. Nhưng thường thì thất bại lần ngược về một tool còn thiếu, một quy tắc mơ hồ, một rào chắn vắng mặt, hay một cửa sổ ngữ cảnh nhồi đầy nhiễu. Phần lớn thất bại của agent, nếu xét cho thành thật, là thất bại về cấu hình.

Vai trò đang đổi của lập trình viên: nhạc trưởng và người điều phối

Khi AI đảm nhận ngày càng nhiều phần triển khai, vai trò của lập trình viên đang biến đổi theo những cách vừa hào hứng vừa hoang mang. Chúng tôi thấy hữu ích khi nghĩ về hai chế độ mà lập trình viên chuyển qua lại linh hoạt: nhạc trưởng và người điều phối.[12]



Hình 8: Nhạc trưởng và Người điều phối (Hai chế độ làm việc với agent AI).

Nhạc trưởng: trực tiếp, chỉ huy theo thời gian thực

Ở chế độ nhạc trưởng, lập trình viên làm việc theo thời gian thực với một AI đóng vai bạn lập trình cặp. Họ ở trong IDE, nhìn code hiện ra, dẫn dắt AI bằng prompt và chỉnh sửa, và giữ quyền kiểm soát chi tiết với những gì được viết ra. AI là một nhạc cụ mạnh mẽ, nhưng lập trình viên mới là người chủ động điều khiển từng chuyển động.

Chế độ này điển hình khi xử lý logic phức tạp, gỡ những lỗi hóc búa, hoặc làm trong những codebase lạ nơi lập trình viên cần hiểu từng thay đổi ngay khi nó được tạo ra. Các công cụ như GitHub Copilot, Gemini Code Assist của Google, Cursor và Windsurf chủ yếu hỗ trợ chế độ này qua hoàn thành nội dòng, giao diện chat, và khả năng sửa tại chỗ.

Chế độ nhạc trưởng tự nhiên với những lập trình viên xuất thân từ nền kỹ thuật truyền thống. Nó giữ lại cảm giác thấu hiểu và kiểm soát mà nhiều kỹ sư trân trọng. Rủi ro là nó cũng có thể trở thành nút thắt - nếu lập trình viên đích thân chỉ đạo từng phím gõ, thì mức cải thiện thông lượng từ AI sẽ bị giới hạn.

Người điều phối: bất đồng bộ, giao việc cho nhiều agent

Ở chế độ người điều phối, lập trình viên làm việc ở mức trừu tượng cao hơn. Họ định ra mục tiêu, giao cho các agent, và review kết quả - chứ không ngồi nhìn code hiện ra từng dòng. Các agent có thể đang làm việc ở hậu trường, song song, trên những phần khác nhau của codebase. Lập trình viên ghé kiểm tra định kỳ, review đầu ra, và đưa ra điều chỉnh hướng đi.

Chế độ này điển hình với những nhiệm vụ đã được định rõ như sửa lỗi, triển khai tính năng theo mẫu hình có sẵn, di trú codebase, và sinh test. Các công cụ như Jules của Google, chế độ agent của GitHub Copilot, các agent chạy nền của Cursor, và Claude Code hỗ trợ chế độ này qua việc thực thi tác vụ bất đồng bộ, thường làm trong môi trường sandbox với toàn quyền truy cập repository, công cụ build và bộ test.[13]

Chế độ người điều phối đòi hỏi một bộ kỹ năng khác. Thay vì chuyên môn sâu về cú pháp và lối hành văn của ngôn ngữ, nó đòi hỏi kỹ năng mạnh ở:

- **Đặc tả:** Định nghĩa nhiệm vụ đủ chính xác để agent thực thi mà không mơ hồ

- **Phân rã:** Chia nhiệm vụ lớn thành những đơn vị có kích cỡ phù hợp cho agent thực thi
- **Đánh giá:** Nhanh chóng thẩm định xem đầu ra của agent có đạt chuẩn chất lượng không
- **Thiết kế hệ thống:** Thiết kế các ràng buộc, test và vòng phản hồi giữ cho agent làm việc hiệu quả

Bài toán 80%

Một thách thức dai dẳng trong phát triển có AI hỗ trợ là thứ chúng tôi gọi là bài toán 80%: agent AI có thể sinh nhanh khoảng 80% code cho một tính năng, nhưng 20% còn lại - các trường hợp biên, xử lý lỗi, điểm tích hợp, và những yêu cầu đúng đắn tinh tế - đòi hỏi tri thức bối cảnh sâu mà các model hiện tại thường thiếu.[14]

Bản chất lỗi của AI đã tiến hóa từ những lỗi cú pháp đơn giản sang những thất bại khái niệm âm hiểm hơn: giả định sai về logic nghiệp vụ, không chịu hỏi lại khi yêu cầu mơ hồ, bỏ sót trường hợp biên, và những quyết định kiến trúc tạo ra gánh nặng bảo trì tinh vi về lâu dài. Những lỗi này khó phát hiện hơn chính vì code "trông có vẻ đúng" và thậm chí có thể vượt qua các test cơ bản.

Những lập trình viên vượt qua thách thức này hiệu quả nhất có một tư thế cụ thể: họ dùng AI cho thứ nó giỏi (triển khai nhanh các nhiệm vụ đã đặc tả kỹ) trong khi dành sự chú ý của mình cho thứ AI chật vật (yêu cầu mơ hồ, đánh đổi kiến trúc, và kiểm chứng tính đúng đắn). Họ không cố nhanh hơn bằng cách chấp nhận mọi thứ AI tạo ra. Họ cố nhanh hơn bằng cách dồn chuyên môn vào nơi quan trọng nhất.

Vượt qua bài toán 80% này một cách hiệu quả đòi hỏi áp đúng công cụ cho đúng giai đoạn công việc. Một lập trình viên đang ở vai "Nhạc trưởng" cần một bộ công cụ khác với người đang ở vai "Người điều phối". Để hiểu cách ánh xạ các chế độ vận hành này vào quy trình hằng ngày của bạn, ta cần phân loại bức tranh hiện tại của các agent AI dựa trên mức tự chủ và mức tích hợp của chúng.

Coding agent trong thực tế

Một lập trình viên xây agent ngày nay làm phần lớn công việc từ terminal, thường bằng ngôn ngữ tự nhiên, thường có một coding agent khác gõ giúp. Đây là điều mới. Một năm trước, cùng nhiệm vụ ấy có nghĩa là framework, SDK và các console đám mây. Những mẫu hình đã thay thế chúng đáng được gọi tên rõ ràng, cả cho lập trình viên muốn dùng coding agent trong ngày làm việc, lẫn cho lập trình viên muốn tự xây agent của riêng mình.

Coding agent xuất hiện ở đâu trong ngày làm việc

Coding agent xuất hiện ở ba chỗ trong công việc hằng ngày. Phần lớn lập trình viên dùng cả ba cùng lúc.

Trong trình soạn thảo: Hoàn thành nội dòng gợi ý dòng kế tiếp ngay khi lập trình viên gõ. Các bảng chat giải thích hoặc sửa code tại chỗ. Khả năng nhận biết toàn bộ codebase ngay trong IDE. Đây là nơi phần lớn người ta lần đầu gặp AI trong lập trình, và là nơi công việc giữ được mạch trôi. Ví dụ gồm GitHub Copilot, Cursor, Windsurf, JetBrains AI Assistant.

Trong terminal: Những coding agent mà lập trình viên khởi chạy từ dòng lệnh, giao cho một mục tiêu bằng lời lẽ thông thường, rồi để nó làm việc khắp codebase. Toàn quyền truy cập hệ thống file, sửa nhiều file, khả năng chạy tool và test rồi lặp lại dựa trên kết quả. Đây là nơi việc vibe coding nghiêm túc đang diễn ra hôm nay. Ví dụ gồm Antigravity CLI, Claude Code, Codex CLI, Open Code, và Cline.

Ở hậu trường: Những agent nhận một nhiệm vụ và chạy tự chủ trong các sandbox đặt trên đám mây, thường hàng giờ, thường tạo ra một pull request làm đầu ra. Lập trình viên bàn giao rồi review sau. Ví dụ gồm Google Jules, chế độ agent của GitHub Copilot, các agent chạy nền của Cursor, và agent chuyên biệt AlphaEvolve của Google dùng để thiết kế các thuật toán nâng cao.

Trong thực tế, một agent trong trình soạn thảo giúp ích khi lập trình viên đang giữa lúc viết code và muốn gợi ý, sửa nhanh, hoặc giải thích mà không rời mạch. Một agent terminal hợp với công việc nhiều file, thăm dò những codebase lạ, và những nhiệm vụ mà agent cần chạy code rồi phản ứng theo điều nó quan sát. Một agent hậu trường hợp với những nhiệm vụ đã đặc tả kỹ mà lập trình viên có thể mô tả trong một đoạn văn rồi bỏ đi, như sửa một lỗi đã biết, sinh một bộ test, hoặc di trú code từ framework này sang framework khác. Cùng một lập trình viên thường dùng cả ba trong một ngày.

Điểm khởi đầu phù hợp tùy vào nhiệm vụ, chứ không phải vào việc nhóm nào đứng cao nhất trên một cái thang tự chủ nào đó.

Vibe coding để dựng agent sẵn sàng chạy thật

Mọi thứ bàn tới nãy giờ đều xoay quanh việc dùng coding agent để xây phần mềm: viết tính năng, sửa lỗi, sinh test, tái cấu trúc code. Nhưng chuyện gì xảy ra khi thứ bạn cần xây bản thân nó là một agent?

Một bot hỗ trợ khách hàng xử lý yêu cầu hoàn tiền. Một trợ lý nghiên cứu đối chiếu chéo nguồn và tạo ra các báo cáo có căn cứ. Một công cụ nội bộ giám sát tuân thủ và gắn cờ những bất thường. Đây không phải những nhiệm vụ bạn giải bằng một coding agent trong terminal. Chúng là những sản phẩm cần có tool riêng, bộ nhớ riêng, đánh giá riêng, và hạ tầng triển khai riêng.

Chính quy trình dựa trên terminal vốn tạo ra các script nguyên mẫu nay vươn tới được những agent chạy thật này. Việc xây, đánh giá và triển khai một agent thật chạy ở quy mô lớn, với bộ nhớ bền vững, quản trị và khả năng quan sát, đã chuyển từ một công việc cần framework và console đám mây thành một việc diễn ra ngay trong cùng cái terminal, thường chỉ bằng cách trò chuyện với chính coding agent mà lập trình viên vốn đang dùng.

Quy trình này quan trọng khi người xây cần một agent chạy ổn định cho người dùng thật: bộ nhớ bền vững xuyên suốt các phiên, quyền hạn giới hạn theo phạm vi trên tool và dữ liệu, độ phủ eval bắt được hồi quy trước khi phát hành, khả năng quan sát truy vết được agent thực sự đã làm gì. Với những script dùng một lần hay tự động hóa cá nhân, một coding agent thông thường là đủ; bản thân agent là đích đến. Với những agent phục vụ người dùng thật ở quy mô lớn, agent là sản phẩm, và nó cần một nền tảng bên dưới.

Agents CLI của Google được xây quanh ý tưởng này.[14] Đó là một công cụ dòng lệnh nhỏ gói sẵn một bộ skill để xây agent trên Google Cloud, và quan trọng là nó hoạt động với bất kỳ coding agent nào lập trình viên thích, Claude Code, Codex, hay cái khác. Sau một lần cài đặt, coding agent có thêm bảy skill mới phủ trọn vòng đời ADK: dựng khung dự án, viết code agent, đánh giá nó, triển khai nó lên Agent Runtime, và nối dây khả năng quan sát. Lập trình viên không phải học một SDK mới. Họ mô tả thứ mình muốn, và coding agent dùng các skill để làm đúng việc ở mỗi bước.

Cụ thể, toàn bộ vòng lặp xây-đánh giá-triển khai trông như thế này:

```
# Cài đặt một lần
uvx google-agents-cli setup
# Rồi trong coding agent của bạn:
> Xây một support agent trả lời câu hỏi từ tài liệu của chúng ta.
> đánh giá nó trên bộ dữ liệu FAQ
> Triển khai nó lên Agent Engine
```

Đoạn lệnh 1: Cài đặt và xây dựng với Agents CLI.

Đằng sau một câu lệnh duy nhất đó, coding agent dựng khung dự án từ một template, viết code ADK, sinh một evalset, chạy nó trên agent, triển khai lên Agent Runtime, và báo cáo lại. Với những lập trình viên thích tự lái trực tiếp, các thao tác tương tự cũng có sẵn dưới dạng lệnh CLI thường (`agents-cli create`, `agents-cli playground`, `agents-cli eval`, `agents-cli deploy`).

Agent chạy thật trước đây đòi hỏi một stack riêng và một quy trình riêng so với nguyên mẫu. Giờ thì nguyên mẫu chạy trên laptop của lập trình viên hôm qua có thể trở thành agent chạy thật phục vụ người dùng thật hôm nay, mà không cần viết lại.

Cùng một quy trình mở rộng được từ một agent thành nhiều agent. ADK cung cấp các quy trình dạng đồ thị, các quy trình nhiều agent để xây những agent cộng tác, và các cơ chế tương tác như trạng thái phiên dùng chung, giao việc do LLM dẫn dắt, và gọi tường minh, kết hợp lại thành bất kỳ mẫu hình nhiều-agent nào hợp với bài toán.

Sự phối hợp giữa các agent diễn ra qua trạng thái phiên dùng chung cho các trường hợp đơn giản, qua Model Context Protocol (MCP) để truy cập tool, và qua giao thức Agent2Agent (A2A) để giao việc giữa các agent.[15] Đội kỹ thuật của Anthropic đã công bố một thử nghiệm đầu năm 2026, trong đó các đội agent chạy trên kiểu kiến trúc này đã xây một trình biên dịch C chạy được bằng

Rust trong hai tuần, với con người đặt định hướng và review đầu ra nhưng không viết phần triển khai.[16] Nút thắt dời từ việc viết code sang việc đặc tả thứ nó cần làm và kiểm chứng rằng các agent đã làm đúng.

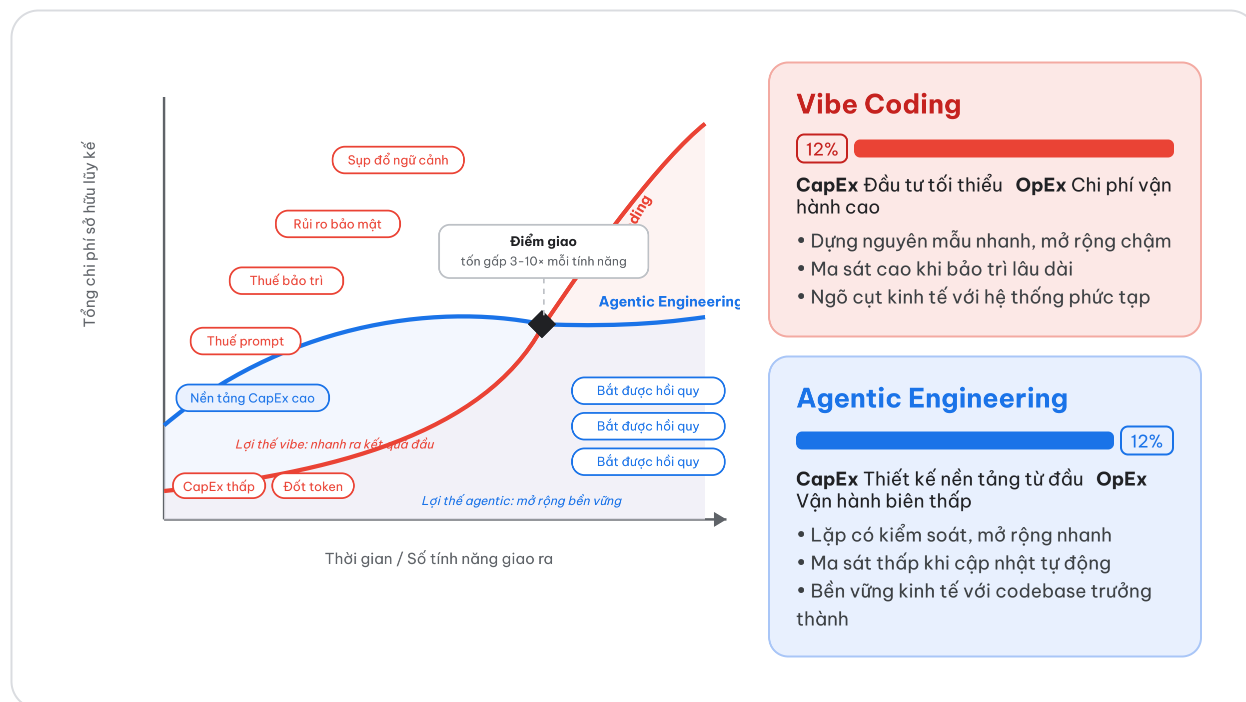
Với người xây, hệ quả thực tiễn rất đơn giản. Cùng một quy trình vibe coding tạo ra một script hôm nay sẽ tạo ra một agent chạy thật ngày mai. Vòng đời - xây, đánh giá, triển khai, quan sát, tinh chỉnh - nằm gọn ở một nơi. Con đường từ ý tưởng đến agent đang chạy đã rút từ hàng tuần xuống vài giờ, và phần lớn công việc giờ diễn ra bằng ngôn ngữ tự nhiên.

Những cách làm giúp quy trình này đạt chuẩn chạy thật ở quy mô đội - từ phát triển dựa trên đặc tả (spec-driven development) và review code có cấu trúc cho tới rào chắn, sandbox và phát triển zero-trust - được bàn trong tài liệu đồng hành Ngày 5: *Spec-Driven Production Grade Development in the Age of Vibe Coding*.

Bài toán kinh tế của phát triển phần mềm bằng AI

Khi đánh giá tác động của AI lên vòng đời phát triển phần mềm, cuộc trò chuyện thường bắt đầu và kết thúc ở tốc độ của lập trình viên: ta viết code nhanh tới đâu? Tuy nhiên, với lãnh đạo kỹ thuật, thước đo quan trọng hơn là Tổng chi phí sở hữu (TCO).

Để hiểu chi phí thật của phát triển có AI hỗ trợ, ta phải nhìn vào cách các quy trình khác nhau dịch chuyển gánh nặng tài chính và vận hành giữa Chi phí đầu tư (CapEx) - khoản bỏ ra ban đầu để xây một thứ gì đó - và Chi phí vận hành (OpEx) - khoản chi liên tục để chạy, sửa và bảo trì nó. Quan trọng là, trong thời đại AI, OpEx bị chi phối nặng nề bởi nền kinh tế token.



Hình 9: Bài toán kinh tế của phát triển phần mềm bằng AI.

Món nợ ẩn của vibe coding (CapEx thấp, OpEx cao)

Thoạt nhìn, vibe coding có vẻ rẻ một cách khó tin. Rào cản gia nhập gần như bằng không: một gói thuê bao hằng tháng tiêu chuẩn cho một trợ lý AI và vài câu prompt tùy hứng. CapEx không đáng kể vì lập trình viên dựa hoàn toàn vào năng lực nền của model thay vì bỏ thời gian thiết kế hệ thống.

Tuy nhiên, bài toán kinh tế của vibe coding giấu một gánh nặng OpEx khổng lồ, dồn tích theo thời gian:

- **Tốc độ đốt token:** Mỗi lần tương tác với một mô hình ngôn ngữ lớn (LLM) đều phát sinh chi phí dựa trên token đầu vào và đầu ra. Trong vibe coding, lập trình viên thường đổ những file khổng lồ, không cấu trúc vào cửa sổ ngữ cảnh và liên tục yêu cầu model sửa chính những lỗi chưa được kiểm chứng của nó. Việc này tạo ra một "vòng lặp prompt" tốn kém, ngốn token API với tỷ lệ thành công ngay lần đầu thấp.

- **Thuế bảo trì:** Code viết bằng kiểu ra lệnh tùy hứng thường thiếu sự nhất quán về cấu trúc. Khi một lỗi xuất hiện sáu tháng sau, kỹ sư con người phải bỏ ra hàng ngày trời để mò ngược thứ code "rối như mì" do AI sinh ra mà không có cấu trúc.
- **Khắc phục bảo mật:** Thiếu một harness đánh giá tự động, việc sinh code nhanh dẫn tới việc sinh lỗ hổng nhanh. Chi phí vá một lỗ hổng bảo mật khi đã chạy thật cao hơn theo cấp số nhân so với bắt được nó ngay từ giai đoạn thiết kế.

Khoản đầu tư của agentic engineering (CapEx cao, OpEx thấp)

Agentic engineering đảo ngược mô hình kinh tế này. Nó đòi hỏi một khoản đầu tư có chủ đích, trả trước, về thời gian và nguồn lực kỹ thuật trước khi một dòng code chạy thật nào được sinh ra.

CapEx trong agentic engineering bao gồm thiết kế schema API, xây các bộ test tất định, và quan trọng nhất là kiến tạo ngữ cảnh cho agent. Dù chi phí trả trước cao hơn, chi phí biên để giao và bảo trì một tính năng lại giảm mạnh. AI vận hành bên trong một "nhà máy" được quản trị chặt, nghĩa là đầu ra của nó vững về cấu trúc, đã được kiểm thử trước, và ăn khớp với chuẩn của công ty.

Context engineering như một đòn bẩy tài chính

Trong nền kinh tế token, context engineering không chỉ là một kỹ năng kỹ thuật - nó là một chiến lược tài chính. LLM tính tiền cho từng mẩu thông tin bạn gửi đi. Đẩy nguyên một repository 100.000 token vào mọi prompt là điều bất khả thi về tài chính khi mở rộng quy mô.

Context engineering hiệu quả đảm bảo model nhận được một gói thông tin cô đọng, đậm tín hiệu (chẳng hạn một file AGENTS.md chính xác và các rào chắn kiến trúc) thay vì một mớ lan man, nhiều nhiễu. Bằng cách cung cấp đúng ngữ cảnh từ đầu, lập trình viên tăng mạnh tỷ lệ thành công ngay lần đầu của agent, tránh được những vòng thử-và-sai tốn kém vốn đeo bám vibe coding.

Mở rộng hiệu quả nhờ ngữ cảnh động và Skill

Để thực sự tối ưu OpEx, agentic engineering nâng cao dựa vào ngữ cảnh động thông qua việc dùng "skill" hoặc gọi tool (như các Model Context Protocol server) mà chúng tôi bàn chi tiết trong tài liệu Ngày 3.

Định tuyến model thông minh

Hơn nữa, agentic engineering cho phép định tuyến model thông minh. Trong một quy trình vibe coding, lập trình viên thường dựa vào một model tiên phong duy nhất, đồ sộ cho mọi tương tác - trả giá token hạng cao chỉ để nhờ AI sửa một lỗi gõ phím hay sinh một unit test cơ bản.

Một mô hình nhà máy được thiết kế tốt tránh sự lãng phí này. Nó dùng các model lớn, tiên tiến cho những nhiệm vụ rất phức tạp (Yêu cầu, Kiến trúc, và Triển khai ban đầu) nhưng tự động định tuyến những nhiệm vụ tất định, ít phức tạp hơn (Sinh test, Review code, và giám sát CI/CD) sang những model nhỏ hơn, nhanh hơn và rẻ hơn nhiều. Bằng cách điều phối một hệ sinh thái nhiều model, các đội kỹ thuật có thể giữ chất lượng đầu ra ở đỉnh cao trong khi giảm dần một cách hệ thống chi phí token vận hành.

Bắt đầu từ đâu

Bước chuyển từ cú pháp sang ý định không phải là một trạng thái tương lai. Nó là công việc trước mắt ngay hôm nay. Dù bạn đọc tài liệu này với tư cách một người xây cá nhân hay một lãnh đạo đang nghĩ về cách một đội hay một tổ chức áp dụng các công cụ này, vẫn cùng một nguyên lý nền: AI khuếch đại văn hóa kỹ thuật mà nó rơi vào. Những cách làm dưới đây biến nguyên lý đó thành hành động.

Với lập trình viên cá nhân

1. Lập một AGENTS.md (hoặc tương đương) cho dự án. Chọn quy ước khớp với coding agent bạn dùng. Bắt đầu với mười dòng: stack, quy ước, quy tắc cứng, quy trình làm việc. Thêm một quy tắc mỗi khi agent làm điều gì đó mà nó không nên lặp lại.
2. Cài một bộ skill cho coding agent của bạn (như Agents CLI) để xây, đánh giá, triển khai và tối ưu agent.
3. Chọn một quy trình lặp đi lặp lại và biến nó thành agent đầu tiên. Một quy trình nghiên cứu, một quy trình review code, một báo cáo định kỳ, một loại nội dung được tạo ra đều đặn. Dùng một coding agent cho nguyên mẫu, rồi nâng nó lên thành agent chạy thật qua Agents CLI khi nó chứng minh được giá trị. Xây trọn một agent từ đầu đến cuối dạy cho bạn nhiều hơn đọc về cả trăm cái.
4. Viết test và eval trước khi sinh code. Cùng nhau, chúng là bản hợp đồng với AI. Một bộ test và eval viết tốt truyền đạt ý định chính xác hơn bất kỳ prompt ngôn ngữ tự nhiên nào, và biến phát triển có AI hỗ trợ từ vibe coding thành agentic engineering.
5. Review từng dòng agent tạo ra mà sắp được đưa lên chạy. Hãy hoài nghi với bất cứ thứ gì trông có vẻ khôn khéo. Kiểm tra các import xem có phải package có thật không. Xác minh rằng xử lý lỗi bao quát được các tình huống hỏng thực tế. Code mà cả đội không hiểu sẽ trở thành chi phí gỡ lỗi mà đội không kham nổi.

6. Giữ cho kỹ năng lập trình của bạn luôn sắc. AI lo phần thường nhật để lập trình viên tập trung vào phần hóc búa. Sự phân vai đó chỉ hiệu quả nếu các kỹ năng nền tảng - gỡ lỗi, thiết kế hệ thống, trực giác về hiệu năng và tính đúng đắn - vẫn sắc bén. Hãy coi AI là cách áp dụng chuyên môn ở quy mô lớn hơn, chứ không phải vật thay thế cho nó. Thường xuyên luyện gỡ lỗi phức tạp, review code do AI tạo, và thảo luận kiến trúc vẫn thiết yếu để trưởng thành với tư cách một kỹ sư.

Với lãnh đạo kỹ thuật

1. Biến context engineering thành một thực hành kỹ thuật hạng nhất trong đội. Hãy coi AGENTS.md, system prompt, bộ eval và thư viện skill như code: được review trong pull request, quản lý phiên bản cùng dự án, có kỹ sư đứng tên sở hữu. Thiếu kỷ luật này, harness sẽ trôi dạt và hành vi agent trở nên không tái lập được trong cả đội.

2. Đặt thước đo ở eval, không phải ở demo. Một demo chạy được chứng minh agent có thể thành công một lần. Một bộ eval đạt chứng minh nó thành công một cách ổn định. Nhưng một eval không có thang chấm rõ ràng thì không đo được gì. Hãy định rõ bạn đang chấm cái gì: mức hoàn thành nhiệm vụ, chất lượng dùng tool, độ tuân thủ lộ trình, mức bịa đặt, và chất lượng câu trả lời. Hãy đòi hỏi độ phủ eval với thang chấm tường minh như một điều kiện tiên quyết để bất kỳ agent nào được đưa vào một quy trình dùng chung, giống như độ phủ test làm cổng cho việc triển khai một dịch vụ.

3. Định hình lại việc review code cho code do AI sinh ra. Code do AI sinh đòi hỏi sự soi xét bằng hoặc hơn code do người viết, với sự chú ý đặc biệt tới những phụ thuộc bịa đặt, xử lý lỗi sơ sài, và những lỗ hổng đúng-đắn tinh vi trông thoáng qua có vẻ ổn. Hãy huấn luyện người review về các dạng lỗi của code được sinh ra, và tinh chỉnh checklist review cho phù hợp.

4. Phân biệt việc dựng nguyên mẫu với việc làm sản phẩm chạy thật trong chuẩn mực của đội. Vibe coding là tốc độ đúng cho việc thăm dò. Agentic engineering là kỷ luật đúng cho sản phẩm chạy thật. Hãy làm rõ ranh giới: dự án nào, nhánh nào, môi trường nào xứng với chế độ làm việc nào. Những đội để ranh giới này mập mờ sẽ tạo ra những nguyên mẫu vô tình bị đẩy lên chạy thật.

5. Đầu tư vào các thành phần harness như một tài sản chung của đội. System prompt tái dùng được, thư viện skill, kết nối MCP server, và các harness đánh giá đều bồi đắp giá trị qua nhiều dự án. Hãy coi chúng là hạ tầng: được ghi tài liệu, được bảo trì, và được cải thiện có chủ đích. Những đội tích lũy giá trị nhiều nhất từ phát triển có AI hỗ trợ là những đội xây harness một lần rồi tinh chỉnh nó nhiều lần.

Với tổ chức

1. Coi phát triển có AI hỗ trợ là một khoản đầu tư kỹ thuật, không phải một tính năng năng suất. Những đội thấy lợi ích lớn nhất là những đội ghép công cụ AI với độ phủ eval, khả năng quan sát, và chuẩn kiến trúc rõ ràng. Triển khai một coding agent mà thiếu bộ khung đó sẽ tạo ra tốc độ mà không có chất lượng, và nó dồn thành nợ kỹ thuật nhanh hơn tốc độ bất kỳ đội nào trả nổi.

2. Đầu tư vào nền tảng chạy thật trước khi mở rộng. Một nguyên mẫu vibe code trên laptop không phải là một hệ thống chạy thật. Thứ nâng cái này lên cái kia là kỷ luật vận hành quanh nó: eval lộ trình và câu trả lời cuối chạy trong CI, trace của mọi lần chạy agent, quyền hạn giới hạn theo phạm vi cho từng agent, và review bảo mật được tinh chỉnh cho các dạng lỗi của code được sinh ra. Hãy xây nền tảng này trước khi agent chạy thật đầu tiên phát hành, chứ không phải sau.

3. Áp dụng các chuẩn mở cho tool và giao tiếp giữa các agent. Model Context Protocol (MCP) để truy cập tool và Agent2Agent (A2A) để giao việc giữa các agent đang hội tụ thành mô liên kết của các hệ thống nhiều agent. Chọn chúng ngay bây giờ giữ cho lựa chọn trộn nhà cung cấp và framework luôn mở, và tránh phải đổi nền tảng về sau.

4. Lên kế hoạch cho những đội lai giữa con người và agent, chứ không phải quy trình chỉ-con-người hay chỉ-agent. Những kết quả chạy thật mạnh nhất năm qua đến từ những kiến trúc nơi con người đặt định hướng, agent lo phần triển khai, và các giao thức bàn giao rõ ràng quản lý ranh giới. Quy trình review code, lịch trực, và cơ cấu đội đều cần tiến hóa để phản ánh rằng agent giờ là một bên tham gia, không chỉ là công cụ.

5. Định hình lại việc tuyển dụng và phát triển kỹ năng quanh phán đoán, không chỉ triển khai. Khi triển khai trở nên nhanh hơn và tự động hơn, nút thắt dời sang đặc tả, đánh giá, phán đoán kiến trúc, và review. Hãy tuyển và phát triển những kỹ năng đó một cách có chủ đích. Những kỹ sư giá trị nhất trong vài năm tới sẽ là người chỉ huy agent giỏi, chứ không phải người viết được nhiều code nhất.

Kết luận: Ý định là giao diện mới

Bước chuyển từ cú pháp sang ý định không phải một dự đoán tương lai - nó là một thực tại hiện tại. Lập trình viên đã dành nhiều thời gian mô tả thứ mình muốn hơn là đặc tả cách xây nó. SDLC đã đang bị nén lại, tái cấu trúc, và hình dung lại quanh năng lực AI. Câu hỏi không phải là cuộc chuyển đổi này có xảy ra hay không, mà là các lập trình viên cá nhân, các đội và các tổ chức sẽ điều hướng nó hiệu quả tới đâu.

Khung tư duy chúng tôi trình bày trong tài liệu này - dải quang phổ từ vibe coding đến agentic engineering, mô hình vai trò lập trình viên từ nhạc trưởng đến người điều phối, cách phân loại agent theo kiểu ngầm nền, theo quy trình và tự hành, và mô hình nhà máy của việc sản xuất phần mềm - cung cấp một bộ mô hình tư duy để hiểu một bức tranh đang đổi thay nhanh chóng. Những mô hình này vẫn sẽ hữu ích ngay cả khi các công cụ và năng lực cụ thể thay đổi.

Ba nguyên tắc nổi lên như những điều bền vững:

1. **Cấu trúc thì mở rộng được, cảm hứng thì không.** Vibe coding là một cách tiếp cận hợp lệ cho thăm dò, dựng nguyên mẫu, và dự án cá nhân. Nhưng với phần mềm mà các tổ chức dựa vào, kỷ luật của agentic engineering - đặc tả, test, rào chắn, và sự giám sát của con người với kiến trúc - là không thể bỏ qua. Khoảng cách giữa "trông có vẻ chạy" và "chạy đúng trong mọi điều kiện" chính là nơi trú ngụ của những sự cố chạy thật, lỗ hổng bảo mật, và những cơn ác mộng bảo trì.

2. **AI khuếch đại văn hóa kỹ thuật của bạn.** Những tổ chức có thực hành kiểm thử vững, chuẩn kiến trúc rõ ràng, và quy trình review code lành mạnh thu được giá trị từ phát triển có AI hỗ trợ nhiều hơn hẳn những tổ chức không có. AI là một cấp số nhân lực - và nó nhân lên cả điểm mạnh lẫn điểm yếu của bạn.

3. **Vai trò con người đang tiến hóa, không teo đi.** Những người xây hiểu kiến trúc, định được đặc tả chính xác, đánh giá đầu ra một cách phê phán, và thiết kế được những hệ thống ràng buộc và vòng phản hồi hiệu quả thì giá trị hơn bao giờ hết. Những kỹ năng quan trọng đang dịch từ triển khai sang phán đoán, từ viết code sang thiết kế những hệ thống tạo ra code.

Chúng ta đang ở khởi đầu của một cuộc chuyển đổi sẽ định hình lại không chỉ cách phần mềm được xây, mà cả loại phần mềm nào có thể xây được. Những đội nhỏ hơn sẽ giải quyết được những bài toán lớn hơn. Lập trình viên cá nhân sẽ xây và bảo trì được những hệ thống trước đây đòi hỏi cả một phòng ban. Rào cản để tạo ra phần mềm sẽ tiếp tục hạ xuống, mở ngành phát triển phần mềm cho một lượng người rộng hơn.

Những đội phát đạt sẽ là những đội đón nhận AI như một công cụ mạnh mẽ trong khi vẫn giữ kỷ luật kỹ thuật vốn luôn là nền tảng của phần mềm đáng tin cậy. Họ sẽ là những người hiểu rằng tương lai của kỹ thuật phần mềm không phải là chọn giữa chuyên môn con người và năng lực AI - mà là thiết kế những hệ thống nơi cả hai cùng đóng góp thế mạnh riêng.

Sinh code thì đã giải quyết xong. Kiểm chứng, phán đoán và định hướng mới là nghề mới.

Ghi chú nguồn

1. GetPanto, "AI Coding Assistant Statistics 2025-2026," https://www.getpanto.ai/blog/ai-coding-assistant-statistics; Index.dev, "Developer Productivity Statistics with AI Tools," https://www.index.dev/blog/developer-productivity-statistics-with-ai-tools
2. Karpathy, A., "Vibe Coding," bài đăng trên X/Twitter, tháng 2 năm 2025. https://x.com/karpathy/status/1886192184808149383; Wikipedia, "Vibe coding," https://en.wikipedia.org/wiki/Vibe_coding
3. Osmani, A., "Agentic Engineering," https://addyosmani.com/blog/agentic-engineering/
4. Karpathy, A., "From Vibe Coding to Agentic Engineering," 2026; The New Stack, "Vibe Coding is Passe," https://thenewstack.io/vibe-coding-is-passe/
5. Glide Blog, "What is Agentic Engineering?" https://www.glideapps.com/blog/what-is-agentic-engineering; The New Stack, "Vibe Coding, Agentic Engineering," https://thenewstack.io/vibe-coding-agentic-engineering/
6. CircleCI, "AI-Native SDLC," https://circleci.com/blog/ai-sdlc/
7. GroovyWeb, "SDLC in the AI Era: Software Development 2026," https://www.groovyweb.co/blog/sdlc-ai-era-software-development-2026; EPAM, "From Traditional Software to a Native AI SDLC," https://www.epam.com/about/newsroom/in-the-news/2026/from-traditional-software-to-a-native-ai-sdlc-how-genai-is-redefining-engineering
8. Osmani, A., "The Factory Model," https://addyosmani.com/blog/factory-model/
9. Deloitte, "AI in Software Engineering: Productivity Gains 2025-2026," dự báo mức tăng 30-35% trên toàn bộ quá trình phát triển.
10. METR, "Uplift Update: Measuring the Impact of AI Coding Tools," tháng 2 năm 2026, https://metr.org/blog/2026-02-24-uplift-update/
11. Google, "Introduction to Agents," Agents Whitepaper Series, tháng 11 năm 2025.
12. Osmani, A., "From Conductors to Orchestrators: The Future of Agentic Coding," https://addyosmani.com/blog/future-agentic-coding/

13. Google, "Jules: AI-Powered Coding Agent," https://developers.googleblog.com/en/the-next-chapter-of-the-gemini-era-for-developers/

14. Osmani, A., "The 80% Problem in Agentic Coding," https://addyo.substack.com/p/the-80-problem-in-agentic-coding

15. Medium, Dave Patten, "The State of AI Coding Agents 2026: From Pair Programming to Autonomous AI Teams," https://medium.com/@dave-patten/the-state-of-ai-coding-agents-2026-from-pair-programming-to-autonomous-ai-teams-b11f2b39232a

16. Lawfare, "When the Vibes Are Off: The Security Risks of AI-Generated Code," https://www.lawfaremedia.org/article/when-the-vibe-are-off--the-security-risks-of-ai-generated-code

17. Google, "Introduction to Agents," mục Multi-Agent Systems and Design Patterns, tháng 11 năm 2025.

18. Google, "Agent Development Kit (ADK)," https://google.github.io/adk-docs/; Kartakis, S., "From Zero to Multi-Agents: A Beginner's Guide to Google Agent Development Kit (ADK)," https://medium.com/@sokratis.kartakis/from-zero-to-multi-agents-a-beginners-guide-to-google-agent-development-kit-adk-b56e9b5f7861

19. Google, "Agent-to-Agent (A2A) Protocol," https://google.github.io/a2a-protocol/; Kartakis, S. và Hotz, H., "Generative AI in the Real World: Understanding A2A," O'Reilly Podcast, https://www.oreilly.com/radar/podcast/generative-ai-in-the-real-world-understanding-a2a-with-heiko-hotz-and-sokratis-kartakis/

20. TLDL, "AI Coding Tools 2026," https://www.tldl.io/resources/ai-coding-tools-2026; Kanerika, "GitHub Copilot vs Claude Code vs Cursor vs Windsurf," https://kanerika.com/blogs/github-copilot-vs-claude-code-vs-cursor-vs-windsurf/

21. Google, "Gemini Code Assist," https://cloud.google.com/gemini/docs/codeassist/overview

22. Dark Reading, "Coders Adopt AI Agents, but Security Pitfalls Lurk in 2026," https://www.darkreading.com/application-security/coders-adopt-ai-agents-security-pitfalls-lurk-2026

23. Google, "Gemini CLI," https://github.com/google-gemini/gemini-cli

24. Google, "Agent Tools: Interoperability with Model Context Protocol (MCP)," Agents Whitepaper Series, tháng 11 năm 2025

25. Google, "Agent Quality" và "Prototype to Production," Agents Whitepaper Series, tháng 11 năm 2025

26. Lawfare, "When the Vibes Are Off: The Security Risks of AI-Generated Code," https://www.lawfaremedia.org/article/when-the-vibe-are-off--the-security-risks-of-ai-generated-code

27. DevOps.com, "AI-Generated Code Packages Can Lead to Slopsquatting Threat," https://devops.com/ai-generated-code-packages-can-lead-to-slopsquatting-threat/

28. Osmani, A., "Beyond Vibe Coding," O'Reilly Media, 2025-2026, https://www.oreilly.com/library/view/beyond-vibe-coding/9798341634749/

29. "Awesome LLM Apps," https://github.com/Shubhamsaboo/awesome-llm-apps

30. Osmani, A., "My LLM Coding Workflow Going Into 2026," https://addyosmani.com/blog/ai-coding-workflow/

31. Questera, "7 AI Coding Trends to Watch in 2026," https://www.questera.ai/blogs/7-ai-coding-trends-to-watch-in-2026

32. DEV Community, "Programming in the Age of AI: From Code to Intent," https://dev.to/robertobutti/programming-in-the-age-of-ai-from-code-to-intent-46eo

Bản tiếng Việt được biên dịch và Việt hóa bởi **Phong Hồ**.

Nội dung gốc thuộc bản quyền Google. Đây là bản dịch phi thương mại dành cho cộng đồng và học viên, không phải ấn phẩm chính thức và không có liên kết tài trợ với Google.