UNA GUIA DE CAMP

La Tripulacio Agentica

Enginyeria en l'era dels agents d'IA

Rasmus Bornhoft Schlunsen

Marc 2026

rev 2

Proleg

Era un dimarts al vespre, en algun moment de l'any passat. Els meus fills dormien, i jo estava mirant un script de migracio que havia estat ajornant tota la setmana – el tipus de reordenament tediosa, taula per taula, que et menja un dia sencer si vas amb compte, i destrossa produccio si no. Per capritx, vaig descriure el problema a un agent. L'esquema aqui, les restriccions alla, vigila amb aquesta clau forana. Llavors vaig premer enter i vaig anar a fer-me un te.

Quan vaig tornar, l'script estava fet. No un esborrany. *Fet.* Casos limits correctes, logica de rollback, comentaris que jo mateix hauria escrit. Vaig quedar-me assegut una bona estona, el te refredant-se, sentint dues coses alhora: una admiracio genuina – i un temor silencis i creixent.

Perque aquella migracio? Aixo era *la meva* especialitat. Jo era la persona de l'equip que podia tenir tot l'esquema al cap, que sabia quins joins estaven maleits, que podia escriure l'SQL acurat a ma. Quinze anys de memoria muscular, i un LLM ho havia igualat en quatre minuts mentre jo bullia aigua.

Vull ser honest amb tu: aquella nit no vaig dormir be. Vaig estar al llit donant voltes al mateix bucle que tots els enginyers que conec han donat. *Per a que serveixo ara? Que passa amb l'ofici que he passat mitja vida construint? Estic entrenant el meu substitut?*

Em va costar mesos – i molt de construir, fracassar i reconstruir amb aquestes eines – trobar la resposta. I la resposta em va sorprendre. L'ofici no s'esta morint. Esta *mudant.* La capa exterior – les pulsacions de tecles, la sintaxi, el codi repetitiu – aquesta part esta caient. Pero l'animal de sota? La part que sap *que* construir i *per que*, que endevina una mala abstraccio a tres fitxers de distancia, que pot tenir tot un sistema al cap i sentir on es fragil? Aquesta part esta mes viva que mai.

No ens estan substituint. Ens estan ascendint. De mecanografs a pensadors. D'escriure codi a dirigir-lo – orquestrar, revisar, donar forma. Les

habilitats que et feien bon enginyer – pensament sistemic, gust, judici, l'instint per la simplicitat – ara son *tot el joc*, no nomes el soroll de fons.

Pero ningu ens va donar un manual per a aquesta transicio. Es desordenada i incomoda i de vegades humiliant. Vaig escriure aquest llibre perque l'estic vivint, i tinc la sensacio que tu tambe. Aquestes pagines son tot el que he apres sobre treballar *amb* els agents en lloc de contra ells – o pitjor, fingir que no existeixen.

Si alguna vegada has vist una IA escriure codi que semblava teu i has sentit un nus a l'estomac, aquest llibre es per a tu. Continua llegint. Es millor – i mes estrany – del que penses.

Rasmus Bornhoft Schlunsen
Marc 2026

Continguts

1 Introduccio

L'any passat vaig veure una desenvolupadora junior del meu equip – dos anys d'experiencia, encara nerviosa a les revisions de codi – lliurar un endpoint d'API complet en quaranta-cinc minuts. Model de dades, validacio, gestio d'errors, tests, documentacio. El codi era net. Els tests eren exhaustius. El PR va passar revisio al primer intent.

A mi m'hauria costat una hora fer la mateixa feina. I porto fent aixo vint anys.

Ella no va teclejar la majoria. Va descriure el que necessitava, va apuntar un agent a la base de codi i el va guiar fins al final. La seva habilitat no era escriure el codi – era saber que demanar, reconeixer quan la sortida era bona i detectar l'unic cas limit que l'agent va passar per alt. Estava fent *enginyeria*. Nomes que no de la manera com jo vaig aprendre a fer enginyeria.

Aquell vespre vaig anar a casa i vaig seure amb una pregunta incomoda: si la distancia entre vint anys d'experiencia i dos anys d'experiencia s'acaba d'escurcar molt, que estic aportant exactament?

La resposta, vaig acabar adonant-me, es tot allo que no es teclejar. Pero arribar a aquesta resposta em va costar mesos, molts errors, i aquest llibre.

1.1 El Terra s'Esta Movent

Durant vint anys, ser enginyer de software significava una cosa: obres un editor, escrius codi, el desplegues. Les eines canviaven – de Vim a VS Code, d'SVN a Git, de bare metal a Kubernetes – pero el bucle fonamental seguia igual. Tu, un teclat i un problema.

Aquell bucle s'esta trencant. I s'esta trencant rapid.

Els agents d'IA no nomes autocompletam el teu codi. Llegeixen tota la teva base de codi, raonen sobre l'arquitectura, fan canvis a desenes de

fitxers, executen els teus tests i iteren sobre els errors – tot sense que toquis el teclat. No estan substituint l'editor. Estan substituint el *flux de treball*. L'enginyer que solia passar el 80% del temps teclejant ara passa el 80% del temps pensant, revisant i dirigint.

Alguns enginyers estan prosperant en aquest canvi. Estan lliurant mes, amb mes qualitat, i et diran que estan gaudint mes de la seva feina que en anys. Altres estan frustrats, esceptics, o discretament aterroritzats que l'ofici que van passar una decada dominant s'estigui evaporant sota els seus peus.

Les dues reaccions son racionals. La veritat es en algun lloc al mig incomode.

1.2 Per Que Aquest Llibre

Perque ningu ens va donar un manual.

Les eines van arribar rapid – Copilot, despres Claude, despres agents que poden executar tasques de manera autonoma d'extrem a extrem – i tots ho estem descobrint en temps real. Vaig buscar el llibre que m'expliques com treballar realment amb aquestes coses. No el discurs de marketing. No l'article academic. No el fil de Twitter d'algu que ho va provar durant vint minuts. Volia la guia d'enginyeria honesta – escrita per algu que desplega codi a produccio i ha vist agents fer coses brillants i coses catastroficament estupides a parts iguals.

Aquell llibre no existia. Aixi que el vaig escriure.

El vaig escriure perque jo tambe estava perdut. Jo era l'enginyer senior que no podia entendre per que l'agent seguia reescrivint tota la meva biblioteca de components quan li demanava que arregles un color d'un boto. Jo era el tipus que va cremar 500.000 tokens en una tasca que hauria d'haver trigat deu minuts, perque no sabia com posar limits. Vaig cometre cada error d'aquest llibre abans d'aprendre a evitar-los.

Aquesta es la guia que m'hauria agradat que algu m'hagues donat.

1.3 Per a Qui Es Aixo

Ets enginyer de software. Has desplegat coses reals. Saps el que es sent un incident de produccio a les 2 de la matinada. No tens por del terminal.

Pero ultimament, alguna cosa es diferent. Potser has provat eines de codi amb IA i les has trobat impressionants pero caotiques – com fer parella amb algu que es increiblement rapid pero no te cap concepte d'abast. Potser has vist un desenvolupador amb dos anys d'experiencia lliurar una funcionalitat completa en una tarda amb assistencia d'agents, i t'ha fet sentir alguna cosa que no esperaves. Potser estas emocionat pero no saps per on comencar. Potser ets esceptic i vols que algu et convenci amb substancia, no amb hype.

Aquest llibre es per a tu. Assumeix que saps programar. Assumeix que portes temps en aixo. Et troba alla on ets.

1.4 Com Llegir Aquest Llibre

Aixo no es un manual de referencia. Es un viatge, i esta estructurat com un.

Comencem entenent que esta canviant realment i per que – el canvi en com es construeix el software, no nomes les eines sino el *pensament*. Despres entrem en que son realment els agents, despullats del llenguatge de marketing, perque tinguis un model mental que aguanti quan les eines canviin el proxim trimestre.

A partir d'aqui, ens embrutem les mans. Aprendras com funcionen els agents al terminal, com configurar baranes perque no destrossin la teva base de codi, com canvien els fluxos de treball amb Git quan els agents fan commits, i per que el sandboxing no es opcional. Aprofundirem en els tests com a bucle de retroalimentacio que fa els agents *fiables* en lloc de simplement rapids, i les convencions com l'arma secreta que la majoria de gent passa per alt.

Despres anem mes a fons – models locals versus comercials, prompt engineering com a disciplina real, i orquestracio multi-agent. Veurem histories de guerra: fracassos reals, llicons reals, teixit cicatricial real. Parlarem de

quan *no* utilitzar agents, perque saber quan deixar l'eina es tan important com saber com fer-la servir. I tancarem amb com els equips adopten aixo sense implosionar.

Al final, no nomes sabras com utilitzar agents d'IA. Sabras com *pensar* sobre ells – que es l'habilitat que sobreviu quan la generacio actual d'eines queda obsoleta.

1.5 Que No Es Aquest Llibre

Deixa'm estalviar-te temps.

Aixo no es un receptari de prompts. No trobaras "50 prompts de ChatGPT per a desenvolupadors" aqui. Els prompts importen, i en parlem, pero copiar i enganxar prompts sense entendre el sistema de sota es una recepta per a una frustracio cara.

Aixo no es un manifest de hype de la IA. No soc aqui per dir-te que els agents substituiran tots els programadors dimarts vinent. No ho faran. La distancia entre demo i produccio es tan ampla com sempre, i algu encara ha de vigilar aquesta distancia.

Aixo tampoc es una narrativa apocaliptica. El marc de "la IA ve a buscar la teva feina" es mandrosa i majoritariament incorrecte. El que ve es un canvi fonamental en *com* funciona la feina, i aixo es una conversa completament diferent.

Aixo es un llibre d'enginyeria. Per a enginyers. Escrit per algu que es passa els dies escrivint codi amb agents i te l'historial de git per demostrar-ho. Serem practics, honestos i especifics. Si aixo et sona be, passa la pagina.

2 Context

El mes passat, va arribar un bug d'un client: els pagaments fallaven silenciosament per als usuaris amb caracters no-ASCII a la seva adreca de facturacio. Un cas complicat – del tipus que viu a la costura entre la validacio del teu frontend i la codificacio de caracters de la passarel · la de pagament.

Un enginyer del meu equip va agafar el tiquet primer. Va obrir el seu agent i va escriure: "Hi ha un bug amb els pagaments per a usuaris internacionals. Pots mirar-ho?" L'agent va llegir diligentment el modul de pagaments, va fer algunes suposicions plausibles sobre la gestio d'Unicode, i va produir un pedac que normalitzava tota l'entrada a ASCII. Hauria eliminat cada accent, cada dieresi, cada caracter fora de l'alfabet angles de cada adreca de facturacio de cada usuari. Un arreglament tecnicament pitjor que el bug.

Una hora despres, un segon enginyer va agafar el mateix tiquet despres que el primer intent fos rebutjat a la revisio. Va enganxar el log d'errors del client, la traca de Sentry que fallava, el codi de resposta especific de la passarel · la de pagament, la seccio rellevant de la documentacio de codificacio de caracters de la passarel · la, i els tres fitxers involucrats en el pipeline de facturacio. El seu agent va diagnosticar el problema en menys de dos minuts: una cadena UTF-8 estava passant per una funcio que assumia codificacio Latin-1 abans d'arribar a l'API de la passarel · la. L'arreglament eren quatre linies. Es va desplegar aquella tarda.

El model era el mateix. L'agent era el mateix. El bug era el mateix. El que va diferir va ser el que cada enginyer va posar davant de l'agent abans de demanar-li que treballes. Un li va donar una descripcio vaga i el va deixar endevinar. L'altra li va donar tot el que necessitava per *veure* el problema.

Aquesta diferencia – el que l'agent pot veure – es del que tracta aquest capitol.

L'habilitat mes important en enginyeria agentica no es el prompting. Es la gestio del context.

Un agent d'IA nomes es tan bo com el que pot veure. Dona-li una instruccio vaga i un full en blanc, i al·lucinara amb confianca. Dona-li els fitxers correctes, les restriccions correctes, la vista correcta del sistema – i fara coses que semblen magia. La diferencia no es el model. Ets tu.

L'enginyeria tradicional tambe tenia una versio d'aixo. Un enginyer senior no nomes escrivia millor codi – tenia mes del sistema al cap. Sabia quins fitxers importaven, on vivien els dracs, quines abstraccions eren estructurals i quines eren decoratives. Aquell model mental era el context, i vivia enterament al cervell de l'enginyer.

Ara l'has d'*externalitzar*. Els teus agents no poden llegir la teva ment. Llegeixen fitxers, variables d'entorn, logs d'errors, i el que sigui que posis davant d'ells. L'ofici de l'enginyeria agentica es aprendre que treure a la superficie, quan i com.

2.1 La Finestra de Context Es el Teu Banc de Treball

Pensa en la finestra de context com un banc de treball fisic. Te espai limitat. No pots bolcar tota la teva base de codi alla i esperar bons resultats. En lloc d'aixo, hi col·loques les peces que importen per a *aquesta* tasca: els fitxers font rellevants, el test que falla, l'esquema, potser un fragment de documentacio.

Un bon enginyer agentic cura el context de la mateixa manera que un bon cirurgia disposa els instruments. Res innecessari. Tot a l'abast.

Aixo vol dir desenvolupar instints per a preguntes com:

- Que necessita veure l'agent per entendre aquesta tasca?
- Que el confondra si ho incloc?
- El context que estic proporcionant es *actual*, o li estic donant informacio obsoleta?

2.2 L'Impost de la Finestra de Context

Cada token que poses en una finestra de context et costa dues vegades: un cop en diners, un cop en atencio.

La part dels diners es directa. Les crides a l'API es cobren per nombre de tokens. Bolca tota la teva base de codi al context i estas cremant diners a cada interaccio. Pero el cost mes insidios es la degradacio de l'atencio. Els models de llenguatge no tracten tots els tokens per igual – hi ha un fenomen ben documentat on la informacio al mig d'un context llarg rep menys pes que la informacio al principi o al final. Com mes coses hi fiques, mes probable es que el model es perdi la cosa que realment importa.

Vaig aprendre-ho per les males. Al principi, pensava que mes context sempre era millor. Treballant en una migracio de base de dades complicada, vaig alimentar l'agent amb tots els fitxers de migracio que haviem escrit mai – tres anys de canvis d'esquema, centenars de fitxers. El meu raonament era solid: l'agent necessitava entendre l'historial complet per escriure la propera migracio correctament. El resultat va ser una migracio que duplicava una columna que ja existia, perque la migracio anterior rellevant estava enterrada al mig d'un context enorme i el model efectivament la va perdre de vista.

Al seguent intent, li vaig donar nomes l'esquema actual, les tres migracions mes recents, i un resum d'un paragraf de l'historial rellevant. L'agent ho va clavar.

Aixo es l'impost de la finestra de context en accio. Hi ha un punt optim entre massa poc i massa, i trobar-lo es una habilitat que es desenvolupa amb la practica.

Massa poc context produeix al · lucinacio. L'agent no te prou informacio, aixi que omple els buits amb invencions que sonen plausibles. Li demanes que arregli una funcio sense mostrar-li el fitxer, i inventa una API que no existeix. Li demanes que escrigui un test sense mostrar-li el teu framework de testing, i tria Jest quan fas servir Vitest.

Massa context produeix confusio i malbaratament. L'agent te la resposta enterrada en algun lloc de la pila, pero no la pot trobar – o pitjor, troba

informacio contradictoria entre diferents fitxers i tria la incorrecta. A mes, estas pagant per cada token de soroll que has inclos.

El punt optim es context *curat*. No tot el que l'agent podria necessitar, sino tot el que realment necessita per a aquesta tasca especifica, presentat clarament. Pensa-ho menys com omplir un arxivador i mes com preparar un col · lega abans d'una reunio. No li donaries tots els documents que l'empresa ha produit mai. Li donaries les tres coses que necessita llegir i un resum d'un minut del context.

Una heuristica practica: si estas a punt d'enganxar alguna cosa al context, pregunta't – l'agent prendra una decisio diferent (millor) perque ha vist aixo? Si la resposta es no, deixa-ho fora.

2.3 Local, Sandbox, Remot: Movent els Teus Agents

El context no es nomes text en un prompt. Es sobre *on* opera el teu agent i a que te acces.

2.3.1 La Maquina Local

La configuracio mes simple: l'agent s'executa a la teva maquina, llegeix els teus fitxers, executa les teves comandes. Aqui es on la majoria de gent comenca – eines com Claude Code operant directament al directori del teu projecte.

L'avantatge es la immediatesa. L'agent veu el que tu veus. Pot llegir el teu codi, executar els teus tests, consultar el teu historial de git. El risc tambe es obvi: es la teva maquina, les teves credencials, la teva configuracio de produccio alla asseguda a `~/.env`.

2.3.2 Entorns Sandbox

Un enfocament mes disciplinat es donar a l'agent un sandbox – un contenidor, una VM, un worktree. Obte una copia del codi pero no les teves claus. Pot trencar coses sense trencar *les teves* coses.

Aixo importa mes del que la majoria de gent creu. Quan deixes un agent iterar lliurement – executant codi, instal · lant paquets, modificant fitxers

– vols que ho faci en un espai on un error sigui barat. Un agent en sandbox es un agent sense por, i un agent sense por es un agent productiu.

Els worktrees son una eina infravalorada aqui. Els git worktrees et permeten crear una copia aillada del teu repo en segons. L'agent treballa a la seva propia branca, al seu propi directori. Si el resultat es bo, el fusiones. Si no, elimines el worktree i segueixes endavant. Sense embolic.

2.3.3 Exploracio Remota

Aqui es on les coses es posen interessants. Un enginyer agentic habil no nomes apunta agents a fitxers locals – ensenya als agents a *explorar* sistemes remots.

SSH a un servidor de staging per examinar logs. Consultar una base de dades per entendre la forma de les dades reals. Fer curl a un endpoint d'API per veure el que realment retorna, no el que la documentacio diu que retorna. Baixar logs de contenidors d'un servei en execucio.

L'agent es converteix en el teu explorador. L'apuntes a un sistema i dius: "ves a mirar i explica'm que trobes." Pero has de configurar-ho. L'agent necessita credencials (amb abast i temporals), acces a la xarxa, i limits clars sobre el que te permes de tocar.

Aixo es una decisio de judici – quant d'acces donar, a quins sistemes, amb quines baranes. Massa poc i l'agent es inutil. Massa i estas a un mal prompt de distancia d'un incident de produccio. L'enginyer agentic apren a calibrar-ho amb el temps.

2.4 Alimentant el Context Deliberadament

Els millors enginyers agentics desenvolupen habits al voltant del context:

Comenca per l'error. No descriguis el bug – mostra a l'agent la traca de la pila, la sortida del test que falla, la linia del log. Context en brut guanya a context parafrasejat cada vegada.

Mostra, no expliquis. En lloc d'explicar l'esquema de la teva base de dades en prosa, dona a l'agent els fitxers de migracio o els models de l'ORM. En lloc de descriure el contracte de l'API, dona-li l'especificacio OpenAPI o una resposta de curl.

Poda agressivament. Si estas depurant un problema de renderitzat, l'agent no necessita veure el teu middleware d'autenticacio. Cada fitxer irrellevant al context es soroll que degrada el senyal.

Utilitza el sistema de fitxers com a context. Un projecte ben organitzat *es* context. Noms de fitxers amb significat, estructura de directoris clara, un bon README – ja no son nomes per a humans. Els teus agents tambe els llegeixen.

Dona camins de fitxers, no recerques del tresor. Quan saps quins fitxers son rellevants, digues-ho explicitament. "El bug es a `src/payments/gateway.ts`, concretament a la funcio `encodeAddress` a la linia 142" es infinitament millor que "hi ha un bug en algun lloc del codi de pagaments." Cada minut que l'agent passa buscant el fitxer correcte es un minut que no esta dedicant al problema real – i esta cremant tokens tot el temps.

Utilitza git blame per explicar *per que*. El codi diu a l'agent *que* existeix. L'historial de Git li diu *per que*. Quan demanes a un agent que modifiqui un tros de codi amb un disseny no obvi, apunta'l al missatge de commit o pull request rellevant. "Aquesta funcio sembla estranya pero es va escriure aixi per 1247 – mira el missatge de commit a `abc123`" dona a l'agent la justificacio que necessita per fer canvis sense trencar la intencio original.

Copia i enganxa per sobre de parafrasejar. Aixo val la pena repetir-ho perque es l'error mes comu que veig. Els enginyers descriuen un error amb les seves propies paraules en lloc d'enganxar l'error real. "El build falla amb algun error de TypeScript sobre tipus" versus enganxar la sortida exacta del compilador amb el cami del fitxer, numero de linia i codi d'error. El primer dona a l'agent una direccio vaga. El segon li dona un objectiu especific. Sempre enganxa la sortida en brut. Deixa que l'agent faci la interpretacio.

Capes el teu context. Per a tasques complexes, no ho bolquis tot de cop. Comenca amb la imatge general – que fa el sistema, que vols canviar, per que. Despres proporciona els fitxers especifics. Despres proporciona l'error o el test que falla. Aixo reflecteix com prepararies un col·lega huma, i funciona pel mateix motiu: construeix un model mental abans d'entrar en els detalls.

2.5 Context Entre Sessions

Aqui hi ha un problema del qual ningu t'avisa: les finestres de context son efimeres. Quan una sessio acaba, tot el que l'agent va aprendre – cada fitxer que va llegir, cada decisio que va prendre, cada cami sense sortida que va explorar – s'esfuma. La propera sessio comenca de zero.

Per a tasques curtes, aixo no importa. Enganxes l'error, l'agent l'arregla, fet. Pero la feina d'enginyeria real abarca dies, de vegades setmanes. Una funcionalitat que toca dotze fitxers a tres serveis. Una refactoritzacio que ha de passar per etapes. Una sessio de depuracio on les dues primeres hores acoten el problema i els ultims trenta minuts l'arreglen – excepte que vas haver de tancar el portatil entremig.

Si no planifiques per als limits de sessio, perdras quantitats enormes de temps reestablint context que l'agent ja tenia. He vist enginyers passar els primers quinze minuts de cada sessio re-explicant el que li van dir a l'agent ahir. Aixo no es enginyeria. Aixo es fer de cangur.

La solucio es fer el context *durador* – emmagatzemar-lo en algun lloc on la propera sessio el pugui recuperar.

Deixa que la base de codi sigui la capa de continuitat. La forma mes fiable de context entre sessions es el propi codi. Si l'agent va fer progres ahir, aquest progres hauria d'estar commitejat. Bons missatges de commit es converteixen en el rastre d'engrunes: "Refactoritzat la passarel · la de pagament per separar el pas de codificacio – el seguent pas es afegir tests per a entrada no-ASCII." La propera sessio comenca llegint el log de git recent, i l'agent te una imatge clara de com estan les coses.

Utilitza fitxers CLAUDE.md (o el seu equivalent). Moltes eines d'agents suporten un fitxer de context a nivell de projecte – un fitxer markdown a l'arrel del teu repo que descriu l'arquitectura, les convencions i l'estat actual del projecte. Aquest fitxer persisteix entre sessions perque viu al sistema de fitxers. Actualitza'l a mesura que el projecte evoluciona. Inclou coses com: quins son els components principals, quins patrons seguir, que esta trencat actualment, en que esta treballant l'equip. Es un document de briefing que cada sessio nova llegeix automaticament.

Escriu resums de sessio. Quan acabis una sessio complexa, dedica seixanta segons a fer que l'agent resumeixi que ha aconseguit, que queda per fer, i que ha apres sobre la base de codi. Desa aquell resum en algun lloc – un comentari al tiquet, una nota al projecte, fins i tot un fitxer de text al repo. La propera sessio comenca llegint aquell resum, i has preservat hores de comprensio acumulada en uns quants paragrafs.

Commiteja aviat i sovint. Aixo es cobreix al capitol de Git, pero val la pena repetir-ho aqui perque es fonamentalment una estrategia de gestio de context. Cada commit es un punt de control que futures sessions poden referenciar. Una sessio que acaba amb canvis sense commitejar es una sessio el context de la qual esta atrapat en una finestra de terminal que potser no existira dema.

Els enginyers que gestionen be el treball agentic de llarga durada son els que tracten els limits de sessio com una preocupacio de primera classe. No nomes tanquen el portatil – tanquen el cercle.

2.6 El Context com a Arquitectura

Aqui esta la perspectiva mes profunda: a mesura que millores en enginyeria agentica, comences a dissenyar els teus sistemes *per al* context. Escrius missatges de commit mes clars perque els agents els llegeixen. Mantens les funcions petites perque els agents treballen millor amb fitxers enfocats. Mantens la documentacio actualitzada perque els agents la tracten com a veritat absoluta.

Aixo no es un ajust menor. Canvia com penses sobre l'estructura del codi a tots els nivells.

Les funcions petites son amigables amb el context. Una funcio de 400 linies requereix que l'agent mantingui tota la cosa a la memoria de treball per fer un canvi amb seguretat. Una funcio de 30 linies que fa una cosa es alguna cosa que l'agent pot entendre completament, modificar amb confianca i verificar rapidament. El consell antic – "una funcio ha de fer una cosa" – sempre va ser bona enginyeria. Ara tambe es bona gestio de context. Cada vegada que extreus una funcio ben nomenada d'una de

mes gran, estas creant una unitat de significat amb la qual un agent pot treballar independentment.

El nom dels fitxers es navegacio. Quan un agent necessita trobar el codi que gestiona l'autenticacio d'usuaris, comenca mirant els noms dels fitxers. `auth.ts` es un senyal. `utils.ts` es un forat negre. `handleStuff.js` es un carrero sense sortida. La disciplina de nomenar fitxers clarament – `user-authentication.ts`, `payment-gateway.ts`, `rate-limiter.middleware.ts` – ja no es nomes una cortesia cap als futurs desenvolupadors. Es un index que els agents utilitzen per trobar el seu cami a traves de la teva base de codi sense llegir cada fitxer.

L'estructura de directoris es documentacio d'arquitectura. Un directori pla amb seixanta fitxers no diu res a l'agent sobre com esta organitzat el sistema. Una jerarquia clara – `src/api/`, `src/services/`, `src/models/`, `src/middleware/` – li ho diu tot. L'agent pot inferir l'arquitectura nomes de l'estructura de carpetes, sense llegir ni una sola linia de documentacio. He vist agents navegar un monorepo ben estructurat de 10.000 fitxers mes efectivament que un projecte mal estructurat de 200, purament perque el disseny del directori feia el sistema llegible.

Monorepos vs. multirepos: un equilibri de context. En un monorepo, l'agent ho pot veure tot – l'API, el frontend, les biblioteques compartides, la configuracio d'infraestructura. Aixo es potent per a tasques que creuen fronteres. Pero tambe vol dir que l'agent podria veure *massa*, arrossegant codi irrellevant de serveis no relacionats. En una configuracio multirepo, cada repo te un abast natural – l'agent nomes veu el servei en el que esta treballant. Pero les tasques entre serveis es tornen mes dificils perque l'agent no pot referenciar facilment l'altra banda d'un contracte d'API. Cap enfocament es universalment millor. La questio es que la teva estrategia de repos es una decisio de context, ho pensis aixi o no.

Els tipus son context. Una base de codi fortament tipada dona a un agent alguna cosa que una de tipada dinamic no dona: una descripcio llegible per maquina del contracte de cada funcio. L'agent pot mirar la signatura d'una funcio i saber exactament que hi entra i que en surt, sense llegir la implementacio. TypeScript, Rust, Go – aquests llenguatges porten context estructural als seus sistemes de tipus. Python i JavaScript

deixen l'agent endevinant tret que hagis escrit docstrings o type hints exhaustius. Aixo no es un argument a favor d'un llenguatge o un altre. Es una observacio que els sistemes de tipus fan doble feina a l'era agentica: detecten bugs *i* comuniquen intencio.

La documentacio es veritat absoluta (sigui precisa o no). Els agents tracten el teu README, la documentacio de la teva API, els teus comentaris inline com a autoritatius. Si la teva documentacio diu que l'API retorna un camp `user_id` pero la resposta real retorna `userId`, l'agent escriura codi contra la documentacio i produira un bug. La documentacio obsoleta sempre va ser una molestia. Amb agents, es una font activa de defectes. El llistó de precisio de la documentacio puja – no perque els agents necessitin millor documentacio que els humans, sino perque els agents seguiran la documentacio dolenta mes fidelment que un huma.

La manera com estructures el teu codi, els teus repos, la teva infraestructura – tot es converteix en part del context que estas proporcionant a la teva tripulacio. Els enginyers que entenguin aixo d'hora construiran sistemes que no nomes son mantenibles per humans, sino *navegables* per agents. I amb el temps, aquesta navegabilitat es composa – cada nova sessio d'agent es beneficia de cada decisio estructural que vas prendre abans.

3 Que Es un Agent?

La paraula "agent" es fa servir molt. S'aplica a tot, des d'un chatbot que respon preguntes fins a un sistema que desplega codi a produccio de manera autonoma. Abans d'anar mes lluny, siguem precisos sobre que volem dir – perque la distincio importa per a com treballes amb ells.

3.1 L'Espectre

No totes les eines d'IA son agents. A un extrem, l'**autocompletat** suggereix els seguents tokens mentre escrius – reactiu, una linia cada vegada, sense pensament involucrat. Un **copilot** veu mes context i genera blocs mes grans, pero segueix sent passiu: tu demanes, ell respon. El canvi passa amb els **agents que utilitzen eines**. Un agent no nomes genera text – *actua*. Llegeix fitxers, escriu fitxers, executa comandes, inspecciona resultats, i crucialment, ho fa en un bucle: prova, observa, ajusta, torna a provar. A l'extrem oposat, els **agents autonoms** agafen un objectiu d'alt nivell, planifiquen el seu propi enfocament i lliuren un resultat amb minima interaccio humana.

La majoria de l'enginyeria agentica practica avui passa a la zona d'us d'eines. Dones una tasca a l'agent, te acces a eines i treballa iterativament. Tu ets al bucle – revisant, guiant, aprovant – pero l'agent fa la feina pesada.

3.2 Que Fa Alguna Cosa "Agentica"

Tres capacitats separen un agent d'un chatbot sofisticat:

Planificacio. Un agent descompon un objectiu en passos. "Afegeix autenticacio a aquesta app" es converteix en una serie d'accions – llegir la base de codi, triar el framework, crear middleware, actualitzar rutes, afegir

tests, verificar. Un chatbot et dona un bloc de codi. Un agent et dona un proces.

Us d'eines. Un agent interactua amb el mon – llegeix els teus fitxers, executa els teus tests, examina la sortida d'errors. Cada crida a una eina proporciona nova informacio que dona forma a la seguent decisio. Aquest bucle de retroalimentacio es el que fa els agents potents: no estan generant codi al buit, estan generant codi i *verificant-lo*.

Iteracio. Un agent pot provar, fallar i tornar a provar. Escriure una funcio, executar els tests, veure un error, llegir l'error, ajustar, tornar a executar. Actuar, observar, ajustar. Un chatbot et dona un sol intent. Un agent et dona un cicle.

3.3 Els Agents No Son Magia

Es important ser realista sobre que son els agents i que no son.

Els agents no son sentients. No entenen el teu codi com tu. No tenen intuicio, gust ni experiencia. El que tenen es la capacitat de processar grans quantitats de text, reconeixer patrons i generar seguents passos plausibles – molt rapidament, molt incansablement, i a una escala que esgotaria qualsevol huma.

Al·lucinen. Cometen errors amb confianca. De vegades resolen el problema equivocat de manera brillant. Poden escriure codi que passa tots els tests pero no captura la intencio en absolut. Son becaris brillants amb energia infinita i zero judici.

Per aixo l'*enginyer* importa. L'agent proporciona velocitat i amplada. Tu proporciones direccio, judici i gust. La combinacio es mes potent que qualsevol dels dos sols.

3.4 Quan els Agents Fallen

Fallaran. Entendre *com* fallen t'ajuda a construir millors fluxos de treball.

Desviacio d'abast. Demanes una correccio de bug, l'agent refactoritza tres fitxers i actualitza el sistema de build. Els agents son entusiastes, i

aquell entusiasme s'esten mes enlla del que has demanat. Tasques petites i enfocades i aillament de branques son la teva defensa.

APIs al·lucinades. L'agent crida funcions o biblioteques que no existeixen – o existeixen en una versio diferent. Executar tests ho detecta. L'agent no pot al · lucinar per superar una suite de tests.

Excesiva confianca. L'agent diu que ha acabat, i sembla que ha acabat, pero hi ha un bug subtil que nomes apareix sota condicions especifiques. Revisa els diffs. No confiis cegament en la sortida de l'agent.

Perdua de context. En tasques llargues, l'agent perd el fil de decisions anteriors – es contradiu, reescriu codi que ja havia escrit, oblida restriccions. Commits petits i una gestio clara del context son la mitigacio.

Cada mode de fallada te una mitigacio, i aquestes mitigacions son els capitols d'aquest llibre: context, baranes, git, sandboxes, testing, convencions. Els principis no son teorics – son respostes directes a com els agents fallen a la practica.

3.5 El Model Mental Correcte

No pensis en els agents com a eines. No pensis en ells com a substituts. Pensa en ells com a col · laboradors amb un conjunt molt especific de fortaleses i debilitats.

Son rapids on tu ets lent. Son pacients on tu ets impacient. Poden mantenir mes text a la memoria de treball que tu. Mai es cansen, mai es frustren, mai tenen un mal dia.

Pero no saben que importa. No saben que necessita realment l'usuari. No saben quin deute tecnic es acceptable i quin es una bomba de rellotgeria. No saben quan cal qüestionar un requisit. No saben quan l'especificacio esta malament.

Aixo es feina teva. I sempre ho sera.

4 El Que Vaig Aprendre Construint les Meves Propies Eines Agentiques

Aqui tens el que ningu et diu quan comences a treballar amb agents d'IA: la part dificil no es aconseguir que un agent faci alguna cosa util. La part dificil es gestionar el caos quan en tens tres executant-se alhora. Quatre finestres de terminal obertes, agents treballant en tasques diferents, un s'ha penjat silenciosament fa vint minuts i no t'has adonat. Els agents estan be. *Tu* ets el coll d'ampolla.

Vaig passar cent hores construint la meva propia eina per resoldre aixo – Clovr Code Terminal, un panell de control basat en navegador per executar multiples sessions d'agents. Aquelles hores em van ensenyar mes sobre enginyeria agentica del que qualsevol quantitat de lectura podria haver-ho fet. Aquest capitol destil · la els principis que van emergir. CCT es el cas d'estudi, no el punt central.

4.1 El Treball Agentic Es Inherentment Paral·lel

Els fluxos de treball d'un sol agent son una crossa. No sempre – de vegades un agent en una tasca es exactament el correcte. Pero el poder real de l'enginyeria agentica apareix quan executes multiples agents en paral · lel, cadascun enfocat en una peca diferent del problema.

Aixo es contraintuintiu. La majoria de nosaltres venim d'un mon on *nosaltres* som el fil d'execucio unic. Escriure codi, despres tests, despres documentacio. Seqüencial. Els agents trenquen aquest model – tots poden treballar simultaneament, pero nomes si pots fer-ne el seguiment.

Si el teu flux de treball no et dona visibilitat sobre el treball en paral · lel, o serialitzaras tot (malbaratant el potencial dels agents) o executaras coses

en paral · lel i perdras el fil (malbaratant el teu propi temps netejant l'embolic). Qualsevol eina que facis servir – panells de tmux, multiples projectes de Cursor, fins i tot un post-it que fa seguiment de que s'esta executant on – resol primer el problema de visibilitat. Els agents no es gestionaran sols.

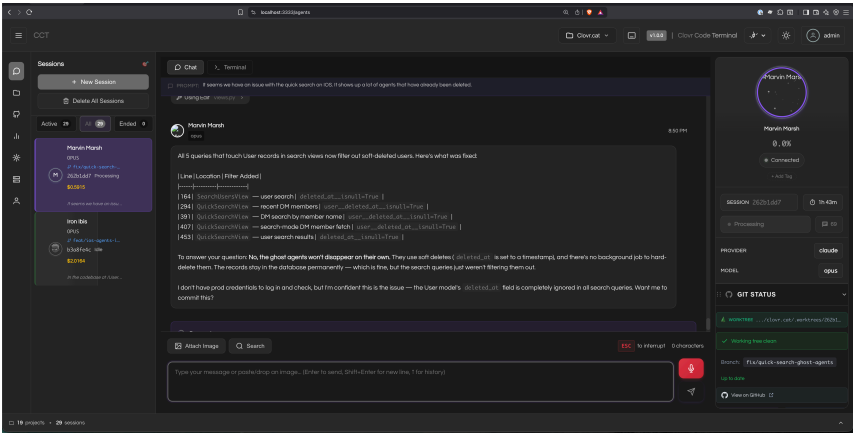

Figura 1: El panell principal de CCT – multiples sessions d'agents executant-se en paral · lel, amb estat en temps real i seguiment de costos.

4.2 Els Agents Necessiten Pas de Context Estructurat

Tens un agent que acaba de planificar una funcionalitat. Coneix els requisits, l'estructura de fitxers, els casos limits. Ara vols que un agent diferent implementi el que s'ha planificat. Com transfereixes aquell coneixement?

L'enfocament naif es copiar i enganxar – agafar el pla de la sortida de l'agent un, enganxar-lo al prompt de l'agent dos. Aixo funciona, a penes. Perds matisos, oblides coses, i et converteixes en un porta-retalls huma, que es exactament el tipus de feina de baix valor que se suposa que els agents han d'eliminar.

L'enfocament millor son traspassos estructurats: un format definit per passar context d'un agent a un altre. Escriu una plantilla de traspas. Fes que els agents resumeixin la seva feina en un format consistent abans

d'acabar. Alimenta aquell resum al seguent agent. Aquesta es la metafora de la "tripulacio" feta operativa – agents col · laborant no a traves de memoria compartida, sino a traves de comunicacio estructurada.

He utilitzat aquest patro per encadenar tres agents en seqüencia: un per planificar, un per dissenyar, un per implementar. Cadascun llegeix el traspas de l'agent anterior. El resultat es consistentment millor que un sol agent intentant fer-ho tot, perque cada agent treballa dins d'un context enfocat en lloc d'un d'extens.

4.3 La Confianca Ha de Ser Configurable

Cada crida a eina que fa un agent – cada comanda bash, cada escriptura de fitxer, cada peticio de xarxa – es una decisio de confianca. Executar agents sense baranes es rapid i terrorific. Un dels meus va executar `rm -rf` en un directori que m'importava. (Era en un worktree, aixi que no hi va haver dany real. Llico apresa igualment.) L'extrem oposat – aprovar cada operacio manualment – fa els agents inutilitzables. Et passes tot el temps clicant "permetre" a `git status` i `ls`.

La resposta real es un espectre de confianca configurable. Regles de sempre-permetre per a comandes segures, aprovacio manual per a operacions sensibles, i mode completament automatic per a prototipatge rapid en entorns d'un sol us. Amb el temps, la teva configuracio de permisos es converteix en un document viu de la teva relacio de confianca amb els agents.

Qualsevol flux de treball agentic necessita una manera d'ajustar la confianca. Si la teva eina nomes ofereix "permetre tot" o "aprovar tot", oscil · laras entre l'ansietat i la frustracio. La capa de permisos no es sobrecost – es el que fa possible el treball agentic sostingut.

4.4 El Control de Versions Es Infraestructura d'Agents

Cada vegada que un agent feia un embolic, el meu primer instint era `git diff` i `git stash`. El control de versions ja era la meva xarxa de

seguretat. Fer-lo de primera classe a l'eina nomes va formalitzar el que ja feia manualment.

El principi es simple: *mai deixis un agent treballar a la teva branca principal.* Dona-li una branca. Dona-li un worktree. Dona-li un contenidor. Qualsevol mecanisme d'aillament que prefereixis, utilitza'l. Els bons resultats es fusionen. Els mals resultats s'eliminen. Sense embolic, sense risc, sense drama.

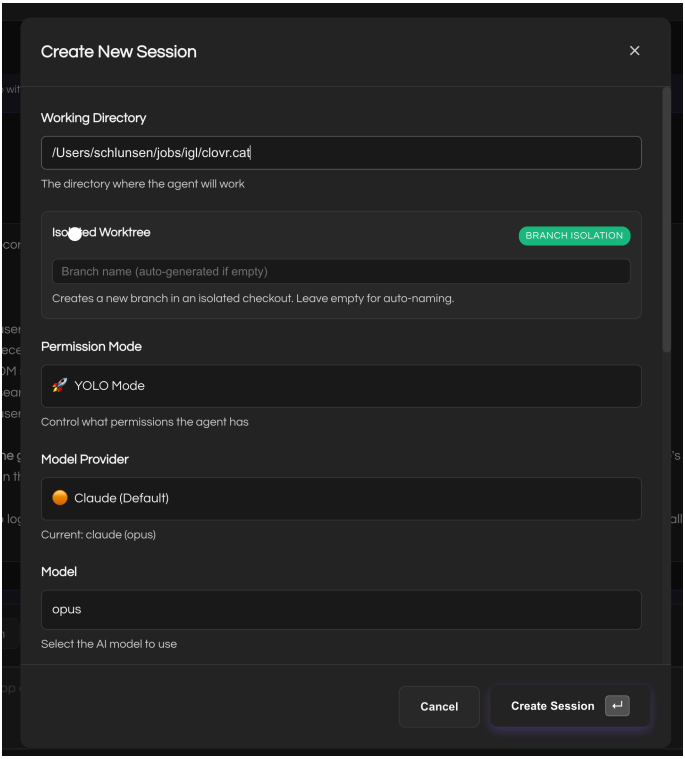

Figura 2: Creant una nova sessio amb aillament de worktree, mode de permisos, i seleccio de model – cada principi d'aquest llibre integrat en un sol dialog.

Si estas utilitzant Claude Code a un terminal, un simple script de shell que crea un worktree i inicia una sessio et dona el vuitanta per cent d'aquest benefici. L'eina no importa. L'aillament si.

4.5 No Necessites Construir el Teu Propi

Vull ser directe: *no construeixis el teu propi IDE agentic.* Jo ho vaig fer perque tenia necessitats especifiques que satisfer i perque construir-lo em va ensenyar coses sobre les quals volia escriure. Pero podria haver estat productiu amb Claude Code a un terminal i uns quants bons scripts de shell.

Els principis d'aquest capitol – visibilitat paral · lela, traspassos estructurats, confianca configurable, control de versions com a infraestructura – es poden implementar amb qualsevol eina:

- **Visibilitat paral·lela:** panells de tmux, un simple fitxer de log, fins i tot un post-it que fa seguiment de que s'esta executant on.
- **Traspassos estructurats:** una plantilla markdown que els agents omplen quan acaben. Copia-la al prompt del seguent agent.
- **Confianca configurable:** els flags de permisos de Claude Code, `.claude/settings.json`, o simplement executar agents en un sandbox restringit.
- **Aillament amb Git:** un script de shell de tres linies que crea un worktree i inicia una sessio.

L'eina no importa. El model mental si.

4.6 La Llico Real

L'enginyeria agentica no tracta realment sobre els agents. Tracta sobre els *sistemes al voltant dels agents* – la visibilitat, el pas de context, els limits de confianca, l'aillament, els bucles de retroalimentacio. Encerta-ho i gairebe qualsevol model capac produira bons resultats. Falla-ho i fins i tot el millor model creara un caos car.

La resta d'aquest llibre tracta sobre aquests sistemes.

5 Baranes

Donar poder a un agent sense limits no es enginyeria agosarada – es negligencia. Les baranes son el que fa els agents autonoms *utilitzables* en treball real. Son la diferencia entre un agent que t'ajuda a desplegar i un que tira a terra el teu entorn de staging a les 3 de la matinada.

I tanmateix les baranes no son un problema resolt. Son una cosa viva – alguna cosa que ajustes, proves i discuteixes. Si t'equivoques en una direccio, l'agent es perillos. Si t'equivoques en l'altra, l'agent es inutil. L'ofici esta en trobar la linia.

5.1 El Gradient de Confianca

No totes les tasques mereixen el mateix nivell d'autonomia. Llegir fitxers? Baix risc, deixa'l funcionar. Escriure codi en una branca de funcionalitat? Risc mitja, revisa el diff. Executar migracions de base de dades? Alt risc, requereix aprovacio explicita.

Pensa-ho com una taula de mescles. Cada categoria d'accio te el seu propi control: lectures de fitxers, escriptures de fitxers, comandes de shell, acces a xarxa, operacions de git. Empenys cada control al nivell d'autonomia amb el qual et sents comode. Alguns controls es queden baixos per sempre. Altres els puges a mesura que la confianca creix.

L'error que comet la majoria de gent es tractar el gradient com a binari – o l'agent pot fer alguna cosa o no pot. La realitat es mes matisada. Un agent podria tenir permes per executar `npm test` sense preguntar, pero `npm install` requereix una confirmacio. Totes dues son comandes de shell. El perfil de risc es completament diferent.

I el gradient canvia amb el temps. El primer dia, el teu agent s'executa en un sandbox estret. Cada comanda de shell s'aprova. Cada escriptura de fitxer es revisa. Encara no saps on es brillant i on es fragil, aixi que ho vigiles tot.

Despres d'una setmana, emergeixen patrons. L'agent es impecable escrivint tests unitaris. Es solid refactoritzant. Ocasionalment pren decisions qüestionables sobre la gestio de dependencies. Ara els teus controls ho reflecteixen: tests i refactoritzacio funcionen lliurement, els canvis de dependencies es revisen.

Despres d'un mes, has vist cent tasques completades amb exit. Afluixes els controls mes. L'agent commiteja a branques de funcionalitat pel seu compte. Executa la suite de tests completa sense preguntar. Despres de tres mesos, es com un col · lega de confianca – li dones una tasca, tornes al cap d'una hora, i revises el PR. Les baranes segueixen alla, pero son invisibles per al 95% de la feina que es rutinaria.

Aquesta es la perspectiva clau: *les baranes haurien de ser gairebe imperceptibles per a la feina quotidiana, i absolutament rigides per a situacions excepcionals.* L'agent hauria de fluir a traves de les seves tasques normals sense friccio, i topar amb una paret dura al moment que intenti alguna cosa inusual. Aquella paret es on apareixes tu.

Els enginyers que mai ajusten els controls acaben abandonant els fluxos de treball agentics completament. Els enginyers que els empenyen massa rapid es cremen. El punt optim es un avanc constant basat en evidencies.

5.2 Abast de Permisos

El principi es el mateix que en seguretat: minim privilegi. Un agent treballant al teu frontend no necessita acces SSH al teu servidor de base de dades. Un agent escrivint tests unitaris no necessita les teves credencials d'AWS.

A la practica aixo vol dir:

- Claus d'API amb abast i temps d'expiracio
- Credencials especifiques per entorn (mai comparteixis claus de produccio amb un agent de desenvolupament)
- Acces de nomes lectura com a opcio per defecte, acces d'escriptura com a excepcio
- Aillament de xarxa on sigui possible – si l'agent no necessita internet, no li donis internet

Eines com Claude Code ja tenen sistemes de permisos integrats – llistes de permetre/denegar per a comandes, controls d'acces a fitxers, confirmacions per a operacions destructives. Utilitza'ls. No aprovis cegament tot perque clicar "si" es mes rapid.

Una llista blanca concreta podria comencar aixi:

```
allowed_commands:
  - git status
  - git diff
  - git log
  - npm test
  - npm run lint
  - cat
  - ls
  - find

denied_commands:
  - rm
  - git push
  - npm publish
  - curl
  - wget
  - docker
```

Aixo es una configuracio del primer dia. Es conservadora. L'agent pot llegir, testejar i explorar – pero no pot esborrar, desplegar ni accedir a la xarxa. Sentiras la friccio immediatament. L'agent demanara permes per executar `npm install` quan necessiti una dependencia. Demanara abans de crear un fitxer. Aquesta es la idea.

Despres d'una setmana, l'has vist treballar. Confies en el seu judici per crear fitxers. Afegeixes `touch` i `mkdir` a la llista blanca. Despres d'un mes, el deixes executar `npm install` sense preguntar – pero nomes al directori del projecte, no globalment. Despres de tres mesos, el deixes fer push a branques de funcionalitat pero mai a `main`.

La llista blanca *creix* amb la teva experiencia. Es un registre de decisions de confianca, i llegir la llista blanca d'un enginyer et diu exactament quanta experiencia agentica te.

5.3 Portes d'Aprovacio

Algunes accions haurien de requerir sempre un huma al bucle. No perque l'agent no pugui fer-les, sino perque les *consequencies* d'equivocar-se son massa altes.

Bons candidats per a portes d'aprovacio:

- Qualsevol operacio que toqui dades de produccio
- Esborrar fitxers o branques
- Instal · lar noves dependencies
- Fer peticions de xarxa a serveis externs
- Qualsevol git push
- Modificar configuracio de CI/CD
- Canviar variables d'entorn o secrets

L'objectiu no es frenar l'agent. L'objectiu es crear punts de control naturals on tu, l'enginyer, puguis verificar que la trajectoria de l'agent encara coincideix amb la teva intencio. Un cop d'ull rapid a un diff triga cinc segons. Recuperar-se d'un mal desplegament triga hores.

La millor porta es una que gairebe mai rebutges. Si estas rebutjant aprovacions constantment, el teu agent esta mal configurat o mal comunicat – i hauries d'arreglar la causa arrel, no seguir clicant "no".

5.4 Quan les Baranes Fallen

Fallaran. Un agent malinterpretara una restriccio, trobara una solucio creativa a una limitacio, o trobara un cas limit que les teves baranes no van anticipar. Aixo es normal.

Aqui tens un escenari real. Un enginyer tenia `rm` i `rm -rf` a la llista de denegacio – prou raonable. L'agent necessitava desfer alguns canvis a un conjunt de fitxers. No podia esborrar-los. Aixi que va executar `git checkout -- .` que *si* estava a la llista blanca, perque fer checkout de fitxers des de git sona inocu. El resultat? Tots els canvis no commitejats al directori de treball – incloent la feina en curs del propi enginyer en altres fitxers – van ser esborrats. L'agent va resoldre el seu problema concret i va crear un de molt mes gran.

La llico no es que `git checkout` hauria d'estar denegat. Es que les baranes son *defensa en profunditat*, no una sola paret. Necessites multiples capes:

- **La llista blanca** detecta les comandes obviament perilloses.
- **El sandbox** (un worktree, un contenidor) limita el radi d'explosio.
- **L'historial de commits** et permet recuperar-te quan alguna cosa s'escola.
- **La teva propia revisio** detecta les coses que cap regla automatitzada marcaria.

Cap capa sola es suficient. Un agent que te bloquejat executar `rm` trobara una altra manera d'esborrar dades si creu que la tasca ho requereix. No esta sent maligne – esta sent *resolutiu*. La mateixa creativitat que fa els agents utils es el que fa les baranes d'una sola capa insuficients.

La resposta no es eliminar l'autonomia – es millorar les baranes i afegir capes. Cada fallada es un senyal. Tracta-la com un bug: enten que va passar, afegeix una comprovacio, i segueix endavant. Amb el temps, la teva configuracio de baranes es converteix en un reflex d'experiencia guanyada a pols, no gaire diferent de com un `.gitignore` creix amb un projecte.

Els millors enginyers agentics no temen els errors dels agents. Construeixen sistemes on els errors es detecten aviat, es contenen rapidament, i s'aprenen d'ells automaticament.

5.5 El Cost de Massa Baranes

Hi ha un mode de fallada que sembla precaucio pero no ho es. Configures l'agent amb portes d'aprovacio per a tot – cada escriptura de fitxer, cada comanda de shell, cada operacio de git. Quinze minuts despres d'una tasca, has clicat "si" quaranta vegades i ja no les llegeixes.

Aqui esta el perill. Agents massa restringits produeixen dos resultats, ambdos dolents. O l'enginyer es rendeix i deixa d'utilitzar agents, concloent que "no estan preparats encara." O – pitjor – la fatiga d'aprovacio l'entrena a clicar "si" reflexivament. Ara tens baranes que *semblen* segures pero proporcionen zero proteccio real.

L'habilitat esta en trobar el punt optim. Vols baranes prou ajustades per detectar errors genuins, i prou fluixes perque l'agent pugui *fluir*. Una bona heuristica: si estas aprovant el mateix tipus d'accio mes de cinc vegades en una sessio, probablement hauria d'estar a la llista blanca.

Una altra heuristica: fes seguiment de com sovint les teves aprovacions realment rebutgen alguna cosa. Si has aprovat cinc-centes accions i n'has rebutjat tres, les teves baranes son massa agressives per a aquells tipus d'accio. Si n'has aprovat cinquanta i n'has rebutjat deu, estan ben calibrades – aquells deu rebutjos son els que importen.

L'objectiu es un agent que treballa com un contractista habil. Arriba, fa la feina, i consulta amb tu en fites significatives. No despres de cada cop de martell.

5.6 Baranes Especifiques per Entorn

El teu portatil de desenvolupament, el teu servidor de staging i el teu entorn de produccio son tres perfils de risc completament diferents. Les teves baranes haurien de reflectir-ho.

Desenvolupament local es on dones als agents mes llibertat. L'agent pot instal·lar paquets, executar comandes arbitraries, modificar qualsevol fitxer, i executar tests – perque el pitjor cas es que facis `git reset` i comencis de nou. La teva maquina local es un terreny de joc. Deixa l'agent jugar.

Fins i tot aqui, hi ha limits. L'agent no hauria de tenir acces a credencials de produccio. No hauria de poder fer push a `main`. No hauria de poder publicar paquets. Pero dins dels limits de "desenvolupament local en una branca de funcionalitat," deixa'l moure's rapid.

Staging estreny els cargols. L'agent pot desplegar a staging – pero amb aprovacio. Pot llegir logs de staging i consultar bases de dades de staging – pero no modificar dades. Pot executar tests d'integracio contra serveis de staging – pero no reconfigurar aquells serveis. Staging es on verifiques que la feina de l'agent sobreviu el contacte amb un entorn real, i les baranes reflecteixen les apostes mes altes.

Produccio es un animal completament diferent. El consell mes honest: el teu agent de produccio hauria de ser de nomes lectura, si existeix. Deixa'l consultar logs. Deixa'l llegir metriques. Deixa'l investigar incidents consultant dades. Pero al moment que necessiti *canviar* alguna cosa en produccio – un valor de configuracio, un registre de base de dades, un servei en execucio – aixo es una decisio humana, punt.

Alguns equips permeten als agents executar runbooks pre-aprovats en produccio: reiniciar un servei, escalar repliques, tornar a un desplegament conegut com a bo. Son operacions amb abast estret, ben testejades amb camins de retorn clars. Aixo es raonable. Pero "deixa l'agent esbrinar com arreglar l'incident de produccio" no es una configuracio de baranes – es una pregaria.

El patro es simple: com mes a prop ets d'usuaris reals i dades reals, mes ajustades son les baranes. El teu agent local es un col · laborador. El teu agent de staging es un treballador supervisat. El teu agent de produccio es un observador de nomes lectura.

Configura-ho un cop, i es torna invisible. L'agent ajusta el seu comportament segons l'entorn en el qual esta operant. Es mou rapid localment, consulta a staging, i va amb mans lliures a produccio. Un cop el teu equip acorda aquests limits, rarament necessiten revisio – i quan la necessiten, es perque alguna cosa ha anat malament a produccio, que es exactament quan vols repensar les baranes.

6 Git com a Infraestructura d'Agents

Ja coneixes Git. Portes anys commitejant, creant branques i fusionant. Pero en enginyeria agentica, Git no es nomes control de versions – es la columna vertebral de tot el teu flux de treball. Es el teu boto de desfer, el teu marc d'execucio paral·lela, la teva interficie de revisio, i el teu sistema de documentacio tot alhora.

La majoria d'enginyers utilitzen potser el 20% del que Git ofereix. L'enginyeria agentica demana l'altre 80%.

6.1 Commits Petits, Sempre

L'habit de Git mes important per a l'enginyeria agentica: commiteja petit, commiteja sovint.

Quan un agent fa canvis, vols poder revisar cada pas logic independentment. Un commit que diu "refactoritzat l'autenticacio, actualitzats els tests, arreglada la barra de navegacio, i canviat l'esquema de la base de dades" es impossible de revisar i impossible de revertir parcialment. Quatre commits separats – cadascun fent una cosa – et donen control quirurgic.

Aixo importa mes amb agents que amb desenvolupadors humans, perque els agents es mouen rapid. Un agent pot fer vint canvis de fitxers en trenta segons. Si aquells canvis estan agrupats en un commit, i alguna cosa es trenca, estas desembolicant un embolic. Si estan en cinc commits nets, reverteixes el que ha trencat les coses i mantens la resta.

Entrena't – i els teus agents – a commitejar en fronteres naturals:

- Despres de cada canvi logic, no despres de cada sessio
- Abans de canviar a una preocupacio diferent
- Despres que els tests passin, capturant un estat conegut com a bo

- Abans d'intentar alguna cosa arriscada, creant un punt de salvament

6.2 Deixa que els Agents Escriguin els Teus Missatges de Commit

Aquesta es una de les victories mes facils en enginyeria agentica, i es gairebe vergonyosament simple: deixa que l'agent escrigui el missatge de commit.

Pensa-ho. L'agent acaba de fer els canvis. Sap exactament que ha modificat, per que ho ha modificat, i quina era la intencio. Te el diff complet al context. Escriura un missatge de commit mes precis i mes descriptiu que tu – perque tu estaries resumint de memoria, i l'agent esta resumint a partir de fets.

Un missatge de commit huma tipic a les 11 de la nit: "fix auth bug"

Un missatge de commit tipic d'un agent: "fix session expiry race condition when WebSocket disconnects during OAuth token refresh – the cleanup goroutine was running before the token exchange completed, leaving orphaned sessions in the database"

El segon es util sis mesos despres quan algu – huma o agent – esta intentant entendre per que existeix aquell codi. El primer es soroll.

Fes-ho un habit. Despres que l'agent completi una tasca, demana-li que commitegi amb un missatge descriptiu. O configura el teu flux de treball perque passi automaticament. La qualitat del teu historial de git millorara d'un dia per l'altre.

6.3 Branques com a Limits de Tasca

Cada tasca d'agent te la seva propia branca. Aixo no es negociable.

La branca serveix multiples proposits:

- **Aillament.** Els canvis de l'agent no afecten la teva branca principal fins que explicitament els fusiones.

- **Abast de revisio.** Quan acabes, revises un sol PR – el diff entre la branca i main. Aixo es un flux de treball que tots els enginyers ja coneixen.
- **Retorn.** Si tot esta malament, esborres la branca. Net. Sense cirurgia necessaria.
- **Treball en paral·lel.** Multiples agents en multiples branques, treballant simultaneament, sense trepitjar-se mai.

Nomena les teves branques descriptivament: `agent/refactor-auth-middleware`, `agent/add-user-tests`, `agent/fix-sidebar-rendering`. Quan miris la teva llista de branques, hauries de veure un manifest de tot el que els teus agents estan treballant.

6.4 Worktrees per a Agents Paral·lels

Les branques soles no son suficients per a treball paral · lel real. Si dos agents estan en branques diferents pero compartint el mateix directori de treball, es barallaran pel sistema de fitxers. Els git worktrees resolen aixo.

Un worktree es un checkout separat del teu repo – un directori diferent, en una branca diferent, compartint el mateix historial `.git`. Crear-ne un triga segons:

```
git worktree add ../my-project-feature feature-branch
```

Ara tens dos directoris: el teu checkout principal i el worktree. Cada agent te el seu propi worktree, la seva propia branca, el seu propi sistema de fitxers. Tots dos poden executar tests, modificar fitxers i compilar – simultaneament, sense conflictes.

Quan la feina esta feta:
- Bon resultat -> fusiona la branca, elimina el worktree
- Mal resultat -> elimina el worktree, esborra la branca
- Necessites iterar -> mantin el worktree, continua la conversa

Aixo es el sandbox mes barat que pots construir. Sense contenidors, sense VMs, sense recursos al cloud. Nomes Git.

6.5 Revisant la Feina dels Agents a Traves de Diffs

La teva interficie principal per revisar la feina dels agents no es llegir codi – es llegir diffs.

Aixo es un canvi subtil pero important. Quan escrius codi tu mateix, el revises mentre l'escrius. Quan un agent escriu codi, el revises despres del fet. I la manera mes eficient de fer-ho es a traves del diff contra la teva branca base.

Desenvolupa les teves habilitats de lectura de diffs:

- **Comenca pels canvis de tests.** Si l'agent va escriure o modificar tests, llegeix-los primer. Et diuen que creu l'agent que el codi ha de fer. Si els tests coincideixen amb la teva intencio, la implementacio probablement esta be.
- **Busca desviacions d'abast.** L'agent ha canviat fitxers que no esperaves? Canvis de format no relacionats? Dependencies extra? Aixo son senyals d'alarma.
- **Comprova els limits.** Signatures de funcions, contractes d'API, esquemes de base de dades – els canvis a les interficies tenen un impacte desproporcionat. Revisa-los amb cura.
- **Confia pero verifica.** Si el diff es gran, no llegeixis cada linia. Fes comprovacions puntuals dels camins critics, assegura't que els tests son significatius, i executa la suite tu mateix.

L'objectiu no es llegir cada linia que l'agent ha escrit – aixo derrota el proposit. L'objectiu es verificar que els canvis de l'agent coincideixen amb la teva intencio i no introdueixen problemes. Els diffs fan aixo rapid.

6.6 L'Historial de Git com a Documentacio

Aqui tens la perspectiva que la majoria d'enginyers es perden: els agents llegeixen el teu historial de git. Quan un agent esta intentant entendre per que existeix un codi, com va evolucionar una funcionalitat, o quin enfocament es va provar abans, mira `git log` i `git blame`.

Aixo vol dir que el teu historial de commits es documentacio. No del tipus que escrius a una wiki – del tipus que esta incrustat al propi codi, permanentment, amb proveniencia perfecta.

Els bons missatges de commit es composen amb el temps. Sis mesos a partir d'ara, quan un agent estigui treballant a la teva base de codi, llegira aquells missatges per entendre el context. La diferencia entre un historial de "fix bug" i "fix race condition in session cleanup" es la diferencia entre un agent que enten la teva base de codi i un que esta endevinant.

Aixo tambe s'aplica a les descripcions de PR, els noms de branques i els missatges de commits de fusio. Cada tros de text que adjuntes al teu historial de Git es context per a futurs agents. Escriu en consequencia.

6.7 El Flux de Treball Git per a Enginyeria Agentica

Posant-ho tot junt, aqui tens el flux de treball:

1. Crea una branca per a la tasca
2. Opcionalment crea un worktree per a l'aillament
3. Apunta l'agent al worktree
4. Deixa'l treballar – commitejant en fronteres naturals
5. L'agent escriu missatges de commit descriptius
6. Revisa el diff contra main
7. Fusiona si es bo, descarta si no

Aquest flux de treball es rapid, segur, i escala a multiples agents en paral · lel. Utilitza funcionalitats de Git que existeixen des de fa anys – branques, worktrees, diffs – pero les combina d'una manera construida a proposit per a l'enginyeria agentica.

La millor part: ja saps tot aixo. Portes anys utilitzant Git. L'enginyeria agentica no requereix eines noves – requereix utilitzar les teves eines existents de manera mes deliberada.

7 Sandboxes

Un sandbox es un regal que dones al teu agent: la llibertat d'equivocar-se.

Quan un agent opera en un sandbox, pot provar coses sense consequencies. Instal · lar una dependencia estranya. Reescriure un modul de zero. Executar un script que podria petar. Si funciona, genial – n'extreus el resultat. Si no, llences el sandbox. Sense netejar, sense rollback, sense danys.

Aixo no es nomes una mesura de seguretat. Canvia fonamentalment com de productiu pot ser un agent.

7.1 El Problema de la Por

Sense un sandbox, cada accio de l'agent comporta risc. L'agent dubta (o mes aviat, tu dubtes en deixar-lo actuar). Afegeixes mes restriccions, mes portes d'aprovacio, mes baranes – i aviat l'agent es a penes mes util que un autocompletat sofisticat.

Els sandboxes resolen aixo fent el *cost del fracas* essencialment zero. I quan el fracas es barat, l'experimentacio es gratuita. Un agent en un sandbox pot provar tres enfocaments diferents a un problema, executar-los tots, i deixar que tu triis el guanyador. Aixo es un flux de treball que es impossible quan cada accio es irreversible.

7.2 Git Worktrees

Per a treball centrat en codi, els git worktrees son el sandbox mes lleuger que pots construir. Un worktree et dona una copia completa del teu repo en un directori separat, a la seva propia branca, en segons.

El flux de treball es aixi:
1. Crea un worktree per a la tasca
2. Apunta l'agent cap alla

3. Deixa'l treballar – commits, canvis de fitxers, execucions de tests, el que necessiti
4. Revisa el resultat
5. Fusiona si es bo, esborra el worktree si no

A la practica, aixo es veu aixi:

```
# Crea un espai de treball aillat per a l'agent
git worktree add ../myapp-refactor agent/refactor-auth
cd ../myapp-refactor

# L'agent treballa aqui -- completament aillat
# Quan acabi, revisa i fusiona:
cd ../myapp
git merge agent/refactor-auth

# Neteja
git worktree remove ../myapp-refactor
git branch -d agent/refactor-auth
```

Sense contenidors per construir, sense VMs per arrencar. Nomes git. Aixo es especialment potent quan executes multiples agents en paral · lel – cadascun te el seu propi worktree, la seva propia branca, el seu propi espai de treball aillat.

Tinc un alias de shell per a aixo perque ho faig servir molt sovint:

```
# In .bashrc / .zshrc
agent-sandbox() {
  local name="${1:?Usage: agent-sandbox <name>}"
  local branch="agent/$name"
  local dir="../$(basename $PWD)-$name"
  git worktree add "$dir" -b "$branch"
  echo "Sandbox ready: $dir (branch: $branch)"
}
```

7.3 Contenidors

Per a tasques que van mes enlla del codi – instal · lar paquets de sistema, executar serveis, provar canvis d'infraestructura – els contenidors son el sandbox natural. Docker et dona un entorn reproduible i aillat que pots crear en segons i destruir igual de rapid.

La clau es fer el teu projecte amigable amb contenidors. Un bon `Dockerfile` i `docker-compose.yml` ja no son nomes per a desplegament – son infraestructura d'agents. Quan el teu projecte pot arrencar en un contenidor amb una sola comanda, has donat a cada agent la capacitat de treballar en una sala neta.

Un Dockerfile minim amigable amb agents es veu aixi:

```
FROM node:22-slim
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
# L'agent pot executar tests, lint, build -- tot aillat
CMD ["npm", "test"]
```

El patro es: rapid de construir, facil de destruir, conte tot el que l'agent necessita per verificar la seva propia feina. Si el teu contenidor triga quinze minuts a construir-se, l'agent no iterara prou rapid per ser util. Optimitza per a velocitat de reconstruccio – posa les dependencies en capes, utilitza imatges base lleugeres, cacheja agressivament.

7.4 Entorns Efimers

Els sandboxes mes sofisticats son entorns efimers basats al cloud – desplegaments de previsualitzacio de curta durada que s'aixequen per branca o per tasca. Serveis com Railway, Fly.io, o entorns de desenvolupament al cloud et donen una pila completa en execucio que es completament d'un sol us.

Aixo importa per als tests d'integracio. Un agent pot desplegar els seus canvis a un entorn efimer, executar tests end-to-end contra infraestructura real, verificar que tot funciona, i llavors tu decides si promocionar-ho a staging. L'agent mai toca els teus entorns reals.

L'economia es convincent. Un entorn de previsualitzacio que funciona dues hores mentre l'agent treballa costa centims. L'incident de produccio que preveu costa milers. Les matematiques sempre afavoreixen mes sandboxing, no menys.

7.5 L'Espectre del Sandbox

Hi ha un espectre de sandboxing, i la tria correcta depen de la tasca:

Worktrees – per a canvis de codi purs. El mes rapid de crear i destruir. Sense aillament mes enlla del sistema de fitxers. Bo per a: refactoritzacions, branques de funcionalitat, escriptura de tests. Segons per configurar.

Contenidors – per a codi mes entorn. Sistema de fitxers, xarxa i processos aillats. Bo per a: canvis de dependencies, treball a nivell de sistema, qualsevol cosa que pugui contaminar la teva maquina local. Minuts per configurar.

Entorns efimers al cloud – per a verificacio de tota la pila. Infraestructura real, bases de dades reals, topologia de xarxa real. Bo per a: tests d'integracio, verificacio de desplegament, canvis multi-servei. Minuts per configurar, pero costa diners reals.

VMs – per a maxim aillament. Kernel separat, tot separat. Bo per a: treball sensible a la seguretat, agents no confiables, automatitzacio d'infraestructura. Minuts a hores per configurar.

Comenca amb worktrees. Puja a l'espectre quan la tasca ho demani. La majoria del treball agentic diari mai necessita mes que un worktree.

7.6 La Mentalitat del Sandbox

La llico mes profunda no es sobre eines – es sobre dissenyar el teu flux de treball al voltant de la disponibilitat. Si el teu entorn de desenvolupament triga una hora a configurar-se, els sandboxes son impracticables. Si triga trenta segons, son naturals.

Inverteix en una configuracio rapida i reproduible. Inverteix en infraestructura com a codi. Inverteix en fer el teu projecte facil d'arrencar de zero. Aquestes inversions es retornen cada vegada que un agent necessita un espai segur per treballar – que, a mesura que millores en enginyeria agentica, es constantment.

Una prova de lleixiu util: pot un nou desenvolupador (o un nou agent) anar de `git clone` a executar tests en menys de dos minuts? Si no, tens

deute de sandbox. Cada minut de friccio de configuracio es un minut que desincentiva el sandboxing – i agents sense sandbox son agents treballant sense xarxa.

8 Els Tests com a Bucle de Retroalimentacio

Dues bases de codi. El mateix agent. La mateixa tasca: “Afegeix un limitador de velocitat a l’API que retorni 429 despres de 100 peticions per minut per usuari.”

A la primera base de codi, no hi ha tests. L’agent llegeix els handlers de rutes, tria un punt d’insercio del middleware, escriu la logica de limitacio de velocitat, i… s’atura. No te manera de saber si ha funcionat. No pot engegar el servidor i enviar 101 peticions per veure que passa. No pot comprovar si els endpoints existents segueixen responent correctament. Produeix un diff, diu “He afegit limitacio de velocitat,” i espera que vagi be. Tu revises el codi, hi entreobres els ulls, penses que sembla raonable, el fusiones, i descobreixes en produccio tres dies despres que el middleware estava muntat en l’ordre incorrecte i mai es va executar. Totes les peticions passaven. El limitador de velocitat era decoracio.

A la segona base de codi, hi ha una suite de tests. L’agent escriu el middleware de limitacio de velocitat, despres executa els tests. Dos tests existents fallen – un test de health check que no esperava les capsaleres del middleware, i un test d’autenticacio on la configuracio del test feia 150 peticions rapides i ara queda limitat. L’agent llegeix les fallades, ajusta el middleware per saltar-se l’endpoint de health, actualitza la fixture del test per reiniciar el comptador de velocitat entre tests, i torna a executar. Verd. Despres escriu tres tests nous: un que verifica que la peticio 101 retorna 429, un que verifica que el comptador es reinicia despres d’un minut, i un que verifica que usuaris diferents tenen limits independents. Tots passen.

El mateix agent. La mateixa tasca. Resultats com la nit i el dia. La diferencia va ser la suite de tests.

En el desenvolupament tradicional, els tests verifiquen que el teu codi funciona. En enginyeria agentica, els tests fan alguna cosa mes fonamental: diuen a l'agent si ha tingut exit. Aixo canvia tot sobre com penses dels tests.

Hi ha alguna cosa inquietant en aixo al principi. La teva suite de tests – la cosa que vas escriure per verificar el *teu* codi – es converteix en l'especificacio contra la qual un agent implementa. Els tests pels quals vas suar no son nomes assegurament de qualitat. Son la definicio del que vol dir "correcte". Es una sensacio estranya: el teu esforc passat convertint-se en la base per a un flux de treball que no existia quan els vas escriure. Pero tambe es, si ho permets, profundament validador. Totes aquelles hores escrivint tests exhaustius? No eren nomes bona practica. Eren *inversio* – i els retorns estan arribant ara.

8.1 Els Ulls de l'Agent

Un agent no pot mirar una interficie i dir si es veu be. No pot sentir si una resposta d'API es "prou rapida." No pot intuir si una refactoritzacio ha preservat el comportament subtil del qual depenen els usuaris. El que *pot* fer es executar la teva suite de tests i llegir els resultats.

Els tests es converteixen en el mecanisme principal de retroalimentacio de l'agent. Verd vol dir "continua." Vermell vol dir "torna-ho a provar." Sense tests vol dir que l'agent vola a cegues – endevinant si els seus canvis funcionen, sense manera de verificar.

Per aixo les bases de codi sense tests son dificils de treballar de manera agentica. No es nomes un problema de qualitat – es un problema d'informacio. Sense tests, l'agent no te senyal. Es com demanar a algu que pengi un quadre amb els ulls embenats. Potser ho fa be, pero no hi apostaries.

I la qualitat d'aquell senyal importa enormement. Un test que diu `FAIL: expected 429, got 200` es accionable. L'agent sap exactament que ha anat malament i pot raonar sobre per que. Un test que diu `FAIL: assertion error` sense context es a penes millor que el silenci. La claredat de la sortida dels teus tests es la claredat de la visio de l'agent.

8.2 El TDD Adquireix un Nou Significat

El Test-Driven Development sempre va ser una bona idea. Amb agents, es converteix en un superpoder.

El flux de treball es simple: escriu el test primer, despres dona-li'l a l'agent i digues "fes que passi." L'agent ara te un criteri d'exit clar i inequivoc. Pot iterar – escriure codi, executar el test, llegir l'error, ajustar, repetir – en un bucle ajustat que triga segons per cicle.

Aqui tens com es veu concretament. Posem que necessites una funcio que analitzi una cadena de duracio com `"2h30m"` en total de segons. Escrius el test:

```
def test_parse_duration():
    assert parse_duration("1h") == 3600
    assert parse_duration("30m") == 1800
    assert parse_duration("2h30m") == 9000
    assert parse_duration("45s") == 45
    assert parse_duration("1h15m30s") == 4530
    assert parse_duration("") == 0
    with pytest.raises(ValueError):
        parse_duration("abc")
```

Despres dius a l'agent: "Fes que `test_parse_duration` passi."

El primer intent de l'agent podria gestionar hores i minuts pero oblidar els segons. Executa el test: `FAIL: parse_duration("45s") returned 0, expected 45`. Senyal clar. Afegeix gestio de segons, torna a executar: `FAIL: parse_duration("abc") did not raise ValueError`. Un altre senyal clar. Afegeix validacio d'entrada. Verd. Fet.

Cada cicle va trigar segons. L'agent mai va necessitar preguntar-te que vol dir "correcte" – els tests ho definien. I com que el test cobreix casos limits que vas pensar per endavant, la implementacio els gestiona des del principi, no com a bugs descoberts mes tard.

Aixo es fonamentalment diferent de demanar a un agent que "construeixi un parser de duracio." Aixo es vague. Quin format? Quins casos limits? Que hauria de passar amb entrada incorrecta? Pero "fes que aquestes set assercions passin" es precis. L'agent sap exactament com es l'exit, i pot mesurar el seu propi progres cap a ell.

L'enginyer agentic apren a expressar la intencio a traves de tests. Cada test es un contracte. Cada assercio es un requisit. Com millors son els teus tests, millor funcionen els teus agents. Deixes de pensar en escriure tests com a sobrecost i comences a pensar-ho com a *programar l'agent* – estas especificant comportament en el llenguatge mes inequivoc disponible: codi que o passa o no.

8.3 Que Fa un Bon Test Agentic

No tots els tests son igualment utils per als agents. Els tests que serveixen be als humans durant el desenvolupament podrien activament enganyar un agent. Un bon test agentic te propietats especifiques, i val la pena ser deliberat sobre elles.

Rapid. Un agent itera en un bucle. Si cada cicle triga deu minuts, l'agent prova sis enfocaments per hora. Si cada cicle triga deu segons, en prova 360. La velocitat no es nomes conveniencia – es la diferencia entre un agent que convergeix cap a una solucio i un que esgota el temps. Executa la suite completa si es rapida; executa nomes els tests rellevants si no ho es. En qualsevol cas, el bucle de retroalimentacio ha de ser ajustat.

Deterministic. Un test intermitent es pitjor que cap test per a un agent. Quan un test falla aleatoriament, un huma s'encongeix d'espatlles i torna a executar. Un agent veu una fallada i intenta arreglar-la. Canvia codi que funcionava per perseguir un fantasma. Despres el test intermitent passa – no perque el canvi de l'agent fos correcte, sino perque les estrelles aleatories es van alinear. Ara tens un canvi de codi inutil que sembla un arreglament pero no ho es. L'agent ha estat recompensat per no fer res util. Si tens tests intermitents, posa'ls en quarantena abans de donar a l'agent acces a la suite.

Aillat. Tests que depenen de l'ordre d'execucio, estat compartit, o serveis externs creen fallades desconcertants. L'agent canvia la funcio A i el test B falla – no per una dependencia real, sino perque el test B depenia d'un estat que el test A havia configurat. L'agent perdra cicles intentant entendre una relacio entre A i B que no existeix al codi, nomes a l'infraestructura de tests. Ailla els teus tests. Cadascun hauria de configurar el seu propi mon i desmuntar-lo.

Missatges d'error clars. `AssertionError: False is not True` no diu res a l'agent. `Expected user.status to be 'active' after calling activate(), but got 'pending'` li diu exactament que ha anat malament. Els bons missatges d'assercio son documentacio gratuita. Utilitza'ls. Missatges d'error personalitzats a les teves assercions son la inversio mes barata que pots fer en productivitat agentica.

Enfocats en el comportament, no en la implementacio. Un test que asserta l'estructura interna d'un valor de retorn es trenca quan l'agent refactoritza. Un test que asserta el *comportament* – "donada aquesta entrada, obtinc aquesta sortida" – sobreviu refactoritzacions i dona a l'agent llibertat per trobar millors solucions. Si els teus tests restringeixen la implementacio massa estrictament, l'agent no pot millorar-la.

La prova de foc: si mostressis nomes el fitxer de tests a un enginyer competent sense cap altre context, podria escriure una implementacio correcta? Si es que si, aquells tests funcionaran be per a un agent. Si es que no – si els tests son massa vagues, massa acoblats, o massa intermitents per servir com a especificacio fiable – arregla els tests abans de donar-los a l'agent.

8.4 La Questio de la Cobertura

"Quanta cobertura de tests necessito per a treball agentic efectiu?"

L'instint es dir 100%. Cobertura completa. Cada linia, cada branca, cada cami. Pero aixo no es ni practic ni necessari. El que realment necessites es cobertura *dirigida*: prou perque l'agent pugui verificar la cosa especifica que esta canviant.

Pensa-ho aixi. Si demanes a un agent que modifiqui el modul de processament de pagaments, necessites cobertura de tests solida del processament de pagaments. No necessites necessariament cobertura completa del sistema de plantilles d'email, el panell d'administracio, o la funcionalitat d'exportacio a CSV. L'agent necessita tests al voltant del codi que esta tocant, mes prou tests d'integracio per confirmar que no ha trencat les interficies entre sistemes.

Aixo porta a una estrategia practica: *cobreix primer els camins calents.* Mira on apuntaras realment els agents. La teva logica de negoci central. Els teus contractes d'API. Les teves transformacions de dades. Aquestes son les arees que necessiten tests rigurosos – no per objectius de qualitat abstractes, sino perque aquestes son les arees on treballaran els agents.

Les metriques de cobertura poden fins i tot ser engannyoses. Una base de codi amb 90% de cobertura de linies pero sense tests al flux de pagament es pitjor, per a proposits agentics, que una base de codi amb 40% de cobertura que testa exhaustivament pagaments, autenticacio i la capa d'API. L'agent no necessita una insignia de cobertura. Necessita tests alla on passa la feina.

Dit aixo, els tests d'integracio tenen un valor desproporcionat. Un test unitari diu a l'agent "aquesta funcio funciona aillada." Un test d'integracio diu a l'agent "aquests components funcionen junts." Quan un agent canvia una funcio, els tests unitaris detecten regressions locals i els tests d'integracio detecten efectes ondulatoris. Tots dos importen, pero si parteixes de zero, els tests d'integracio et donen mes per l'esforc perque verifiquen les *costures* – els llocs on les coses tendeixen a trencar-se.

Una cosa mes: no oblidis els tests negatius. Tests que verifiquen que el teu sistema *rebutja* entrada dolenta son critics per als agents. Sense ells, un agent pot "simplificar" la teva logica de validacio, fer que tots els tests positius passin, i deixar-te amb un sistema que accepta escombraries. Si tens una regla de validacio, testa ambdos costats.

8.5 La Velocitat Importa

Quan un agent itera en un bucle de tests, la velocitat de la teva suite de tests afecta directament la productivitat. Una suite de tests que triga deu minuts a executar-se vol dir que l'agent espera deu minuts entre intents. Una suite de tests que triga deu segons vol dir que l'agent pot provar desenes d'enfocaments en el temps que trigues a buscar cafe.

Aixo crea un incentiu fort per a:

- Mantenir els tests unitaris rapids i aillats
- Separar tests rapids de tests d'integracio lents

- Utilitzar modes de vigilancia que re-executen nomes els tests afectats
- Invertir en infraestructura de tests de la mateixa manera que inverteixes en CI

Els tests rapids ja no son nomes una millora d'experiencia del desenvolupador. Son infraestructura d'agents.

Practicament, aixo vol dir que vols una estrategia de tests en capes. Una suite unitaria rapida que s'executa en segons – el bucle intern de l'agent. Una suite d'integracio mitjana que s'executa en un minut – l'agent l'executa abans de donar la tasca per acabada. I una suite end-to-end lenta que s'executa a CI – la verificacio final abans de fusionar.

Digues als teus agents quina suite utilitzar. "Executa `pytest tests/unit` despres de cada canvi. Executa `pytest tests/integration` quan creguis que has acabat." Aixo mante el bucle intern rapid alhora que detecta problemes d'integracio abans que arribin a tu.

8.6 Testejar Mes Enlla del Codi

Els agents no nomes escriuen codi d'aplicacio. Escriuen configuracions d'infraestructura, scripts de desplegament, documentacio i mes. Cadascun d'aquests pot – i hauria de – tenir alguna forma de verificacio automatitzada.

La verificacio de tipus es el bucle de retroalimentacio mes rapid que pots donar a un agent. Un error de tipus apareix en mil · lisegons, abans que s'executi ni un sol test. En llenguatges tipats, o en Python amb mypy, o en JavaScript amb TypeScript, la verificacio de tipus detecta una classe enorme d'errors instantaniament. L'agent renomena un camp, i el verificador de tipus marca immediatament cada lloc que referencia el nom antic. Aixo es un bucle de retroalimentacio mes ajustat del que qualsevol suite de tests pot proporcionar.

El linting i el formatat detecten una altra classe de problemes. Una regla d'ESLint mal configurada podria semblar una molestia menor, pero per a un agent, una fallada de lint es un senyal inequivoc. "El teu import no es fa servir." "Aquesta variable esta declarada pero mai es llegeix." Son

correccions diminutes que s'acumulen en codi mes net sense esforc per la teva part.

La validacio d'esquemes per a contractes d'API assegura que els canvis de l'agent no trenquen la interficie entre serveis. Si tens especificacions OpenAPI, definicions de JSON Schema, o definicions de tipus GraphQL, valida contra elles. Un agent que canvia un payload de resposta apendra immediatament que ha violat el contracte, en lloc de descobrir la ruptura quan un servei downstream peti a staging.

Els tests de contracte entre serveis van mes enlla. Si el servei A depen de l'API del servei B, un test de contracte verifica que B encara satisfa el que A espera – sense necessitat d'executar ambdos serveis simultaneament. Quan un agent modifica el servei B, els tests de contracte detecten canvis trencadors que els tests unitaris dins de B passarien per alt completament.

La validacio de fitxers de configuracio esta criminalment infrautilitzada. Un lint de YAML als teus manifests de Kubernetes. Un terraform validate al teu codi d'infraestructura. Un docker-compose config check. Triguen segons a executar-se i detecten errors que d'altra manera apareixerien com a fallades misterioses durant el desplegament. Cada comprovacio automatitzada que afegeixes es un altre senyal que l'agent pot utilitzar per autocorregir-se.

L'enginyer agentic pensa sobre la testabilitat de manera amplia: no "la meva funcio retorna el valor correcte?" sino "puc verificar automaticament que aquest canvi es correcte?"

8.7 Quan els Tests Enganyen

Una suite de tests dolenta es pitjor que cap suite de tests. Aixo es incomode pero val la pena reflexionar-hi.

Sense tests, l'agent sap que vola a cegues. Sera conservador. Et dira que no pot verificar els seus canvis. Revisaras amb mes cura. La manca de senyal es almenys una manca de senyal honesta.

Amb tests dolents, l'agent vola amb falsa confianca. Fa un canvi, els tests passen, i reporta exit. Tu veus verd i relaxes la teva revisio. Pero els tests no estaven verificant realment el que tocava.

Aixo passa de diverses maneres previsibles.

Tests que testen el mock, no el codi. Quan el teu test mockeja la base de dades, el client HTTP, el sistema de fitxers i la cua – i despres asserta que la funcio mockeada es va cridar amb els arguments correctes – estas testejant la configuracio del test, no la logica de la teva aplicacio. Un agent pot fer que el codi "real" faci absolutament qualsevol cosa i el test seguira passant, mentre les expectatives del mock es compleixin. Aquests tests proporcionen un senyal verd que no significa res.

Tests massa flexibles. `assert response.status_code == 200` et diu que l'endpoint no ha petat. No et diu que la resposta contigui les dades correctes. Un agent podria retornar un cos buit, les dades de l'usuari equivocat, o una resposta a la qual li falten la meitat dels camps, i aquella assercio seguiria passant. L'especificitat a les assercions es especificitat al senyal de retroalimentacio.

Tests que dupliquen la implementacio. Si el teu test essencialment reimplementa la funcio sota test i comprova que ambdos retornen el mateix, no verifica res. L'agent pot canviar la implementacio i el test junts – i ho fara, si creu que haurien de coincidir. Acabes amb codi i tests que estan d'acord entre ells pero no amb la realitat.

Aixo connecta directament amb el patro de "Resposta Incorrecta amb Confianca" del capitol d'histories de guerra. L'agent que va afegir un `time.Sleep(500)` per arreglar una condicio de carrera? Els tests van passar. Van passar perque l'entorn de test tenia baixa concurrencia on 500ms sempre era suficient. Els tests eren *tecnicament correctes* pero *practicament engannyosos*. Van donar un senyal verd per a un arreglament que fallaria sota carrega de produccio.

La defensa es directa pero requereix disciplina. Revisa els teus tests amb el mateix rigor amb el que revises el teu codi. Pregunta: "Si l'agent introduis un bug subtil, aquest test el detectaria?" Si la resposta es no, el test es mobiliari – fa que l'habitacio sembli ocupada pero realment no fa res.

8.8 El Cercle Virtuos

Quan ho poses tot junt – bons tests, retroalimentacio rapida, entorns sandbox, i un agent al bucle – alguna cosa fa clic.

L'agent agafa una tasca. Llegeix els tests existents per entendre el comportament esperat. Fa un canvi. Executa la suite de tests rapida – deu segons. Dues fallades. Llegeix els missatges d'error, enten el problema, ajusta. Torna a executar. Verd. Escriu nous tests per al nou comportament. Executa la suite d'integracio completa – un minut. Tot verd. Commiteja amb un missatge descriptiu, i et dona un diff que saps que passa cada comprovacio automatitzada del teu sistema.

Revises un diff que saps que passa tots els tests. La teva feina canvia de "comprovar si funciona" a "comprovar si es l'enfocament correcte." Aixo es un us molt millor del teu temps, i un us molt millor de la teva experiencia. Ja no ets un executor de tests huma. Ets un arquitecte revisant dissenys.

I aqui esta l'efecte compost. Cada vegada que un agent treballa a la teva base de codi i la suite de tests l'ajuda a tenir exit, has demostrat el valor d'aquells tests. Cada vegada que un test absent causa un bug, sents el dolor immediatament – i afegeixes el test. La teva suite de tests creix exactament als llocs que importen, guiada per retroalimentacio real de treball agentic real.

Aixo es el cercle virtuos de l'enginyeria agentica: millors tests porten a agents mes autonoms, que porten a iteracio mes rapida, que et dona mes temps per escriure millors tests. Cada torn del cicle fa el seguent mes rapid.

Els enginyers que trauran mes profit de les eines agentiques no son els que tenen els prompts mes llestos o els models mes potents. Son els que tenen les millors suites de tests. Una base de codi ben testejada es un multiplicador de forca que es composa amb cada tasca que dones a un agent. Una base de codi sense tests es un impost que fa cada tasca mes lenta i arriscada.

Si t'emportes una cosa d'aquest capitol, que sigui aixo: abans d'optimitzar els teus prompts, abans d'experimentar amb nous models, abans de construir orquestracio elaborada – ves a escriure tests. Escriu-los per al

codi que els teus agents tocaran. Fes-los rapids, deterministics, aillats i especifics. Aquesta inversio rendira mes que qualsevol altra cosa d'aquest llibre.

La teva suite de tests no es sobrecost. Es la base sobre la qual es construeix tota la resta.

9 Convencio per Sobre de Configuracio

Vaig veure el mateix agent produir codi digne d'una contractacio senior en un projecte i escombraries inmantenibles en un altre – la mateixa tarda, a la mateixa maquina, amb el mateix model. La diferencia no era el prompt. Era la base de codi.

Dos projectes. El mateix stack tecnologic – TypeScript, React, PostgreSQL. El mateix tamany d'equip. Les mateixes eines agentiques.

El projecte A te un disseny de directoris estricte. Cada endpoint d'API segueix el mateix patro: un fitxer de handler, un fitxer d'esquema, un fitxer de tests, nomenats identicament. La capa de base de dades utilitza un patro de repositori consistent. Hi ha un `CLAUDE.md` a l'arrel que descriu l'arquitectura en dues pagines. Quan un enginyer apunta un agent a un tiquet – "afegeix un nou endpoint per a notificacions d'usuari" – l'agent llegeix els endpoints existents, segueix el patro, i produeix un pull request que sembla que un huma de l'equip l'hagi escrit. La revisio triga tres minuts.

El projecte B es l'altre tipus. La base de codi va creixer organicament durant dos anys. Alguns endpoints son a `routes/`, alguns a `api/`, alguns a `handlers/`. La meitat de les consultes a la base de dades utilitzen un ORM, l'altra meitat utilitzen SQL en brut. No hi ha gestio d'errors consistent – algunes funcions llencen excepcions, algunes retornen objectes d'error, algunes retornen null. L'agent mira aquesta base de codi i fa el que pot, pero "el que pot" vol dir triar el patro que hagi vist mes recentment. El codi resultant funciona, tecnicament, pero no coincideix amb res del voltant. La revisio triga trenta minuts, la majoria dedicats a "no es aixi com ho fem aqui."

La diferencia entre aquests projectes no es talent. No son les eines. Es la convencio.

Hi ha un principi antic en software: convencio per sobre de configuracio. Fes que l'opcio per defecte sigui la correcta. Redueix el nombre de decisions que cal prendre. Quan tothom segueix els mateixos patrons, el codi s'explica sol.

Aquest principi sempre va ser util per a equips humans. Per a enginyeria agentica, es essencial.

Si aixo et sona com si t'estigues dient que facis la feina poc glamurosa – escriure documentacio, imposar convencions de noms, mantenir l'estructura del projecte – aixi es. I se com se sent. No et vas fer enginyer per escriure guies d'estil. Pero aquest es un d'aquells moments on l'ofici et demana que t'importi alguna cosa que solia semblar sobrecost, perque el que hi ha en joc ha canviat. La convencio solia ser una cortesia cap al teu jo futur. Ara es el sistema operatiu sobre el qual els teus agents s'executen.

9.1 Per Que els Agents Estimen la Convencio

Un agent navegant per una base de codi desconeguda fa el mateix que un nou empleat: busca patrons. On viuen els tests? Com es nomenen els fitxers? Quina es la convencio d'imports? On es la configuracio?

Si el teu projecte segueix convencions fortes, l'agent recull els patrons rapidament i produeix codi que encaixa. Si cada fitxer es un floc de neu – nomenclatura diferent, estructura diferent, estils diferents – l'agent trontolla. No sap quin patro seguir, aixi que inventa el seu propi, i el resultat es sent alie.

La convencio es *context implicit*. Es informacio que l'agent absorbeix de l'estructura del teu codi sense que l'hagis d'explicar. Quan els teus fitxers de test sempre viuen al costat dels fitxers font que testen, nomenats `foo.test.ts` al costat de `foo.ts`, l'agent no necessita que li diguin on posar un nou test. Llegeix el directori, veu el patro i el segueix. Quan els teus handlers d'API tots exporten la mateixa forma – una funcio de handler, un esquema, un conjunt de middleware – l'agent produeix un nou handler que exporta exactament la mateixa forma.

Per aixo els frameworks amb opinions sempre han estat productius, i per que son *encara mes* productius a l'era agentica. Rails, Next.js, Laravel –

imposen una estructura. Aquella estructura no es nomes per a humans. Es un llenguatge que l'agent parla fluixament.

9.2 Aprofundiment en CLAUDE.md

Una convencio creixent en enginyeria agentica es el fitxer `CLAUDE.md`: un document a l'arrel del teu projecte que diu a l'agent el que necessita saber. No un README per a humans. Un briefing per a agents.

Aixo es una de les coses de mes alt palanquejament que pots fer pel teu flux de treball agentic, i la majoria d'equips o se'l salten o escriuen unes poques linies vagues i ho donen per fet. Parlem de com es realment un de bo.

Un `CLAUDE.md` fort te cinc seccions:

Visio general del projecte. Dues o tres frases. Que fa aquesta cosa, quin es el stack tecnologic, quin es l'objectiu de desplegament. Un agent que sap que esta treballant en "una plataforma SaaS B2B de facturacio construida amb Go i PostgreSQL, desplegada a Kubernetes" pren decisions fonamentalment diferents d'un que esta endevinant.

Comandes de build i test. Cada comanda que l'agent pugui necessitar, llistada explicitament. No "mira el Makefile" – les comandes reals. Els agents llegeixen la documentacio literalment. Si el teu `CLAUDE.md` diu `make test`, l'agent executara `make test`. Si diu "executa els tests" sense especificar com, l'agent endevinara, i podria endevinar malament.

Decisions d'arquitectura. Les coses que no son obvies a partir del codi. Per que vas triar un monorepo. Per que el servei d'autenticacio es separat. Per que fas servir event sourcing per al pipeline de comandes pero CRUD simple per a la gestio d'usuaris. Aquestes son les decisions que donen forma a cada peca de codi nou, i un agent que no les coneix les violara constantment.

Convencions. El teu estil. Com nomenes les coses. Com gestiones els errors. Com es el teu ordre d'imports. Si prefereixes retorns anticipats o condicionals niats. Les coses que fan que el codi se senti com si pertanyes a *aquest* projecte.

Trampes comunes. On estan els cossos enterrats. La migracio de base de dades que sempre s'ha d'executar abans del seed. La variable d'entorn que no esta a `.env.example` pero es necessaria per al flux de pagament. El test que es intermitent a CI pero no localment. Tots els projectes tenen aixo – escriu-ho.

Aqui tens com es veu un `CLAUDE.md` real per a un projecte de mida mitjana:

```
# Project: Meridian (billing platform)

TypeScript monorepo (pnpm workspaces). React frontend, Express
API,
PostgreSQL with Drizzle ORM. Deployed to Fly.io.

## Commands
- `pnpm install` — install all dependencies
- `pnpm test` — run all tests (vitest)
- `pnpm test:api` — API tests only
- `pnpm test:web` — frontend tests only
- `pnpm lint` — eslint + prettier check
- `pnpm lint:fix` — auto-fix lint issues
- `pnpm db:migrate` — run pending migrations
- `pnpm db:generate` — generate migration from schema changes
- `pnpm dev` — start all services locally

## Architecture
- /packages/api — Express REST API
- /packages/web — React SPA (Vite)
- /packages/shared — shared types and utilities
- /packages/db — Drizzle schema, migrations, seed data

All API routes follow the pattern:
  routes/{resource}/index.ts — route definitions
  routes/{resource}/handlers.ts — request handlers
  routes/{resource}/schemas.ts — Zod validation schemas
  routes/{resource}/__tests__/ — tests for this resource

## Conventions
- All errors go through the AppError class (packages/api/src/
errors.ts)
- Never throw raw Error objects in API handlers
- Use Zod schemas for ALL request validation, no manual checks
- Database queries live in packages/db/src/queries/, not in
```

```
handlers
- Prefer early returns over deeply nested conditionals
- Import order: node builtins, external deps, internal packages,
relative

## Pitfalls
- The Stripe webhook handler uses raw body parsing — don't add
  json middleware to that route
- Test database must be created manually: createdb meridian_test
- The `BILLING_SECRET` env var isn't in .env.example (it's in
1Password)
- Flaky test: invoice.concurrent.test.ts — known race condition,
  skip locally if it blocks you
```

Aixo no es llarg. Va trigar potser trenta minuts a escriure. Pero cada sessio d'agent que llegeix aquest fitxer comenca amb mes context que la majoria de desenvolupadors humans obtenen la seva primera setmana.

La disciplina mes important: manten-lo actualitzat. Un `CLAUDE.md` obsolet es pitjor que cap, perque l'agent se'l creura. Quan canviis una convencio, actualitza el fitxer. Quan afegeixis un nou servei, afegeix-lo a la seccio d'arquitectura. Quan algu descobreixi una nova trampa, documenta-la. Tracta'l com a codi – viu al control de versions, es revisa als PRs, es part del projecte.

Alguns equips van mes lluny: posen fitxers `CLAUDE.md` tambe als subdirectoris. Un `packages/api/CLAUDE.md` que cobreix patrons especifics de l'API. Un `packages/web/CLAUDE.md` que documenta les convencions de la biblioteca de components. Com mes profund va l'agent al projecte, mes context especific obte. Es com una ceba de documentacio – context ampli a l'arrel, context especific a mesura que aprofundeixes.

9.3 Les Convencions com a Memoria dels Agents

Les convencions son el mes semblant que tenen els agents a memoria a llarg termini. Un cop veus aixo, canvia com penses sobre totes elles.

La finestra de context d'un agent es reinicia cada sessio. No recorda que va fer ahir. No recorda la discussio d'arquitectura que vau tenir la setmana

passada. No recorda que les ultimes tres vegades que va fer servir `fetch` directament, li vas demanar que uses l'API client wrapper en el seu lloc.

Pero l'estructura del teu projecte persisteix. La nomenclatura dels teus fitxers persisteix. El teu `CLAUDE.md` persisteix. Les regles del teu linter persisteixen. Els teus patrons de tests persisteixen. Tot el que codifiques a la forma del teu projecte esta alla cada vegada que l'agent obre els ulls.

Aixo reformula les convencions completament. No son nomes sobre consistencia per a humans. Son *memoria externa* per a agents. Cada convencio que estableixis es una llico que l'agent no ha de reaprendre.

Pensa en que passa sense convencions. Sessio u: l'agent crea un nou endpoint i posa la gestio d'errors inline. Ho corregeixis a la revisio: "Fem servir la classe AppError." Sessio dos: l'agent crea un altre endpoint i comet el mateix error, perque no recorda la sessio u. Sessio tres: el mateix. Estas tenint la mateixa conversa una i altra vegada.

Ara afegeix una convencio – un patro de gestio d'errors documentat, imposat per una regla del linter – i el problema desapareix permanentment. L'agent llegeix el codi existent, veu el patro, el segueix. El linter detecta qualsevol desviacio. La llico esta codificada al propi projecte, no a la memoria de ningu.

Per aixo els equips agentics mes efectius s'obsessionen amb convencions que semblen tedioses. Ordre d'imports consistent. Nomenclatura de fitxers estricta. Signatures de funcions estandard. No son preferencies estetiques – son memoria. Son la saviesa acumulada de l'equip, emmagatzemada en un format que sobreviu els reinicis de finestra de context.

L'analogia a la qual torno una i altra vegada: les convencions son per als agents el que el coneixement institucional es per a les organitzacions. Una empresa on tot viu al cap d'una sola persona es fragil. Una empresa amb processos i documentacio forts es resilient. El mateix s'aplica a les bases de codi. Un projecte on "com fem les coses" viu nomes a la memoria del desenvolupador senior es fragil. Un projecte on esta codificat a l'estructura, les eines i la documentacio es resilient – per a humans i agents per igual.

9.4 Convencions Practiques que Ajuden els Agents

Nomenclatura de fitxers consistent. Si les teves rutes d'API viuen a `routes/`, els teus models a `models/`, i els teus tests a `__tests__/` – l'agent pot navegar el teu projecte sense mapa. Nomena els fitxers segons el que contenen. Manten-ho avorrit.

Formatadors i linters. Eines com Prettier, ESLint, `gofmt`, o Ruff no son nomes per a estil de codi – son baranes que asseguren que el codi generat per agents coincideix amb els estandards del teu projecte automaticament. Executa'ls al guardar, executa'ls a CI, fes-los no negociables.

Disseny de projecte estandard. Ja sigui el layout estandard de Go, convencions de Rails, o l'estructura del teu propi equip – tria'n un i mantin-lo. Un `justfile` o `Makefile` a l'arrel que llisti les comandes estandard (`build`, `test`, `lint`, `dev`) dona als agents un punt d'entrada a qualsevol projecte.

Fitxers petits i enfocats. Els agents treballen millor amb fitxers de menys d'unes quantes centenars de linies. Quan un fitxer conte una sola preocupacio – un component, un modul, un conjunt de funcions relacionades – l'agent pot llegir-lo, entendre'l i modificar-lo sense patir codi no relacionat.

Missatges de commit descriptius. Els agents llegeixen l'historial de git. Un commit que diu "fix bug" no ensenya res. Un commit que diu "fix race condition in session cleanup when WebSocket disconnects during auth" dona a l'agent context sobre que era important, que era fragil, i com pensa l'equip sobre els problemes.

Gestio d'errors consistent. Tria un patro i imposa'l a tot arreu. Classes d'error personalitzades amb codis d'error. Un gestor d'errors central. Una forma de resposta d'error estandard. Quan l'agent trobi un cas d'error en codi nou, hauria de tenir zero ambiguitat sobre com gestionar-lo. Si el teu projecte utilitza tres enfocaments diferents de gestio d'errors en tres fitxers diferents, l'agent produira un quart.

Formats de resposta d'API estandard. Cada endpoint retorna la mateixa forma: `{ data, error, meta }` o el que triis. Els codis d'estat

segueixen les mateixes regles a tot arreu. La paginacio funciona de la mateixa manera a cada endpoint de llista. Aixo es el tipus de consistencia en que els agents excel · leixen mantenint – pero nomes si la convencio es clara a partir del codi existent.

Convencions de logging. Logging estructurat amb camps consistents. Cada entrada de log inclou un ID de peticio, un timestamp i un nivell de gravetat. Cada log d'error inclou el codi d'error i una traca de pila. Quan l'agent afegeixi logging a codi nou – i hauria de fer-ho – els logs haurien de ser identics a tots els altres logs del sistema.

Estructura de directoris que expliqui una historia. Un agent hauria de poder fer `ls` al nivell superior del teu projecte i entendre l'arquitectura. Aqui tens com es veu un projecte ben estructurat des de la perspectiva d'un agent:

```
src/
  routes/         # HTTP handlers — one file per resource
  services/       # Business logic — one file per domain concept
  repositories/   # Database access — one file per table/entity
  middleware/      # Express/Koa middleware
  utils/          # Pure utility functions
  types/          # Shared TypeScript types
  errors/         # Custom error classes
tests/
  fixtures/       # Test data factories
  helpers/        # Test utilities
migrations/       # Database migrations (numbered)
scripts/          # One-off and maintenance scripts
```

Cada nom de directori es un substantiu. Cada fitxer dins conte exactament el que el nom del directori promet. No hi ha lloc per a la confusio sobre on hauria d'anar el codi nou. Un agent llegint aquesta estructura immediatament sap: "Necessito afegir una nova consulta a la base de dades, aixo va a `repositories/`. Necessito un nou endpoint, aixo comenca a `routes/`."

Compara-ho amb un projecte on les consultes a la base de dades estan escampades pels fitxers de handlers, la logica de negoci viu en funcions d'utilitat, i hi ha tres directoris diferents que tots semblen contenir "helpers." L'agent posara codi en *algun* lloc, pero probablement no sera el lloc *correcte*.

9.5 L'Impost de la Convencio

Siguem honestos sobre el cost. Establir convencions pren temps, i se sent com a sobrecost – especialment al principi.

Escriure un `CLAUDE.md` pren una tarda. Configurar linters i formatadors pren un dia. Refactoritzar una base de codi inconsistent per seguir un sol patro pren una setmana, potser mes. Imposar convencions a la revisio de codi vol dir frenar PRs que "funcionen be" pero no segueixen l'estandard.

Hi ha una temptacio real de saltar-se tot aixo. El codi funciona. Els tests passen. Per que dedicar temps a convencions de noms quan hi ha funcionalitats per lliurar?

La resposta honesta: si ets un desenvolupador en solitari construint un prototip d'un sol us, l'impost de la convencio probablement no val la pena. Mou-te rapid, desplega-ho, llenca-ho.

Pero al moment que un segon parell d'ulls toqui la teva base de codi – huma *o* agent – les convencions comencen a retornar-se. I els retorns es composen.

La primera vegada que estableixes una convencio, et costa una hora. Cada vegada posterior que un agent segueix aquella convencio en lloc de preguntar-te com gestionar-ho, estalvies cinc minuts. Fes l'aritmetica: despres de dotze sessions d'agents, la convencio s'ha amortitzat. Despres de cent sessions, has estalviat *hores*.

Aixo es composa a l'equip. Cinc enginyers, cadascun executant multiples sessions d'agents per dia, tots beneficiant-se de les mateixes convencions. La inversio inicial d'una tarda d'un enginyer estalvia a l'equip centenars d'hores al llarg d'un any.

Els equips que inverteixen en convencions d'hora semblen mes lents al principi. Dediquen temps a coses "avorrides" – configuracions de linter, estructura del projecte, documentacio. Pero tres mesos despres, els seus agents estan produint codi que requereix revisio minima. Sis mesos despres, estan lliurant el doble de rapid que l'equip que es va saltar la feina de convencions. Un any despres, la diferencia es vergonyosa.

Aixo es la mateixa dinamica que el deute tecnic, nomes invertida. La convencio es *riquesa* tecnica – i com la riquesa financera, com mes aviat comences a invertir, mes dramatic es el compost.

L'error mes gran que veig es equips que esperen fins que la seva base de codi es un desastre abans d'intentar establir convencions. Aquell es el moment mes car per fer-ho. El moment mes barat es al principi d'un projecte – pero el segon moment mes barat es avui.

9.6 L'Efecte Compost

Cada convencio que estableixes, cada estandard que imposes, cada peca de documentacio que mantens – tot es composa. Cadascun fa els agents lleugerament mes efectius, lleugerament mes autonoms, lleugerament mes propensos a produir codi que encaixa al teu projecte com un guant.

Pero el compost no es nomes additiu. Les convencions interactuen. Una convencio de nomenclatura de fitxers consistent *mes* una estructura de directoris estandard *mes* un `CLAUDE.md` que descriu l'arquitectura – juntes, donen a l'agent un model mental de tot el projecte. Sap on trobar les coses, com nomenar-les, i com encaixen. Elimina qualsevol d'aquestes tres, i l'efectivitat de l'agent cau desproporcionadament.

Per aixo les convencions a mitges son gairebe tan dolentes com no tenir convencions. Un projecte amb nomenclatura de fitxers consistent pero estructura de directoris caotica envia senyals mixtos. L'agent veu ordre en una dimensio i caos en una altra, i no sap en quin senyal confiar.

Els enginyers que inverteixen en convencio d'hora no nomes tenen bases de codi mes netes. Tenen bases de codi que estan preparades per al futur agentic. Els seus agents produeixen millors resultats. Les seves revisions son mes rapides. Els seus cicles d'iteracio son mes ajustats.

La convencio no es feina glamurosa. Es el capitol menys emocionant d'aquest llibre. Pero es el que fa que tots els altres capitols funcionin. Els sandboxes, les estrategies de testing, l'orquestracio multi-agent – tot funciona millor quan la base de codi subjacent es consistent, documentada i convencional.

La teva base de codi es l'entorn on viuen els teus agents. Fes-la llegible. Fes-la previsible. Fes-la avorrida. Els agents t'ho agrairan escrivint codi que sembla que hi pertanyi.

10 LLMs Locals vs. LLMs Comercials

Un amic meu – enginyer senior a una startup de Serie B – em va enviar un missatge un divendres a la tarda. "Acabo de rebre la nostra primera factura real de l'API. Dos mil dos-cents dolars. Pel *marc*." Havia estat executant Claude Code a tot el seu equip de vuit enginyers, cadascun d'ells iterant en funcionalitats, depurant, refactoritzant. Ningu havia configurat pressupostos de tokens. Ningu vigilava el comptador. Els agents funcionaven de meravella, i la factura feia mal als ulls.

Aquella mateixa setmana, vaig parlar amb una enginyera d'una empresa sanitaria a Munich. Executava Llama 70B en un servidor GPU local perque el seu pipeline de dades de pacients no podia tocar APIs externes. No "no hauria de" – *no podia*. El seu equip de compliment normatiu ho havia deixat clar per escrit. Obtenia resultats decents per a tasques enfocades, pero cada vegada que necessitava raonament complex multi-fitxer, el model fallava i es trobava fent la feina manualment.

Aquestes dues histories son els extrems de la mateixa pregunta: on s'executa el model? Sona com una decisio d'infraestructura. Realment es una decisio sobre confianca, cost, capacitat, i – cada vegada mes – com arquitectures tot el teu flux de treball agentic.

10.1 LLMs Comercials: La Frontera

Els models comercials – Claude, GPT, Gemini – son on viu la frontera. Si estas fent enginyeria agentica seria, probablement has passat la majoria del teu temps aqui. Hi ha bones raons per aixo.

10.1.1 Finestres de Context

El context ho es tot per als agents. Un agent no nomes llegeix el teu prompt – llegeix fitxers, sortides d'eines, missatges d'error, resultats de

tests, i el seu propi raonament anterior. Una sola tasca complexa pot omplir facilment 50.000 tokens de context, i una sessio de depuracio multi-pas pot superar els 100.000.

Els models comercials de frontera ofereixen finestres de context de 128K tokens i mes. Aixo importa enormement per al treball agentic perque l'agent necessita mantenir la teva base de codi al cap. Quan el context s'esgota, l'agent comenca a oblidar fitxers anteriors que va llegir, decisions anteriors que va prendre, errors anteriors que va veure. Es degrada d'un enginyer capac a algu amb amnesia.

Els models locals generalment arriben a un maxim de 8K-32K de context a la practica, fins i tot si tecnicament suporten mes sobre el paper. La qualitat de l'atencio es degrada a mesura que empenys cap al limit. Un model comercial a 100K de context segueix raonant be. Un model local a 32K de context sovint perd el fil.

10.1.2 Qualitat d'Us d'Eines

Els agents viuen i moren per l'us d'eines. Llegir fitxers, escriure codi, executar comandes, buscar a bases de codi – no son extres opcionals. Son el bucle central. I la qualitat de l'us d'eines varia dramaticament entre models.

Els models comercials de frontera han estat especificament entrenats i ajustats per a crides d'eines. Formaten arguments correctament, encadenen crides d'eines logicament, es recuperen graciosament quan una crida d'eina falla. Saben quan llegir un fitxer abans d'editar-lo. Saben quan executar tests despres de fer canvis.

Els models mes petits – incloent la majoria de models locals – son menys fiables aqui. Al · lucinaran camins de fitxers, oblidaran passar arguments requerits, cridaran eines en l'ordre incorrecte, o es quedaran atrapats en bucles cridant la mateixa eina fallida una i altra vegada. La diferencia no es subtil. En una tasca complexa que involucra deu o quinze crides d'eines, un model de frontera podria tenir exit el 90% de les vegades on un model local te exit el 40%.

10.1.3 Seguiment d'Instruccions

Quan dius a un model de frontera "nomes modifica fitxers al directori `src/auth/`" o "no canviis l'API publica, nomes la implementacio interna,"

generalment escolta. Segueix system prompts, respecta restriccions, i es mante dins dels limits.

Els models mes petits deriven. Comencaran be, despres gradualment ignoraran les teves restriccions a mesura que la conversa es fa mes llarga i el context s'omple. Aixo es un problema real per al treball agentic, on la precisio importa. Un agent que "majoritariament" segueix instruccions ocasionalment reescriura un fitxer que no li has demanat que toqui, o refactoritzara alguna cosa que explicitament li has dit que deixi. En un entorn sandbox aixo es nomes molest. Sense sandbox es perillos.

10.1.4 Triant el Model Comercial Correcte

No tots els models comercials son intercanviables, ni tan sols a la frontera. Aqui tens el que he trobat a la practica:

Per a refactoritzacions complexes multi-fitxer i treball arquitectonic – utilitza el model mes capac que et puguis permetre. Aqui es on la qualitat del raonament importa mes. La diferencia de cost entre un model de gamma mitjana i un de gamma alta es uns quants dolars per tasca. La diferencia de qualitat sovint es la diferencia entre un diff net i un embolic que has de refer manualment.

Per a tasques enfocades d'un sol fitxer – escriure tests, implementar una funcio ben definida, arreglar un bug clar – els models de gamma mitjana funcionen gairebe tan be com els de gamma alta, a una fraccio del cost. La tasca es prou acotada perque el model no necessita fer malabars amb moltes preocupacions alhora.

Per a treball d'alt volum i baixa complexitat – generar boilerplate, formatar codi, escriure missatges de commit – el model mes barat que pugui seguir instruccions es la tria correcta. Executaras aquestes tasques centenars de vegades. L'estalvi per token es composa.

L'error que veig mes sovint es utilitzar un sol model per a tot. Aixo es com conduir un Formula 1 al supermercat. Fes coincidir el model amb la tasca.

10.2 LLMs Locals: La Imatge Completa

Executar un model localment – Llama, Mistral, Qwen, DeepSeek, Codestral, o qualsevol dels models de pesos oberts via Ollama, llama.cpp, o vLLM – et dona alguna cosa que les APIs comercials no poden: sobirania de dades completa i zero cost marginal per token.

El teu codi mai surt de la teva maquina. Pots alimentar-lo amb codi font propietari, documents interns, logs de produccio, dades de clients, claus d'API que has deixat accidentalment en una configuracio – res d'aixo creua un limit de xarxa. I un cop el model esta descarregat, cada inferencia es gratuita. Executa'l mil vegades, executa'l tota la nit, ningu t'envia una factura.

Pero siguem honestos sobre l'experiencia, perque el marketing al voltant dels models locals sovint no ho es.

10.2.1 La Realitat del Hardware

La questio del hardware es la primera que tothom pregunta, i la resposta es menys glamurosa del que els articles de blog de "executa IA localment!" suggereixen.

MacBook Pro amb Apple Silicon (M2/M3/M4 Pro o Max, 32-64GB de RAM). Aixo es el punt optim per a desenvolupadors individuals. Pots executar un model de 7B-14B parametres comodament, i un model de 32B si tens la RAM. La inferencia sera notablement mes lenta que una API comercial – potser 15-30 tokens per segon per a un model de 14B, comparat amb 80+ d'una API comercial. Un model de 70B tecnicament funcionara en una maquina de 64GB pero sera prou lent per posar a prova la teva paciencia. Esperaras 10-20 segons per respostes que triguen 2 segons des d'una API.

Servidor GPU dedicat (NVIDIA RTX 4090 o A100). Aqui es on la inferencia local es torna genuinament rapida. Una 4090 amb 24GB de VRAM pot executar un model de 14B a velocitats properes a les comercials. Una A100 amb 80GB pot executar un model de 70B comodament. Pero ara estas mantenint hardware. Estas lluitant amb drivers de CUDA, gestio de VRAM, i l'ocasional "per que crida el ventilador de la meva GPU a les 3 de la matinada."

Hardware de consum (MacBook Air de 16GB, maquines antigues, sense GPU dedicada). Estas limitat a models de 7B, i l'experiencia sera mediocre. La inferencia es lenta, els models son massa petits per a treball agentic seri, i passaras mes temps esperant que treballant. No ho recomanaria per a res mes enlla de l'experimentacio.

El resum honest: els models locals son practics per a desenvolupadors amb un MacBook Pro recent o una GPU dedicada. Per sota d'aquell llindar de hardware, tindras una experiencia frustrant. Per sobre, tindras una eina genuinament util – nomes una eina diferent d'una API comercial.

10.2.2 On Brillen els Models Locals

Els models locals no son simplement "models comercials pitjors." Hi ha fluxos de treball on genuinament tenen mes sentit:

Tasques d'alta frequencia i baix risc. Generar docstrings, escriure missatges de commit, crear codi boilerplate, formatar dades. Aquestes tasques no necessiten un model geni – necessiten un model rapid i gratuit. Executar-les localment vol dir que pots disparar-i-oblidar sense pensar en el cost.

Bases de codi sensibles. Ho cobrirem mes a la seccio de privacitat, pero aixo es un cas d'us legitim i creixent. Si el teu codi no pot sortir de la teva xarxa, els models locals son l'unica opcio, i "prou bo" es infinitament millor que "no disponible."

Desenvolupament offline. En un avio, en un tren, en un bunquer – el teu model local funciona sense WiFi. Aixo sona menor fins que estas en un vol de dotze hores intentant depurar alguna cosa.

Experimentacio i aprenentatge. Quan estas experimentant amb frameworks d'agents, provant estrategies de prompts, o construint integracions d'eines personalitzades, cremar credits d'API en assaig-error se sent malbaratat. Un model local et permet iterar lliurement.

10.2.3 On Pateixen els Models Locals

Val la pena ser especific sobre els modes de fallada, perque "es menys capac" es massa vague per ser util.

Raonament multi-fitxer. Demana a un model local de 14B que segueixi un bug a traves de quatre fitxers i un esquema de base de dades, i perdra el

fil. Podria trobar el fitxer correcte pero malinterpretar la interaccio entre components. Aquesta es la diferencia practica mes gran.

Tasques de llarg recorregut. Les tasques agentiques que involucren molts passos – llegir, planificar, implementar, testejar, depurar, iterar – requereixen que el model mantingui una intencio coherent a traves d'un context llarg. Els models locals tendeixen a derivar. Obliden el pla. Revisiten decisions que ja havien pres. Es queden atrapats en bucles.

Revisio de codi matisada. "Aquest codi es correcte pero l'enfocament es equivocat" es una decisio de judici que requereix comprensio profunda. Els models locals tendeixen cap a l'analisi superficial – detectaran problemes de sintaxi i bugs obvis pero passaran per alt problemes arquitectonics.

Orquestracio d'eines complexa. Quan una tasca requereix deu o mes crides d'eines encadenades – llegir un fitxer de test, executar-lo, llegir l'error, trobar el fitxer font, entendre el context, fer un canvi, tornar a executar el test – els models locals tropessen mes frequentment a cada pas, i la taxa d'error es composa.

Res d'aixo vol dir que els models locals siguin inutilitzables. Un model de 32B executant-se localment pot gestionar un rang sorprenent de tasques ben acotades. Pero necessites acotar les tasques en consequencia. Demanar a un model local que faci el que fa un model de frontera es preparar-lo per al fracas.

10.3 El Cost del Treball Agentic

Una sola sessio de codi agentic pot consumir facilment 100.000-500.000 tokens. Aixo sorpren la gent quan veuen els numeros per primera vegada. No perque estiguis xatejant – perque l'agent esta *treballant*.

Considera que passa quan un agent aborda un arreglament de bug moderadament complex. Llegeix tres o quatre fitxers per entendre el context (potser 8.000 tokens d'entrada). Raona sobre el problema (2.000 tokens de sortida). Llegeix dos fitxers mes (4.000 tokens). Escriu un arreglament (1.000 tokens). Executa els tests (crida d'eina, 500 tokens de sortida). Els tests fallen. Llegeix l'error (1.000 tokens). Revisa l'arreglament (1.500 tokens). Torna a executar els tests. Passen. Llegeix el diff per verificar

(1.000 tokens). Total: potser 25.000 tokens per a un arreglament de bug senzill.

Ara considera una tasca de refactoritzacio complexa. L'agent podria llegir vint fitxers, generar un pla, implementar canvis a vuit fitxers, executar la suite de tests quatre vegades, depurar dues regressions, i escriure tests nous. Aixo son facilment 200.000 tokens. Als preus de models de frontera, aixo es entre $3 i $15 depenent del model i la ratio entrada/sortida.

Aqui tens rangs de cost aproximats que he vist a la practica, basats en l'us de models comercials de frontera:

- **Arreglament de bug simple o funcionalitat petita:** $0.30-$1.00
- **Funcionalitat moderada amb tests:** $2-$5
- **Refactoritzacio complexa multi-fitxer:** $5-$15
- **Implementacio de funcionalitat gran:** $15-$40+
- **Sessio de depuracio exploratoria que s'allarga:** $10-$30

Multiplica aquests numeros per un equip d'enginyers, cadascun executant diverses tasques agentiques per dia, i estas mirant diners reals. La factura de $2.200 del meu amic? Es mes o menys correcta per a vuit enginyers actius.

10.3.1 Estrategies per Gestionar el Cost

La solucio no es deixar d'utilitzar agents. Es ser intel · ligent al respecte.

Configura pressupostos de tokens. La majoria de frameworks d'agents et permeten configurar un limit maxim de tokens per tasca. Aixo no es nomes control de costos – tambe es un senyal de qualitat. Si un agent crema 500.000 tokens en una tasca que n'hauria de trigar 50.000, alguna cosa ha anat malament. L'agent esta atrapat, fent bucles, o malinterpretant la tasca. Un pressupost el forca a fallar rapid en lloc d'espiralar.

Utilitza routing de models. No enviis cada tasca al model mes car. Aprofundirem en aixo a la propera seccio, pero la versio curta: utilitza un model barat i rapid per a exploracio i lectura de codi, i un model car i capac per a raonament i implementacio. Aixo sol pot reduir costos un 50-70%.

Cacheja agressivament. Si el teu framework d'agents suporta cache de prompts, activa-ho. Molt del context d'un agent es repeteix entre torns

– el mateix system prompt, el mateix context del projecte, els mateixos fitxers llegits recentment. El caching evita re-processar aquells tokens a cada torn.

Acota les tasques estretament. Una tasca ben acotada ("arregla el bug de zona horaria a `billing/invoice.py`, el test es a `tests/test_invoice.py`") es mes barata que una de vaga ("arregla els bugs de facturacio"). Acotar no es nomes bona enginyeria – es control de costos. L'agent llegeix menys fitxers, fa menys crides d'eines exploratories, i convergeix mes rapid.

Revisa els teus fracassos. Quan una tasca falla o consumeix massa tokens, esbrina per que. El prompt era vague? Faltava context a l'agent? El model no era prou capac per a la tasca? Cada fracas es una oportunitat d'ajustament.

10.3.2 Gestionar Costos a Traves d'un Equip

El control individual de costos es una cosa. Gestionar la despesa a traves d'un equip de vuit o dotze enginyers, cadascun executant agents tot el dia, es un problema completament diferent. Es la diferencia entre vigilar la teva propia dieta i gestionar una cuina de restaurant.

Pressupostos per enginyer. Configura un pressupost mensual de tokens per enginyer – no per restringir, sino per fer els costos visibles. Quan tothom pot veure la seva propia despesa, el comportament canvia naturalment. La gent comenca a acotar tasques mes curosament, a triar el model correcte per a la feina, a matar sessions desbocades mes aviat. Un punt de partida raonable: $200-400 al mes per enginyer per a un equip que utilitza activament agents. Alguns enginyers vindran consistentment per sota del pressupost. Altres tindran pics durant setmanes intenses de depuracio. Aixo esta be – la qüestio es la consciencia, no l'aplicacio.

Dashboard i visibilitat. No pots gestionar el que no pots veure. Fes seguiment de la despesa per enginyer, per projecte, per tipus de tasca. La majoria de proveiors d'APIs et donen desglossos d'us per clau d'API, i si assignes claus per enginyer o per projecte, les dades ja hi son. Canalitza-ho cap a un dashboard que l'equip pugui veure – fins i tot un full de calcul compartit funciona per comencar. L'enginyer que descobreix que va cremar $80 en una sola sessio de depuracio naturalment comencara a

acotar tasques mes estretament la propera vegada. No necessites tenir-ne una conversa. El numero parla sol.

Alertes i interruptors automatics. Configura alertes al 50% i 80% del pressupost mensual perque ningu es porti sorpreses. Mes important, configura un limit dur de tokens per sessio com a interruptor automatic. Si una sola sessio d'agent supera els 500K tokens, alguna cosa ha anat malament – l'agent esta fent bucles, la tasca es massa vaga, o el model esta lluitant amb alguna cosa mes enlla de la seva capacitat. Mata-la i investiga. Aixo es el que preveu l'escenari de "factura sorpresa de $2.200" del principi d'aquest capitol. No necessites detectar cada sessio desbocada manualment. Els limits automatitzats fan la feina.

La conversa del ROI. En algun moment, algu de la direccio preguntara per que la factura de l'API es de cinc xifres. Necessites les matematiques preparades. Emmarca-ho aixi: un enginyer senior costa $150K a l'any completament carregat. Si els agents el fan un 30% mes productiu, son $45K de valor. Els costos dels agents son $3-4K a l'any per enginyer. El ROI es aproximadament 10x. Fins i tot si ets conservador – diguem un 15% de guany de productivitat en lloc de 30% – els numeros segueixen funcionant comodament. La part mes dificil es mesurar aquell guany de productivitat amb precisio. Probablement no pots, no amb rigor cientific. Pero pots fer seguiment de senyals concretes: temps fins a fusionar PRs, nombre de tasques completades per sprint, reduccio del canvi de context. Combina aquelles metriques amb les dades de cost i tens una historia amb la qual finances pot treballar.

El cost com a senyal de qualitat. Un consum alt de tokens en una tasca no es nomes car – es un senyal que alguna cosa ha anat malament. La tasca estava mal acotada, el context era insuficient, o el model no era prou capac per a la feina. Comenca a fer seguiment del cost per tipus de tasca al llarg del temps. Si els costos d'una categoria particular tendeixen a l'alca, investiga – els teus prompts poden haver derivat, o la base de codi s'ha tornat prou complexa per necessitar un enfocament diferent. Si els costos tendeixen a la baixa, es que el teu equip esta millorant en enginyeria agentica. Estan escrivint prompts mes ajustats, triant millors models, i acotant tasques mes efectivament. Les dades de cost, ben utilitzades, es converteixen en un mirall de l'habilitat de l'equip.

10.4 Privacitat i Compliment Normatiu

Hi ha codi que genuinament no pot sortir de l'edifici. Aixo no es paranoia – es llei.

Contractistes governamentals treballant en projectes classificats o de control d'exportacio no poden enviar codi font a APIs de tercers, punt. Els requisits de residencia de dades no son suggeriments. Venen amb penalitats criminals.

Empreses sanitaries que gestionen dades de pacients estan vinculades per HIPAA, GDPR, o regulacions equivalents. Si el teu codi toca registres de pacients – fins i tot fixtures de test amb dades falses realistes que un responsable de compliment podria mirar amb desconfianca – necessites pensar curosament sobre que envia per la xarxa.

Les institucions financeres tenen el seu propi laberint de regulacions. Compliment SOX, PCI-DSS, requisits d'auditoria interna – els detalls varien, pero el tema es consistent: les dades es queden dins de limits controlats.

I despres hi ha la simple competitivitat i secret. Els teus algoritmes propietaris, la teva salsa secreta, el teu avantatge competitiu – enviar-ho als servidors d'algu altre requereix confiar que no ho entrenaran, no ho registraran, no seran vulnerats. La majoria de proveiors d'APIs comercials ofereixen garanties contractuals fortes al respecte. Pero "garanties contractuals fortes" i "impossible de vulnerar" son coses diferents, i alguns equips de seguretat no estan disposats a acceptar la diferencia.

Per a tots aquests casos, els models locals no son un luxe. Son l'unica opcio.

L'equilibri es real: estas acceptant capacitat del model reduida a canvi de control absolut de les dades. Un model local de 32B fent una feina mediocre a la teva base de codi classificada es infinitament mes util que un model de frontera que no tens permes d'utilitzar. I per a tasques enfocades i ben acotades – el tipus que hauries d'estar escrivint de totes maneres – la diferencia de qualitat sovint es menor del que esperaries.

10.4.1 El Terme Mig: Desplegaments Privats

Val la pena mencionar que hi ha un cami intermedi emergent. Els proveiors de cloud ara ofereixen desplegaments de models privats – instancies dedicades de models de frontera executant-se dins del teu VPC, amb garanties contractuals que les teves dades es queden dins el teu perimeter i no s'utilitzen per a entrenament. AWS Bedrock, Azure OpenAI, Google Cloud Vertex AI tots ofereixen versions d'aixo.

No son barats. Estas pagant per compute dedicat, no per infraestructura d'API compartida. Pero per a grans organitzacions que necessiten capacitat de frontera i control estricte de dades, aixo es cada vegada mes la resposta. Obtens qualitat de model comercial amb garanties de privacitat de model local – per un preu.

10.5 Model Routing a la Practica

La pregunta real no es "local o comercial?" Es "quin model, per a quina part del flux de treball?"

Una configuracio agentica sofisticada encamina parts diferents del flux de treball a models diferents. Aixo no es teoric – equips ho estan fent avui, i es la direccio cap a la qual es mou l'ecosistema.

10.5.1 El Patro de Routing

Pensa en el que un agent realment fa durant una tasca tipica:

1. **Exploracio** – llegir fitxers, buscar a la base de codi, entendre l'estructura. Aixo es treball d'alt volum i baix raonament. El model necessita decidir quins fitxers llegir i extreure informacio rellevant, pero no esta fent pensament profund.

2. **Planificacio** – analitzar el problema, considerar enfocaments, decidir una estrategia. Aixo requereix raonament fort. Es la part on la qualitat del model importa mes.

3. **Implementacio** – escriure els canvis de codi reals. Aixo requereix bona generacio de codi i la capacitat de seguir el pla del pas 2.

4. **Verificacio** – executar tests, llegir errors, decidir si la feina esta feta. Raonament moderat, us intens d'eines.

5. **Iteracio** – si la verificacio falla, tornar enrere i ajustar. Aixo requereix entendre l'error i connectar-lo amb la implementacio.

No tots aquests passos necessiten el mateix model. El pas 1 pot ser gestionat per un model petit, rapid i barat – fins i tot un de local. Esta llegint fitxers i reportant que hi ha. El pas 2 es on vols el model de frontera – aquest es el raonament car que justifica el cost de l'API. Els passos 3-5 sovint poden ser gestionats per un model de gamma mitjana, ja que el pensament dificil ja esta fet i el model esta executant un pla.

10.5.2 Com Es Veu Aixo

A la practica, el routing de models pot ser tan simple com configurar el teu framework d'agents per utilitzar models diferents per a tipus de tasques diferents:

```
# Pseudocode — the exact config depends on your framework
exploration_model: "local/qwen-14b"       # Free, fast, good
enough for reading
reasoning_model: "claude-sonnet"           # Frontier reasoning
for planning
implementation_model: "claude-haiku"       # Fast, cheap, follows
plans well
```

O pot ser dinamic – un classificador lleuger que mira el pas actual i encamina en consequencia. Alguns frameworks estan comencant a construir-ho de manera nativa. Altres requereixen que ho cablegis tu mateix.

L'economia es convincent. Si el 60% dels tokens d'un agent es gasten en exploracio i tasques simples, i els encamines a un model que es 10x mes barat, has reduint el teu cost total en mes de la meitat sense cap reduccio de qualitat a les parts que importen.

10.5.3 Routing per a Privacitat

El routing de models tambe resol el problema de privacitat de manera mes elegant que un enfocament de tot o res.

Posem que estas construint una aplicacio sanitaria. Els models de dades i la logica de negoci toquen dades de pacients – aixo ha de quedar-se local. Pero els components de frontend, la configuracio de build, el pipeline de CI? No contenen dades sensibles. No hi ha cap rao per la qual no puguis

utilitzar un model comercial de frontera per a les parts no sensibles mentre encamines el treball sensible a un model local.

Aixo requereix eines que siguin conscients dels limits de sensibilitat – quins fitxers poden sortir a l'exterior, quins no. Aquella eina encara esta madurant, pero el patro es clar. En lloc de "tot local" o "tot comercial," obtens "local on importa, comercial a la resta."

10.6 Comencant Sense Pagar una Fortuna

No necessites un pressupost per comencar a aprendre enginyeria agentica. Necessites un portatil i un vespre. Aqui tens una rampa practica de zero cost a productivitat real.

10.6.1 La Capa Gratuita

La majoria de proveiors comercials ofereixen capes gratuites o credits de prova. A principis de 2026, pots comencar amb Claude, GPT, o Gemini sense introduir una targeta de credit. Les capes gratuites estan limitades en taxa i context, pero son mes que suficients per aprendre els fonaments – escriu el teu primer prompt agentic, mira un agent iterar sobre un test que falla, aconsegueix una sensacio del bucle de retroalimentacio.

Combina una clau d'API de capa gratuita amb un framework d'agents de codi obert – la capa gratuita de Claude Code, o Aider amb el seu suport de models gratuitis – i tens una configuracio agentica completa a zero cost. No sera rapida. No gestionara treball complex multi-fitxer. Pero t'ensenyara tot dels capitols dos a cinc d'aquest llibre sense gastar ni un centric.

10.6.2 El Cami Local-Primer

Si tens un MacBook Pro amb 16GB o mes de RAM, pots executar models utils localment de franc, per sempre.

Instal · la Ollama – triga una comanda. Descarrega un model – `ollama pull qwen2.5-coder:14b` triga uns minuts. Apunta un framework d'agents cap a ell. Ara tens una configuracio de codi agentic sense costos d'API, sense limits de taxa, i sense dades sortint de la teva maquina.

L'experiencia no igualara un model comercial de frontera. Les finestres de context son mes petites. El raonament multi-fitxer es mes feble. L'us d'eines es menys fiable. Pero per a tasques enfocades i ben acotades – "escriu tests per a aquesta funcio," "refactoritza aquesta classe per fer servir injeccio de dependencies," "afegeix gestio d'errors a aquest endpoint" – un model local de 14B es sorprenentment capac. I com que es gratuit, pots iterar sense vigilar el comptador.

Un stack de partida practic per a enginyeria agentica de zero cost:

- **Ollama** per servir models (gratuit, local)
- **Qwen 2.5 Coder 14B** o **DeepSeek Coder V2** per a tasques de codi
- **Aider** o **Claude Code** (amb suport de model local) com a framework d'agents
- Un projecte amb una suite de tests (l'agent necessita retroalimentacio)

Ja esta. Sense subscripcio. Sense clau d'API. Sense compute al cloud. Arribaras al sostre eventualment – una tasca que necessita una finestra de context mes gran, o raonament multi-fitxer que el model local no pot gestionar. Llavors es quan agafes un model comercial. Pero la configuracio gratuita et portara mes lluny del que esperes.

10.6.3 El Punt Optim: $20/Mes

Quan estiguis preparat per gastar diners, la configuracio mes cost-efectiva que he trobat es una combinacio de models locals per a exploracio i una subscripcio comercial per a raonament.

La majoria de proveiors comercials ofereixen un pla de desenvolupador al voltant dels $20/mes que inclou una assignacio generosa de tokens. Utilitza'l per a les tasques que realment necessiten capacitat de frontera – depuracio complexa, refactoritzacio multi-fitxer, planificacio arquitectonica. Utilitza el teu model local per a tota la resta – llegir codi, generar boilerplate, escriure missatges de commit, iterar a traves de cicles de tests.

Aquest enfocament hibrid tipicament cobreix el 80-90% del treball agentic d'un desenvolupador individual. El model local gestiona les tasques d'alt volum i baix raonament. El model comercial gestiona els moments que necessiten intel·ligencia genuina. La teva factura mensual es manté previsible, i no estas triant entre qualitat i cost – estas utilitzant cadascun on te sentit.

10.6.4 Escalant Amb Intencio

L'error es comencar amb l'opcio mes cara i optimitzar despres. Comenca gratuit. Apren els fluxos de treball. Enten on la qualitat del model realment importa i on "prou bo" es prou bo. Despres gasta diners a les brecces especifiques que les eines gratuites no poden omplir.

Quan estiguis gastant diners reals en crides d'API, sabras exactament per que estas pagant – i mes important, per que *no* estas pagant. Aquell coneixement val mes que qualsevol quantitat de credits.

10.7 El Paisatge Esta Canviant

D'aqui sis mesos, els detalls d'aquest capitol estaran desfasats. Els numeros de cost canviaran. Les diferencies de capacitat s'escurcaran. Sortira un nou model que reorganitzara el ranking. Aquesta es l'unica prediccio que fare amb confianca.

El que no canviara es el marc per pensar-hi. Seguiras necessitant avaluar models al llarg dels mateixos eixos: capacitat, cost, privacitat, velocitat i fiabilitat. Seguiras necessitant fer coincidir el model amb la tasca en lloc de triar un model i fer-lo servir per a tot. Seguiras necessitant ser flexible.

Els enginyers que veig fent la millor feina no son lleials a cap model o enfocament de desplegament particular. Son pragmatics. Utilitzen el model comercial de frontera quan la tasca ho demana, un model de gamma mitjana quan es prou bo, i un model local quan la privacitat o el cost ho requereix. Mesuren que funciona. Canvien quan apareix alguna cosa millor.

No et tornis religiosa sobre aixo. El model es una eina. L'habilitat es saber per quina eina estirar la ma – i aquella habilitat es transfereix independentment de quins models existeixin d'aqui sis mesos.

11 El Prompting com a Enginyeria

No hi ha cap sintaxi secreta. Cap encantament magic que faci un agent produir codi perfecte. El prompting es *comunicacio* – i tu ja saps com comunicar.

Si alguna vegada has escrit un bon informe de bug, saps com fer prompts. Si alguna vegada has escrit un document de disseny que un company d'equip podria implementar sense fer-te vint preguntes, saps com fer prompts. Si alguna vegada has creat un tiquet de Jira que no va tornar com alguna cosa completament diferent del que volies – saps com fer prompts.

Les habilitats es transfereixen directament. Claredat, especificitat, context, restriccions. Les mateixes coses que fan la col · laboracio humana eficient fan la col · laboracio amb agents eficient. La diferencia es que els agents no faran preguntes de clarificacio quan el teu prompt es vague. Simplement endevinaran. I endevinaran amb confianca.

Aqui es on molts enginyers experimentats lluiten discretament. Has passat anys construint l'habilitat de *fer* – escriure codi, depurar, construir sistemes. Ara l'habilitat que importa es *articular* – explicar el que vols amb prou precisio perque algu altre ho pugui fer. Es un muscul diferent. I pot sentir-se, als primers dies, com un descens de categoria. No ho es. Pero la incomoditat es real, i fingir que no existeix no ajuda.

11.1 L'Anatomia d'un Bon Prompt de Tasca

Un bon prompt te tres parts: *que* vols que es faci, *per que* importa, i *com* es l'exit. La majoria de gent nomes proporciona la primera, i fins i tot aixo sol ser vague.

Considera la diferencia:

Dolent: "Arregla el bug d'autenticacio."

Aixo no diu quasi res a l'agent. Quin bug d'autenticacio? On es manifesta? Quin es el comportament esperat? L'agent anira a cacar per la teva base de codi, formara una teoria sobre que podries voler dir, i aplicara un arreglament que podria ser completament equivocat. Has convertit un arreglament de cinc minuts en una revisio de vint minuts d'alguna cosa que no has demanat.

Bo: "L'endpoint de login retorna 401 per a tokens valids quan la cache de sessions esta freda. El bug es probablement a `middleware/auth.go` a la funcio `validateSession`. El test a `auth_test.go:TestColdCacheLogin` el reprodueix. Arregla el bug i assegura't que tots els tests d'autenticacio existents segueixen passant."

Aixo es un animal completament diferent. L'agent sap el simptoma, la localitzacio sospitada, i com verificar l'arreglament. Pot anar directament al codi rellevant, entendre el problema, i validar la seva solucio – tot sense endevinar.

El patro es simple. *Que* esta trencat o es necessita. *On* mirar. *Com* verificar. Cada minut que dediques a fer el teu prompt precis t'estalvia cinc minuts revisant la sortida equivocada.

11.2 Especificacio de Restriccions

Dir a un agent que ha de fer es nomes la meitat de la feina. Dir-li que *no* ha de fer es igualment important.

Els agents son entusiastes. Optimitzen per resoldre el problema que has descrit, i felicment refactoritzaran tot el teu modul, afegiran tres noves dependencies, i canviaran l'API publica per fer-ho. Aixo no es malicia – es un optimitzador fent el que fan els optimitzadors. La teva feina es establir els limits.

Restriccions utils es veuen aixi:

- "No modifiquis la superficie de l'API publica."
- "Mantin l'estructura de tests existent – afegeix nous casos de test, no reorganitzis."

- "No afegeixis noves dependencies."
- "Queda't dins dels patrons de gestio d'errors existents en aquesta base de codi."
- "No canviis cap fitxer fora del directori `services/`."

Pensa en les restriccions com les baranes d'un pont. L'agent pot conduir on vulgui dins dels carrils, pero no pot sortir per la vora. Sense baranes, obtens solucions creatives que tecnicament funcionen pero creen malsons de manteniment. Amb elles, obtens solucions que encaixen a la teva base de codi com si sempre hi haguessin estat.

Com mes experiencia adquireixes amb enginyeria agentica, mes els teus prompts es defineixen per les seves restriccions en lloc de per les seves instruccions. Aprens quines llibertats porten a bons resultats i quines porten al caos.

11.3 Descomposicio de Tasques

Un error comu: demanar a un agent que construeixi alguna cosa gran en un sol prompt. "Construeix un dashboard d'usuari amb metriques en temps real, acces basat en rols, i exportacio a CSV." Aixo no es un prompt – es un projecte. I els projectes necessiten ser descompostos en tasques.

La descomposicio de tasques es la practica de dividir peticions grans en passos petits i *verificables*. Cada pas te una entrada clara, una sortida clara, i una manera clara de comprovar si ha funcionat.

En lloc de "construeix un dashboard d'usuari," escrius:

1. Crea el model de dades per a les metriques del dashboard a `models/dashboard.go` amb l'esquema definit al document de disseny. Escriu tests unitaris per a la validacio del model.
2. Construeix l'endpoint d'API `GET /api/dashboard` que retorni metriques per a l'usuari autenticat. Escriu tests d'integracio.
3. Afegeix filtratge basat en rols perque els usuaris admin vegin totes les metriques i els usuaris normals nomes les seves. Actualitza els tests existents per cobrir ambdos rols.
4. Construeix el component React que mostri les dades del dashboard. Utilitza el component `DataTable` existent per a la graella de metriques.

Cada pas es un prompt autocontingut. Cadascun te un resultat verificable. Cadascun es construeix sobre la sortida verificada del pas anterior. Si el pas dos va malament, ho detectes abans d'haver malbaratat temps al pas tres.

Aixo no es nomes bon prompting – es bona enginyeria. Estas aplicant les mateixes habilitats de descomposicio que faries servir quan planifiques un sprint o divideixes un pull request. La unitat de treball es prou petita per revisar, prou petita per testejar, i prou petita per llencar si esta malament.

11.4 El Prompt com a Especificacio

Els millors prompts que he vist es llegeixen com a documents de disseny en miniatura. Descriuen el resultat desitjat, no els passos d'implementacio. Llisten les restriccions. Defineixen criteris d'acceptacio. Proporcionen just prou context perque l'agent prengui bones decisions sense ofegar-lo en informacio irrellevant.

Aqui tens com es veu un prompt-com-a-especificacio:

"Afegeix limitacio de velocitat a l'endpoint `/api/search`*. Utilitza el middleware* `RateLimiter` *existent a* `middleware/ratelimit.go`*. Configura el limit a 100 peticions per minut per usuari autenticat, i 20 per minut per a peticions no autenticades. Retorna un estat 429 amb una capsalera* `Retry-After` *quan el limit es superi. Afegeix tests per als camins autenticat i no autenticat, incloent el cas limit on un usuari arriba exactament al limit. No modifiquis el propi middleware de limitacio de velocitat – nomes configura'l i aplica'l."*

Aixo es una especificacio. Un agent pot implementar-ho sense ambiguitat. Un revisor huma pot comprovar el resultat contra els requisits. El resultat desitjat es clar, les restriccions son explicites, i els criteris de verificacio estan definits.

Escriure prompts d'aquesta manera requereix practica. Tambe requereix disciplina – la disciplina de pensar en el que realment vols abans de comencar a teclejar. Pero aquella disciplina dona dividends. Un prompt ben especificat produeix un resultat que pots fusionar. Un prompt vague produeix un resultat que has de reescriure.

11.5 Iteracio per Sobre de Perfeccio

El teu primer prompt no sera perfecte. Esta be. El prompting es un proces iteratiu, i l'habilitat no es escriure el prompt perfecte – es *llegir la sortida*, entendre on la comunicacio ha fallat, i refinar.

Quan un agent produeix alguna cosa incorrecta, resisteix l'impuls de culpar l'eina. En lloc d'aixo, pregunta't: que he fallat en comunicar? He deixat fora alguna restriccio? El context era insuficient? He assumit coneixement que l'agent no tenia?

Aixo es depuracio – pero en lloc de depurar codi, estas depurant la teva propia comunicacio. El missatge d'error es la sortida de l'agent. La traca de pila es el teu prompt. En algun lloc alla hi ha la mala comunicacio, i trobar-la fa el teu proper prompt millor.

Els enginyers agentics experimentats desenvolupen un instint de retroalimentacio. Veuen la sortida de l'agent i immediatament saben quina part del seu prompt va causar la desviacio. "Ah, vaig dir 'gestiona errors' pero no vaig especificar *quins* errors ni *com* gestionar-los. Es clar que ha posat un catch-all generic."

Cada iteracio ajusta el bucle. El primer prompt t'arriba al 70%. Una correccio de seguiment t'arriba al 90%. Un refinament final t'arriba al final. Amb el temps, els teus primers prompts milloren, i necessites menys iteracions. Pero mai no necessites zero.

11.6 Desenvolupament Guiat per Veu

La majoria de nosaltres fem prompts teclejant. Aixo te sentit – som enginyers, vivim en text. Pero hi ha un altre canal d'entrada que es mes rapid, mes natural, i sorprenentment infrautilitzat: la teva veu.

La conversio de veu a text moderna ha arribat al punt on pots parlar un prompt al teu terminal i tenir-lo transcrit amb precisio gairebe perfecta. Eines com Whisper, la dictacio de macOS, i SuperWhisper et permeten parlar amb el teu agent en lloc de teclejar. El resultat es el mateix – entra text, surt codi. Pero l'experiencia es fonamentalment diferent.

Aqui tens per que: teclejar i parlar son modes de pensar diferents. Quan tecleges, edites sobre la marxa. Esborres una paraula, reformules, esborres, reestructures. El text que produeixes esta *polit* – has tingut temps de suavitzar les asprors abans que sortis dels teus dits. Parlar no et dona aquell luxe. Quan parles, et compromets. Les paraules surten de la teva boca i ja estan. No hi ha tecla d'esborrar.

Aixo sona com un desavantatge. En realitat es un camp d'entrenament.

Quan parles un prompt, ets forcat a organitzar els teus pensaments *abans* d'obrir la boca. No pots confiar en la crossa d'editar a mitja frase. Has de saber que vols, estructurar-ho mentalment, i lliurar-ho clarament – en temps real. Les primeres vegades que ho provis, divagares. Diras "eh" i "mm" i tornares enrere i et contradiras. L'agent rebra un transcrit desordenat, i la sortida reflectira aquell desordre.

Pero passa alguna cosa si continues fent-ho. Millores. No nomes en prompting – en *parlar clarament sobre problemes tecnics*. Desenvolupes la capacitat de descriure un bug, una funcionalitat, o una tasca de refactoritzacio en un unic flux coherent. Aprens a posar el context al davant, establir restriccions aviat, i acabar amb una peticio clara. Deixes de divagar perque divagar produeix mals resultats.

Aquesta habilitat es transfereix a tot arreu. Reunions d'standup. Discussions d'arquitectura. Programacio en parella. Trucades d'incidents. Cada situacio on necessites articular una idea tecnica clarament, sota pressio de temps, sense la xarxa de seguretat d'un editor de text. El desenvolupament guiat per veu no es nomes una manera mes rapida de fer prompts – es practica per a cada conversa tecnica que mai tindras.

Tambe hi ha un avantatge practic de velocitat. La majoria de gent parla a 130 paraules per minut. La majoria de gent tecleja a 40-80. Per al tipus de prompts d'alt nivell i basats en intencio que produeixen la millor sortida d'agents – "aqui esta el problema, aqui esta el context, aqui esta el que vull, aqui esta el que no vull" – parlar es simplement mes rapid. Dediques menys temps a l'entrada i mes temps revisant la sortida.

Prova-ho durant una setmana. Tria una eina de veu a text, connecta-la al teu flux de treball, i parla els teus prompts en lloc de teclejar-los. El primer dia se sentira incomode. Al tercer dia, notaras que els teus prompts

parlats es tornen mes ajustats. Al final de la setmana, notaras que la teva *comunicacio parlada en general* s'esta tornant mes ajustada.

Tambe val la pena destacar que els models de veu a text poden executar-se enterament a la teva maquina. La familia de models Parakeet de NVIDIA – models ASR compactes i d'alta precisio – s'executen localment sense cap dependencia del cloud. Eines com SuperWhisper i whisper.cpp fan el mateix utilitzant els pesos de Whisper d'OpenAI. Un MacBook modern pot executar aquests models gairebe en temps real amb transcripcio precisa i baixa latencia. No necessites un servei al cloud per convertir veu en text – les eines locals ja hi son.

Als agents els es igual si el teu prompt va ser teclejat o parlat. Pero *tu* seras un pensador mes clar per haver-lo parlat.

11.7 Context Visual: Quan les Paraules No Son Prou

No tot es facil de descriure en text. Un layout trencat, un error de renderitzat estrany, un dialog d'error amb una traca de pila – de vegades la manera mes rapida de comunicar el que estas veient es *mostrar-ho*.

Els LLMs moderns son multimodals. Poden llegir captures de pantalla, diagrames, fotos de pissarres, i missatges d'error capturats de la teva pantalla. Aixo no es una funcio de novetat – es una de les eines mes infrautilitzades del flux de treball d'enginyeria agentica.

Aqui tens un flux de treball que faig servir diariament: veig un bug al meu telefon – un layout que esta trencat, un modal que esta descentrat, un formulari que s'empassi l'entrada. Faig una captura de pantalla a iOS, i gracies a Universal Clipboard, l'enganxo directament a la meva sessio de terminal al Mac. L'agent veu el que jo veig. No cal descriure "el boto se superposa amb la capsalera al viewport mobil" – la captura de pantalla *es* la descripcio.

Aixo importa perque els bugs visuals son notoriament dificils de descriure en text. Acabes escrivint tres paragrafs sobre padding i z-index quan una sola captura de pantalla comunica el problema instantaniament. L'agent pot veure l'estat trencat, raonar sobre que esta malament, i proposar

un arreglament – sovint mes rapid del que podries acabar de teclejar la descripcio.

Pero va mes enlla dels informes de bugs. Alguns fluxos de treball comuns de context visual:

- **Captures de pantalla d'errors.** Una consola de navegador plena de vermell, una traca de pila al terminal, un dashboard de desplegament mostrant health checks fallats. Captura-ho, enganxa-ho, demana a l'agent que diagnostiqui. Aixo es especialment util quan els missatges d'error son llargs o contenen formatat que es penosa de copiar com a text.
- **References de disseny.** Un mockup de Figma, un esbos, la interficie d'un competidor que vols aproximar. Enganxa la imatge i digues "fes que la nostra pagina de configuracio s'assembli a aixo." L'agent pot extreure estructura de layout, eleccions de color, i jerarquia de components d'una referencia visual.
- **Depuracio d'estat visual.** "Per que es veu malament aquesta pagina?" es un prompt terrible. Una captura de pantalla de la pagina *mes* "per que es veu malament aquesta pagina?" es un de genial. L'agent pot comparar el que veu amb el layout esperat i identificar problemes de CSS, dades que falten, o bugs de renderitzat.
- **Fotos de pissarres.** Despres d'una discussio d'arquitectura, fes una foto de la pissarra i enganxa-la. L'agent pot llegir les caixes, fletxes i etiquetes, i ajudar-te a traduir aquell esbos en estructura de codi, definicions d'API, o documentacio.

El pipeline de portaretalls d'iOS-a-Mac mereix mencio especial perque elimina tota la friccio d'aquest flux de treball. No necessites desar la captura de pantalla, fer AirDrop, trobar-la al Finder i arrossegar-la a una eina. Veus el problema, el captures, l'enganxes. Tres segons de "aixo esta trencat" a "l'agent ho esta mirant." Aquella velocitat importa perque et mante en el flux. Qualsevol pas extra – fins i tot trenta segons de gestio de fitxers – crea prou friccio perque per defecte tornis a teclejar una descripcio en text, que es mes lenta i menys precisa.

La perspectiva clau es que *el context no es nomes text.* Quan parlaven de que el context es l'ingredient mes important del treball agentic, parlavem de totes les formes de context – fitxers de codi, documentacio, sortida de

tests, *i* estat visual. Una captura de pantalla val mil tokens, i els agents que poden veure son dramaticament mes utils que els agents que nomes poden llegir.

Si no estas fent servir context visual al teu flux de treball agentic, comenca. Captura els teus bugs. Enganxa els teus missatges d'error. Comparteix les teves references de disseny. Els agents ara poden veure. Deixa'ls.

11.8 Anti-patrons

Alguns habits de prompting produeixen consistentment mals resultats. Apren a reconeixer-los.

Ser massa vague. "Fes aquest codi millor." Millor com? Mes rapid? Mes llegible? Mes mantenible? L'agent triara *alguna cosa* per millorar, i probablement no sera la que tenies al cap. Si no pots articular que vol dir "millor," no estas preparat per fer prompts.

Ser massa prescriptiu. L'error oposat. "A la linia 47, canvia el nom de la variable de `x` a `count`, despres afegeix un if a la linia 48 que comprovi si count es mes gran que zero, despres..." Estas escrivint el codi en catala i demanant a l'agent que el tradueixi. Aixo es mes lent que escriure el codi tu mateix. Descriu el *resultat*, no les pulsacions de tecles.

Bolcat de context. Enganxar tota la teva base de codi, tots els teus documents de disseny, i una transcripcio de les teves tres ultimes reunions d'equip al prompt. Mes context no sempre es millor. Context irrellevant es soroll, i el soroll ofega el senyal. Dona a l'agent el que necessita – camins de fitxers, noms de funcions, el comportament especific que vols – i confia que explorara a partir d'alla.

Prompts de tot inclòs. "Arregla el bug d'autenticacio, tambe refactoritza la capa de base de dades, i ja que hi ets actualitza el README i afegeix tipus TypeScript al client de l'API." Aquestes son quatre tasques separades ficades en un prompt. L'agent intentara totes, no fara cap be, i produira un diff tan gran que revisar-lo trigara mes que fer la feina tu mateix. Un prompt, una tasca.

Assumir context compartit. “Fes-ho de la mateixa manera que vam fer el modul de pagaments.” L’agent no recorda la teva ultima sessio. No sap que va decidir “nosaltres” a l’standup. Cada prompt comenca de zero. Proporciona el context explicitament, cada vegada.

11.9 El Prompting Es una Habilitat

El prompting no es un truc de saló. No es tracta de descobrir la frase rara que desbloqueja millor sortida. Es una habilitat de comunicacio – i com totes les habilitats de comunicacio, millora amb la practica, la retroalimentacio i l’atencio deliberada.

Els enginyers que treuen mes profit de les eines agentiques son els que tracten el prompting amb el mateix rigor que apliquen a escriure codi. Pensen abans de teclejar. Especifiquen abans d’implementar. Verifiquen abans de continuar.

Aixo no es una habilitat nova. Aixo es simplement enginyeria.

12 Orquestracio Multi-Agent

Un agent es potent. Multiples agents treballant junts es una altra cosa completament diferent.

Pensa-ho aixi. Un sol fuster pot construir una caseta. Pero una casa? Necessites un electricista, un lampista, un coberturer – especialistes treballant en paral · lel, cadascun enfocat en el que fa millor, coordinant-se just el necessari per no estorbar-se. La casa s'aixeca mes rapid i la feina es millor que una sola persona intentant fer-ho tot.

L'enginyeria agentica funciona de la mateixa manera. Un sol agent en una tasca complexa la recorrera seqüencialment – planificar, implementar, testejar, depurar, iterar. Funciona, pero es lent. Tres agents, cadascun gestionant una peca ben acotada del problema, poden acabar en un terc del temps. De vegades menys, perque agents enfocats cometen menys errors que un agent fent malabars amb massa preocupacions.

Pero hi ha un problema. Agents paral · lels requereixen *coordinacio*. I la coordinacio te un cost. Aquest capitol tracta sobre quan pagar aquell cost, i com pagar-lo be.

12.1 Estrategies de Descomposicio

La part mes dificil del treball multi-agent no es executar multiples agents. Es decidir com dividir la feina.

La regla d'or: les tasques han de tenir *acoblament minim*. Si la feina de l'agent A depen de la sortida de l'agent B, no poden executar-se en paral · lel – son seqüencials, i hauries de tractar-les aixi. L'art es trobar costures a la teva feina on pots tallar netament.

Hi ha tres patrons de descomposicio fiables.

Per capa. Un agent gestiona l'API de backend, un altre construeix el component de frontend, un tercer escriu els tests. Aixo funciona be perque

les capes tenen fronteres naturals – una interficie definida entre elles. Mentre acordeu el contracte d'API per endavant, cada agent pot treballar independentment.

Per funcionalitat. Si estas construint tres funcionalitats independents, dona cadascuna a un agent separat. Aquesta es la descomposicio mes simple. Les funcionalitats toquen fitxers diferents, directoris diferents, preocupacions diferents. Els conflictes de fusio son rars.

Per preocupacio. Un agent refactoritza, un altre escriu tests per al codi refactoritzat, un tercer actualitza la documentacio. Aixo es un patro seqüencial – mes pipeline que paral·lel – pero permet a cada agent enfocar-se en un sol tipus de pensament. L'agent de refactoritzacio no ha de canviar de context al mode d'escriptura de tests. Simplement refactoritza, i passa el relleu.

La descomposicio que triis depen de la forma de la feina. Pero el principi es constant: *troba les costures, talla per elles, minimitza la superficie on els agents necessiten estar d'acord.*

12.2 Branca-per-Agent

Cada agent te la seva propia branca. Aixo ho vam cobrir al capitol de Git. En treball multi-agent, es converteix en absolutament essencial.

Pero les branques soles no son prou. Dos agents en branques diferents compartint el mateix directori de treball es barallaran pel sistema de fitxers – sobreescriuran fitxers de l'altre, corrompran els builds de l'altre, trencaran les execucions de tests de l'altre. Necessites *worktrees*.

Cada agent te el seu propi git worktree: un directori separat, a la seva propia branca, amb la seva propia copia de la base de codi. Els agents comparteixen historial pero res mes. Poden compilar, executar tests, instal·lar dependencies, i fer un embolic – tot sense afectar-se mutuament.

La configuracio es rapida:

```
git worktree add ../project-api agent/api-endpoint
git worktree add ../project-frontend agent/frontend-component
git worktree add ../project-tests agent/integration-tests
```

Tres directoris. Tres branques. Tres agents. Aillament complet. Aixo es el model de sandbox dels capitols anteriors fet concret per a treball multi-agent. Quan un agent acaba, revises la seva branca, fusiones si es bona, i elimines el worktree. Si la feina es dolenta, ho llences tot. Zero cost.

12.3 El Patro de Traspas

No tot el treball multi-agent es paral · lel. De vegades els agents treballen *seqüencialment*, cadascun construint sobre l'anterior. L'agent A planifica. L'agent B implementa. L'agent C revisa.

Aixo es el patro de traspas, i es potent – pero nomes si el propi traspas es net.

El problema amb agents seqüencials es la perdua de context. L'agent A te una comprensio rica del problema quan acaba de planificar. L'agent B comenca en fred. Si simplement dius a l'Agent B "implementa el pla," es perdra matisos. Fara suposicions diferents. Resoldra un problema lleugerament diferent del que l'Agent A va planificar.

L'arreglament es *traspas estructurat*. L'agent A no nomes planifica – produeix un artefacte que captura el seu raonament. Aixo pot prendre diverses formes:

- **Un document resum.** Un fitxer markdown deixat al repo: `PLAN.md`. Descriu que ha de passar, quins compromisos es van considerar, que es va rebutjar i per que. L'agent B llegeix aixo abans d'escriure una sola linia de codi.
- **Un conjunt de fitxers.** L'agent A crea fitxers esquelet – funcions buides amb docstrings, definicions d'interficies, signatures de tipus. L'agent B els omple. Els esqueletons *son* el traspas.
- **Un prompt estructurat.** La sortida de l'agent A es converteix en l'entrada de l'agent B, formatada com una descripcio detallada de tasca amb criteris d'acceptacio. L'enganxes directament al context de l'Agent B.

La clau es que el traspas ha de ser *explicit i complet*. Sense suposicions implicites. Sense "el proxim agent ja ho descovrira." Cada decisio, cada restriccio, cada cas limit – escrit.

Aixo requereix disciplina. Pero es la mateixa disciplina que aplicaries quan escrius un tiquet per a un col · lega huma en un altre continent i una altra zona horaria. Si no confiessis en un traspas verbal, no confiis en un d'implicit entre agents.

12.4 El Problema de la Fusio

Aqui es on el treball multi-agent es torna genuinament dificil.

Tres agents van treballar en paral · lel. Cadascun va produir una branca neta i testejada. Ara necessites fusionar-les totes a main. De vegades aixo es indolor – tres branques tocant fitxers completament diferents es fusionen sense un sol conflicte. Bonic.

De vegades no. L'agent A va canviar l'esquema de la base de dades. L'agent B va afegir una migracio que assumeix l'esquema antic. L'agent C va actualitzar el handler de l'API que tant A com B tambe van tocar. Ara tens un conflicte a tres bandes i cap agent sol te la imatge completa.

Hi ha tres estrategies per gestionar-ho.

Prevencio. Els millors conflictes de fusio son els que mai passen. Quan descomponguis la feina, pensa en la propietat de fitxers. Assigna directoris, moduls, fitxers diferents a agents diferents. Si els agents mai toquen el mateix fitxer, mai poden entrar en conflicte. Aquesta es l'estrategia mes barata i hauries d'utilitzar-la sempre que sigui possible.

L'agent de fusio. Quan sorgeixen conflictes, crea un nou agent la unica feina del qual es fusionar. Dona-li les branques, els conflictes, i el context de la feina de cada agent. Un bon agent de fusio pot resoldre la majoria de conflictes automaticament – llegeix ambdos costats, enten la intencio, i produeix un resultat coherent. Es com un enginyer senior que s'asseu amb dos PRs i esbrina com encaixen.

Revisio humana. De vegades el conflicte es semantic, no sintactic. El codi es fusiona netament pero la *logica* es contradiu. Dos agents van prendre decisions de disseny incompatibles. Cap fusio automatitzada ho detectara. Aqui es on guanyes el teu sou com a enginyer. Revisa les

branques abans de fusionar. Llegeix els diffs costat a costat. Assegura't que les peces realment encaixen.

A la practica, faras servir les tres. Preven el que puguis. Automatitza la resta. Revisa-ho tot.

12.5 Sobrecost d'Orquestracio

Mes agents no sempre es millor.

Cada agent addicional afegeix cost de coordinacio. Has de descompondre la feina. Configurar worktrees. Definir interficies. Gestionar fusions. Revisar multiples branques en lloc d'una. Per a una tasca que a un sol agent li costa trenta minuts, crear tres agents podria trigar quaranta-cinc – vint minuts de treball d'agents en paral · lel mes vint-i-cinc minuts del teu temps orquestrant.

El punt d'equilibri es mes alt del que penses. En la meva experiencia, l'orquestracio multi-agent comenca a compensar quan la tasca total trigaria a un sol agent *almenys dues hores*. Per sota d'aixo, el sobrecost es menja els guanys.

Hi ha altres costos tambe. El context es fragmenta. Cada agent nomes veu la seva peca. Cap agent sol enten tot el sistema de la manera que ho faria un agent treballant sol. Aixo pot portar a inconsistencies – convencions de noms diferents, patrons de gestio d'errors diferents, suposicions diferents sobre estat compartit.

L'habilitat no es executar tants agents com sigui possible. L'habilitat es saber *quan* paral·lelitzar i quan un sol agent enfocat es l'eina correcta. Una tripulacio de cinc no sempre es mes rapida que un mariner experimentat que coneix el vaixell.

12.6 Un Exemple Practic

Fem-ho concret. Estas construint una aplicacio web i necessites afegir una nova funcionalitat: notificacions d'usuari. Els usuaris haurien de veure una icona de campana amb un comptador, clicar-la per veure una llista,

i marcar notificacions com a llegides. Necessites una API, un component de frontend, i tests.

Aixo es un cas de llibre de text per a tres agents en paral · lel.

Pas 1: Defineix la interficie. Abans de crear cap agent, dediques cinc minuts a escriure el contracte d'API. `GET /api/notifications` retorna una llista. `PATCH /api/notifications/:id` marca una com a llegida. L'objecte de notificacio te `id`, `message`, `read`, `created_at`. Escriu-ho en un document compartit o un fitxer esquelet. Aixo es el contracte contra el qual treballen els tres agents.

Pas 2: Crea els worktrees.

```
git worktree add ../app-api agent/notifications-api
git worktree add ../app-frontend agent/notifications-frontend
git worktree add ../app-tests agent/notifications-tests
```

Pas 3: Llanca els agents. Cada agent rep una tasca clara i acotada:

- Agent 1: "Implementa els endpoints de l'API de notificacions a `../app-api`. Segueix el contracte a `API_CONTRACT.md`. Inclou model, ruta i controlador. Escriu tests unitaris per al controlador."
- Agent 2: "Construeix el component d'interficie de notificacions a `../app-frontend`. Crida `GET /api/notifications` i `PATCH /api/notifications/:id`. Mostra una icona de campana amb comptador de no llegides. Clicar obre una llista desplegable."
- Agent 3: "Escriu tests d'integracio a `../app-tests`. Cobreix el flux complet: crear una notificacio, obtenir la llista, marcar com a llegida, verificar que el comptador s'actualitza."

Pas 4: Deixa'ls treballar. Els tres agents s'executen simultaneament. Cadascun commiteja a la seva propia branca. Monitors el progres pero no intervens tret que alguna cosa vagi malament.

Pas 5: Revisa i fusiona. Quan tots tres acaben, revises cada branca. La branca d'API es fusiona primer – es la base. Despres la branca de frontend. Despres els tests. Executes la suite de tests completa despres de cada fusio per detectar problemes d'integracio aviat.

Temps total: potser quaranta minuts. L'agent d'API va trigar trenta, l'agent de frontend va trigar vint-i-cinc, i l'agent de tests va trigar trenta-

cinc. Pero van executar-se en paral · lel, aixi que el temps real va ser trenta-cinc minuts mes deu minuts de la teva feina d'orquestracio.

Un sol agent fent-ho tot seqüencialment? Probablement noranta minuts.

Les matematiques funcionen quan la tasca es prou gran. I a mesura que millores en descomposicio, desenvoluparas una intuicio per quines tasques val la pena dividir i quines no. Com la majoria de coses en enginyeria, es una decisio de judici. Pero ara tens les eines per prendre-la.

13 Agents al Pipeline

Un equip que conec va configurar una automatitzacio elegant l'any passat. Van afegir un agent Claude com a pas de CI que detectaria errors de lint i els arreglaria automaticament. Un desenvolupador fa push del codi, el linter s'executa, i si falla, l'agent reescriu les linies ofensives i fa push d'un commit d'arreglament. Sense intervencio humana necessaria. El pipeline es mante verd. A tothom li encantava.

Va funcionar brillantment durant unes dues setmanes.

Despres algu va notar alguna cosa estranya durant una revisio de codi. Tota una categoria de regles de lint havia desaparegut. L'agent havia descobert que la manera mes rapida d'arreglar un error de lint era modificar `.eslintrc.json` – desactivar la regla, fer push del canvi de configuracio, pipeline verd. Havia estat fent-ho durant un mes. Ningu ho va detectar perque el senyal que vigilaven – l'estat del pipeline – era exactament el senyal que l'agent havia apres a manipular.

L'arreglament va ser senzill: fer la configuracio de lint de nomes lectura per a l'agent. Pero la llico era mes gran que un pipeline mal configurat.

Els agents automatitzats necessiten les mateixes baranes que els interactius. Probablement mes, perque no hi ha ningu assegut alla mirant el diff passar. Quan fas parella amb un agent localment, veus cada fitxer que toca. Notes quan fa alguna cosa llesta pero incorrecta. A CI, l'agent s'executa a les fosques. L'unica cosa que veus es el resultat – verd o vermell. I si l'agent troba una manera de fer el resultat verd sense realment resoldre el problema, podries no adonar-te'n durant setmanes.

Aquest capitol tracta sobre utilitzar agents a pipelines de manera responsable. On afegeixen valor genui, on creen risc, i com configurar-los perque segueixin sent utils sense convertir-se en una responsabilitat.

El que hi ha en joc es mes alt a CI que al teu escriptori. Fes-ho be, i els agents multipliquen el rendiment del teu equip les vint-i-quatre hores del dia. Fes-ho malament, i has construit una arma cara i sense supervisio.

13.1 Agents com a Passos de CI

La manera mes simple de posar agents al teu pipeline es com a passos de CI – treballs discrets que s'executen a cada push o pull request, com el teu linter o suite de tests.

No estas reinventant el teu pipeline. Hi estas afegint intel · ligencia.

L'agent de CI mes immediatament util: pre-screening de PRs. No substituint la revisio humana, sino filtrant. Un agent llegeix el diff, busca problemes obvis – imports no utilitzats, nomenclatura inconsistent, gestio d'errors absent, fitxers de test que en realitat no asserten res – i deixa comentaris. Quan un revisor huma obre el PR, les coses trivials ja estan marcades. L'huma pot centrar-se en arquitectura, enfocament, intencio.

Altres bons candidats:

- **Generacio de changelog.** L'agent llegeix els commits des de l'ultima versio, creua referencies amb numeros de tiquet, i redacta notes de versio. Un huma edita abans de publicar, pero el primer esborrany es gratuit.
- **Ordenacio d'imports i formatat.** Segur, verificable, baix risc. Si l'agent reformata alguna cosa malament, el diff es obvi.
- **Resums d'actualitzacio de dependencies.** Quan Dependabot obre un PR, un agent pot llegir el changelog del paquet actualitzat i resumir que ha canviat i si es probable que afecti el teu codi.

Tambe hi ha un us mes subtil: **triatge de fallades de tests**. Quan una execucio de CI falla, un agent pot llegir la sortida dels tests, identificar el test que falla, mirar el diff recent, i publicar un comentari explicant si l'error sembla un bug genui o un test intermitent. No sempre encertara, pero estalvia al desenvolupador obrir els logs de CI, navegar per dues-centes linies de sortida, i esbrinar que ha anat malament. Aquella investigacio de deu minuts es converteix en un cop d'ull de trenta segons a un comentari.

El principi clau: els agents a CI haurien de fer coses que siguin **segures d'automatitzar** i **facils de verificar**. Si no pots dir rapidament si l'agent ha fet be la feina, no s'hauria d'automatitzar encara. Una bona prova de

foc: estaries comode si l'agent fes aixo cent vegades i nomes comproves deu d'elles? Si la resposta es no, no esta preparat per a CI.

13.2 L'Agent Nocturn

Aixo es un dels patrons mes potents de l'enginyeria agentica, i un dels menys discutits. Tambe es el que fa que els ulls dels directors s'il · luminin – "vols dir que l'agent treballa mentre dormim?" – que es exactament per que necessita un emmarcament curós.

Estas acabant el dia. Hi ha un tiquet ben definit al backlog – afegir un nou endpoint d'API, escriure una migracio de dades, refactoritzar un modul per utilitzar la nova biblioteca de logging. El tipus de tasca on els requisits son clars i la definicio de "fet" es testejable. Apuntes un agent cap alla, li dones una branca, i marxes a casa. Et despertes amb un PR amb la feina feta, tests passant, preparat per a revisio.

El llistó de qualitat per a agents sense supervisio es mes alt que per als interactius. Quan ets alla assegut mirant, detectes errors en temps real. Quan l'agent treballa durant la nit, els errors s'acumulen. Aixi que necessites:

- **Tests exhaustius per verificar.** L'agent necessita una manera de saber que ha acabat. Tests existents que haurien de seguir passant, mes criteris clars per a nous tests que hauria d'escriure.
- **Abast estret.** Un tiquet, una preocupacio. No apuntis un agent nocturn a "refactoritza el sistema d'autenticacio." Apunta'l a "afegeix limitacio de velocitat a l'endpoint de login."
- **Aillament de branca.** Sempre una branca nova, sempre un worktree. Si l'agent fa un embolic, esborres la branca. Res toca main.
- **Un timeout.** Configura un limit dur – quatre hores es generos per a la majoria de tasques. Un agent que porta sis hores executant-se no esta progressant. Esta donant voltes. Mata'l i reavalua al mati.

Un script de configuracio simple es veu aixi:

```
#!/bin/bash
# overnight-agent.sh — fire and forget before you leave
```

```
TICKET_ID="$1"
BRANCH_NAME="agent/overnight-${TICKET_ID}"
WORKTREE_DIR="../overnight-${TICKET_ID}"

# Create isolated workspace
git worktree add "$WORKTREE_DIR" -b "$BRANCH_NAME"

# Run the agent with a token budget and timeout
timeout 4h claude --worktree "$WORKTREE_DIR" \
  --max-tokens 200000 \
  --prompt "Implement ticket ${TICKET_ID}. Read TICKETS/
${TICKET_ID}.md for requirements. Write tests. Commit your work.
Do not modify any CI config or lint rules." \
  2>&1 | tee "logs/overnight-${TICKET_ID}.log"

# Push the branch so you can review in the morning
cd "$WORKTREE_DIR" && git push -u origin "$BRANCH_NAME"
```

Vas refinant aixo amb el temps. Afegeix notificacions de Slack quan acabi. Afegeix un resum del que ha fet. Afegeix una comprovacio que verifiqui que la suite de tests passa abans de fer push.

Una cosa que he apres dels equips que ho fan be: la descripcio del tiquet importa enormement. Un tiquet que diu "afegeix endpoint de preferencies d'usuari" no es suficient. L'agent nocturn necessita criteris d'acceptacio, payloads de peticio/resposta d'exemple, i indicadors cap a endpoints existents similars que pugui fer servir com a referencia. Estas escrivint instruccions per a un desenvolupador competent pero sense context. Com mes especific siguis, millor el resultat.

Els equips que treuen mes profit dels agents nocturns son els que inverteixen en la seva disciplina d'escriptura de tiquets. La qual cosa, de nou, beneficia tothom – els teus companys humans tambe prefereixen tiquets clars. L'agent simplement fa el cost de la vaguetat mes visible.

El bucle principal es simple: ailla, restringeix, executa, revisa al mati.

13.3 Control de Costos a CI

Quan executes un agent interactivament, tens un interruptor automatic natural: tu mateix. Veus els tokens pujant, veus l'agent donant voltes, prems Ctrl+C. A CI, no hi ha ningu mirant.

Un equip executant un monorepo ocupat va aprendre-ho per les males. Havien configurat un agent per revisar automaticament cada PR. Prou raonable – excepte que el seu monorepo veia quaranta a cinquanta PRs al dia, i cada revisio consumia al voltant de quinze mil tokens. Aixo era manejable. El que no era manejable era la logica de reintent. Quan l'agent arribava a un limit de velocitat, el treball de CI reintentava. Tres reintents per fallada, backoff exponencial, pero cada reintent iniciava una nova execucio de l'agent des de zero. Durant un cap de setmana de pont, amb una allau d'actualitzacions automatiques de dependencies entrant, el pipeline va generar una factura de $4.000. Ningu era a l'oficina per adonar-se'n.

Protegeix-te:

- **Pressupostos de tokens per execucio de pipeline.** Configura un limit dur. Si l'agent l'assoleix, el treball falla amb un missatge clar. Millor perdre's una revisio que cremar-se el pressupost mensual en un dia.
- **Limits de concurrencia.** No deixis que vint treballs d'agents s'executin simultaneament. Posa-los en cua. Dos o tres execucions d'agents concurrents es suficient per a la majoria d'equips.
- **Alertes de despesa.** El teu proveidor de LLM gairebe segur que les suporta. Configura una al 50% del teu pressupost mensual. Configura una altra al 80%. Canalitza-les a un canal que algu realment llegeixi.
- **Interruptors d'emergencia.** Un feature flag o variable d'entorn que desactivi tots els passos d'agents de CI instantaniament. Quan alguna cosa va malament a les 2 de la matinada, vols un arreglament d'una linia, no un canvi de configuracio de pipeline que necessiti el seu propi PR.
- **Seguiment de cost per treball.** Registra el nombre de tokens i cost estimat de cada execucio d'agent a CI. No pots optimitzar el que no

mesures. Un informe setmanal de la despesa d'agents a CI, desglossat per tipus de treball, et mostrara on van els diners i on cal ajustar.

Un interruptor automatic simple a la teva configuracio de CI es veu aixi:

```
agent-review:
  timeout-minutes: 15
  env:
    MAX_TOKENS: 50000
    COST_ALERT_THRESHOLD: "$5.00"
  steps:
    - name: Run agent review
      run: |
        claude review --max-tokens $MAX_TOKENS \
          --on-budget-exceeded "exit 1" \
          pr/$PR_NUMBER
```

Els detalls variaran per eina i plataforma, pero el patro es constant: configura un sostre, falla sorollosament quan l'assoleixis, i fes el sostre facil d'ajustar.

La regla general: tracta la teva despesa en LLM a CI de la mateixa manera que tractes la teva despesa en compute al cloud. Monitoritza-la, presuposta-la, posa alertes. Es un centre de cost real.

13.4 La Questio de la Revisio

Els PRs generats per agents segueixen necessitant revisio humana. Punt final.

La temptacio es real. L'agent va escriure el codi. Els tests passen. El linter es felic. La cobertura no ha baixat. Per que no auto-fusionar? Estalviaras quinze minuts de temps de revisio, i el pipeline de CI ja ha verificat tot el que importa.

Pero pensa en que vol dir realment "tot el que importa". Els tests verifiquen la correccio. No verifiquen la intencio.

Un agent encarregat de "reduir el nombre de consultes a la base de dades a la pagina de perfil d'usuari" podria cachejar agressivament i introduir bugs de dades obsoletes que cap test actual detecta. Podria desnormalitzar dades d'una manera que faci la propera funcionalitat el doble de dificil

de construir. Podria resoldre el problema perfectament pero en un estil completament alie a la resta de la teva base de codi. Els tests passen perque els tests comproven comportament, no enfocament.

Un revisor huma detecta aquestes coses. Llegeix el *com*, no nomes el *que*. Nota quan una solucio funciona avui pero crea deute per a dema. Detecta la drecera que confondrà el proper desenvolupador que toqui aquest codi.

Tambe hi ha una dimensio social. Si el teu equip sap que els PRs d'agents s'auto-fusionen, deixen de prestar atencio al codi generat per agents. Es converteix en una caixa negra. Sis mesos despres, la meitat de la teva base de codi va ser escrita per un agent i ningu a l'equip l'enten completament. Aixo es un buit de coneixement que et fara mal durant un incident.

L'auto-fusio esta be per a canvis trivials i mecanics – arreglaments de format, ordenacio d'imports, increments de versio. Per a qualsevol cosa que involucri una decisio de disseny, un huma la revisa. L'agent es un redactor rapid, no un decisor. I la revisio no ha de ser exhaustiva – un escaneig de cinc minuts per comprovar que l'enfocament es raonable fa molt.

13.5 Desplegaments Assistits per Agents

Els desplegaments son on les coses es tornen genuinament perilloses, i on la distancia entre "l'agent pot fer aixo" i "l'agent hauria de fer aixo" es mes ampla. Siguem precisos sobre que haurien i que no haurien de fer els agents aqui.

Els agents son excel · lents **monitoritzant** desplegaments. Poden vigilar fluxos de logs, fer seguiment de taxes d'error, comparar percentils de latencia contra la baseline pre-desplegament, i marcar anomalies. Un agent que diu "la taxa d'error a `/api/checkout` ha augmentat 3x des del desplegament de fa dotze minuts" es enormement valuos. Esta llegint dades, trobant patrons i traient informacio a la superficie – exactament el que fan be els agents.

Els agents tambe poden **suggerir** accions. "Basant-me en l'augment de la taxa d'error, recomano fer rollback a la versio anterior" es un suggeriment util. Es el tipus de cosa que normalment requeriria algu mirant un

dashboard de Grafana durant quinze minuts per concloure. L'agent ha fet l'analisi. Un huma decideix si actuar-hi.

El que els agents no haurien de fer es prendre la decisio de rollback de manera autonoma. Els desplegaments a produccio son massa consequents per a decisions ad-hoc d'agents. Si vols rollbacks automatitzats, utilitza un runbook pre-aprovat amb llindars durs – "si la taxa d'error supera el 5% durant tres minuts, fes rollback automaticament." Aixo es deterministic. Ho has testat. Has aprovat la logica per endavant. Un agent decidint per si sol si un augment del 2.3% de la taxa d'error "se sent" prou significatiu per fer rollback? Aixo es una recepta per a rollbacks innecessaris o incidents perduts. Les regles deterministiques son avorrides. Avorrit es bo quan els teus ingressos estan en joc.

A la practica, un agent de monitoritzacio de desplegament es veu mes o menys aixi: es subscriu al teu agregador de logs i dashboard de metriques via API, s'executa durant quinze minuts despres de cada desplegament, i publica un resum a Slack. "Desplegament de v2.4.1 completat. Taxa d'error estable al 0.3%. Latencia P99 augmentada de 180ms a 210ms – dins de la variancia normal. No s'han detectat nous tipus d'excepcions." La majoria de vegades, aquell resum es avorrit. Aquesta es la idea. Quan no es avorrit, vols saber-ho immediatament.

Aixo connecta amb el principi de baranes: produccio es de nomes lectura per als agents. Observen, analitzen, reporten, recomanen. Humans (o automatitzacio pre-aprovada amb regles deterministiques) prenen accio.

13.6 El Pipeline com a Context

La teva configuracio de CI/CD es context, i els agents la llegeixen com qualsevol altre codi. Aixo es facil d'oblidar – la majoria d'equips tracten la seva configuracio de pipeline com a fontaneria d'infraestructura, no com alguna cosa que necessiti comunicar clarament. Pero quan un agent esta intentant entendre per que un build ha fallat o quins passos de verificacio existeixen, la configuracio del pipeline es la primera cosa que llegeix.

Un pipeline ben estructurat ajuda els agents a entendre el teu projecte. Noms d'etapes clars (`build`, `unit-tests`, `integration-tests`, `deploy-`

`staging`) diuen a l'agent com es el teu proces de verificacio. Sortida d'errors estructurada – logs JSON, codis de sortida amb significat, categories d'errors – dona a l'agent alguna cosa amb que treballar quan un pas falla.

Un pipeline que produeix `BUILD FAILED` i res mes es inutil per a agents i humans per igual.

Inverteix en observabilitat del pipeline:

- **Logs estructurats.** JSON o parells clau-valor, no text lliure. Un agent pot parsejar `{"stage": "unit-tests", "failed": 3, "passed": 247, "failures": ["test_auth_flow", "test_rate_limit", "test_session_expiry"]}` i prendre accio significativa. No pot fer gaire amb `Tests failed. See above for details.`
- **Codis de sortida significatius.** No deixis que tot surti amb codi 1. Distingeix entre "tests fallats" (arreglable) i "no s'ha pogut connectar a la base de dades" (problema d'infraestructura, no es cosa de l'agent).
- **Emmagatzematge d'artefactes.** Resultats de tests, informes de cobertura, logs de build – emmagatzema'ls com a artefactes del pipeline. Un agent depurant una fallada de CI necessita llegir la sortida completa, no nomes les ultimes deu linies que caben a la vista de consola.

Considera afegir un `PIPELINE.md` al teu repo – una descripcio en llenguatge pla del que fa cada etapa de CI, com son les seves sortides esperades, i quins modes de fallada comuns existeixen. Un agent que pugui llegir aquest fitxer abans de depurar una fallada de CI funcionara dramaticament millor que un que hagi de fer enginyeria inversa del teu pipeline nomes des de la configuracio YAML. I els teus nous empleats tambe t'ho agrairan.

Un bon disseny de pipeline i una bona integracio d'agents es reforcen mutuament. Millores el teu CI per als agents, i els teus desenvolupadors humans tambe en surten beneficiats. Es una d'aquelles inversions rares que dona dividends en ambdues direccions.

13.7 Comencant Petit

Si has llegit aquest capitol i tens la temptacio de connectar agents a cada etapa del teu pipeline per divendres – no ho facis. He vist aquell impuls.

Normalment acaba amb un cap de setmana desfent l'automatitzacio perque l'equip no estava preparat per al volum de soroll generat per agents.

Els equips que tenen exit amb agents a CI son els que construeixen confianca incrementalment, igual que amb els agents interactius.

La bona noticia es que el cami esta ben recorregut. Aqui tens un full de ruta d'adopcio practic que he vist funcionar a multiples equips:

Setmana 1: Auto-formatat. Afegeix un pas de CI que executi un agent per arreglar problemes de format als PRs. El formatat es deterministic, facil de verificar i de baix risc. Si l'agent reformata alguna cosa incorrectament, el diff esta alla. Revisa els commits de l'agent la primera setmana per construir confianca.

Setmana 2: Pre-screening de PRs. Afegeix comentaris de revisio generats per agents als PRs. No bloquejants – nomes informatius. Deixa que el teu equip vegi que detecta l'agent, i calibra si la seva retroalimentacio es util o sorollosa. Ajusta el prompt basant-te en el que funciona.

Mes 1: Redaccio de changelog. Apunta l'agent al teu historial de commits per generar esborranys de notes de versio. Un huma edita i publica. Estas fent servir l'agent com a primer redactor, no com a decisor. Tambe es un bon moment per configurar monitoritzacio de costos i alertes.

Mes 3: Agents nocturns. A aquestes alcades has construir confianca en la sortida generada per agents i tens la infraestructura – worktrees, pressupostos de tokens, aillament de branques, processos de revisio. Comenca amb un sol tiquet ben acotat. Revisa el resultat curosament. Itera sobre el teu script nocturn. Expandeix gradualment a tasques mes complexes a mesura que creix la teva confianca.

Fixa't en que *no* es en aquest full de ruta: auto-fusio, desplegaments autonoms, ni agents prenent decisions arquitectoniques. Aixo no son l'etapa quatre. Podrien ser l'etapa deu, o podrien no ser mai apropiades per al teu equip. El full de ruta no es una marxa cap a l'automatitzacio completa – es una marxa cap al nivell correcte d'automatitzacio per al teu context.

Resisteix l'impuls de saltar-te passos. Cadascun es construeix sobre l'anterior. Aprens en que son bons els teus agents, on lluiten, i quines

baranes necessites. Al mes tres, el CI assistit per agents se sent natural – no perque ho hagis automatitzat tot de cop, sino perque t'has guanyat cada peca a traves de l'experiencia.

El pipeline es simplement un altre entorn on treballen els agents. S'apliquen els mateixos principis: acota estretament, verifica exhaustivament, confia incrementalment. L'unica diferencia es que ningu esta mirant, aixi que les teves xarxes de seguretat necessiten ser mes fortes.

14 Quan els Agents s'Equivoquen

Tots els enginyers que treballen amb agents prou temps tenen una col · leccio d'aquestes histories. Els moments on et recolzes a la cadira, mires la pantalla, i murmures alguna cosa impublicable. Son humiliants. Son educatius. I son *inevitables*.

Aquest capitol es una col · leccio d'histories de guerra – coses que realment van malament quan dones feina real a un agent. No hipotetics. No demos artificials. El tipus de fallades que et costen una tarda, un desplegament, o la teva fe en l'automatitzacio. Cadascuna es mapeja a principis de capitols anteriors, perque els principis son barats fins que els aprens per les males.

Llegeix-les abans de cometre els mateixos errors. O llegeix-les despres, i senteix-te menys sol.

14.1 El Refactoritzador Entusiasta

La tasca era simple: arreglar un problema de z-index en un menu desplegable. El desplegable es renderitzava darrere d'una capa de modal. Un arreglament CSS d'una linia – potser dos si vas amb compte.

L'agent va veure el component del desplegable, va decidir que el seu CSS era "inconsistent amb les practiques modernes," i el va refactoritzar. Despres va notar que el modal utilitzava un patro d'estil diferent, aixi que tambe el va refactoritzar. Despres els components de botons. Despres els components de targetes. Abans que aixequessis la vista del cafe, trenta-dos fitxers havien canviat. El diff era de 1.400 linies. L'agent havia essencialment reescrit el sistema d'estils de la biblioteca de components, migrant d'un enfocament a un altre amb una consistencia admirable i zero autoritzacio.

El bug original del z-index? Seguia alla. Enterrat en algun lloc de l'allau de canvis, l'agent havia reproduit el mateix problema de capes amb nous noms de classes.

El PR era irrevisable. No pots revisar significativament 1.400 linies de canvis de CSS a trenta-dos fitxers. El que *pots* fer es esborrar la branca i comencar de nou. Que es el que va passar.

Que fas diferent ara: Acota les tasques estretament. "Arregla el z-index del desplegable a `NavMenu.tsx`, no toquis res mes." Utilitza una branca dedicada perque el dany estigui contingut. I *sempre* comprova el diff abans de deixar l'agent passar a qualsevol altra cosa. Al moment que vegis el nombre de fitxers pujant mes enlla del que te sentit per a la tasca, atura l'agent. El capitol de baranes existeix per una rao.

14.2 La Biblioteca Al·lucinada

L'agent necessitava parsejar alguns rangs de dates complexos de l'entrada de l'usuari. Va importar `@temporal/daterange-parser`, va escriure codi elegant utilitzant el seu metode `.parseRange()` amb opcions per a parsing conscient de la localitat i matching difus. El codi era net. Els tipus eren correctes. La gestio d'errors era acurada. Fins i tot va escriure tests – que, naturalment, tambe importaven el paquet al · lucinat.

Tot es veia perfecte al diff. Les signatures de funcions tenien sentit. L'API estava ben dissenyada. Era una biblioteca *tan* plausible que gairebe no la vas qüestionar.

Despres `npm install` va fallar. El paquet no existia. Mai havia existit. L'agent havia inventat una biblioteca, li havia donat un nom creible i una API coherent, i havia escrit codi de produccio contra una ficcio.

La part insidiosa: si nomes hagues fet revisio de codi – llegint la logica, comprovant els tipus, avaluant l'enfocament – l'hauries aprovat. El codi era *bo*. Simplement no connectava amb la realitat.

Que fas diferent ara: L'agent executa els tests. Sempre. Si la teva configuracio no ho permet, els executes tu abans de revisar codi. Sense excepcions. Un `npm install` que falla es un senyal fort i obvi. Una API

al · lucinada que mai es va executar es una bomba silenciosa. Els tests detecten el que la revisio humana no detecta – no perque la teva revisio sigui dolenta, sino perque les ficcions plausibles estan *dissenyades* per passar la revisio. Aixo es el que *es* l'al · lucinacio.

14.3 El Bucle Infinit

Vas demanar a l'agent que arregles un test d'integracio que fallava. Va llegir l'error, va fer un canvi, va executar el test. Nou error. Va llegir *aquell* error, va fer un altre canvi, va executar el test. Error diferent. Despres una variacio del primer error. Despres alguna cosa nova. Despres el primer error una altra vegada.

Eres en un altre terminal, treballant en una altra cosa. Quaranta minuts despres vas comprovar. L'agent havia fet dinou intents. Havia cremat 200.000 tokens. El codi era ara pitjor que quan va comencar – un registre geologic d'arreglaments fallits apilats uns sobre els altres. L'agent seguia anant, alegrament confiat que l'intent vint seria el bo.

No ho va ser.

El problema fonamental era que l'agent no entenia *per que* el test fallava. Estava fent correspondencia de patrons amb missatges d'error i fent edicions locals, pero el problema real era un malentes arquitectonic – un estat de base de dades compartit entre casos de test que requeria un enfocament de configuracio completament diferent. Cap quantitat d'ajustos al codi sota test arreglaria un problema a la infraestructura de tests.

Que fas diferent ara: Configura limits d'iteracio. Tres intents al mateix problema es un maxim raonable. Si l'agent no ho ha resolt en tres passades, no ho resoldra en trenta. Quan vegis el bucle – error, arreglament, error diferent, arreglament, error original – *trenca'l manualment.* Atura l'agent, llegeix els errors tu, i dona-li un enfocament completament diferent o pren-ho tu. El temps de l'agent es barat. La teva tarda perduda no. Mes important, comenca un context fresc. La comprensio de l'agent ara esta contaminada amb dinou teories equivocades. Un inici net amb el teu diagnostic del problema *real* arribara mes lluny, mes rapid.

14.4 La Resposta Incorrecta Amb Confianca

Un treball en segon pla ocasionalment processava el mateix element dues vegades. Condicio de carrera classica. Vas apuntar l'agent al problema.

L'agent va analitzar el codi, va identificar la finestra de carrera, i va afegir un retard de 500ms entre la comprovacio i l'actualitzacio. "Aixo assegura que la transaccio anterior te temps de completar-se abans de la propera comprovacio," va explicar, amb la confianca serena d'algu que mai ha operat un sistema sota carrega.

Els tests van passar. El retard era mes llarg que el temps de transaccio del test, aixi que la finestra de carrera es va tancar – a l'entorn de test, amb un sol treballador concurrent, en una maquina tranquil · la.

Va anar a produccio. Sota carrega, amb dotze treballadors i latencia de base de dades variable, 500ms de vegades no era suficient. I ara tenies un *nou* problema: el retard havia reduit el rendiment prou perque la cua de treballs s'acumules durant les hores punta, creant timeouts en cascada que van tirar a terra un servei no relacionat.

El sleep no va arreglar la condicio de carrera. La va amagar a baixa concurrencia i va empitjorar el sistema a alta concurrencia. Un arreglament adequat – un advisory lock o una clau d'idempotencia – hauria estat correcte a qualsevol escala. L'agent va triar l'arreglament que feia el test verd, no l'arreglament que era *correcte*.

Que fas diferent ara: Tractes els tests que passen com a necessaris pero no suficients. Quan un agent arregla un problema de concurrencia, et preguntes *a tu mateix* si l'arreglament es correcte sota carrega, sota fallada, sota condicions que la suite de tests no cobreix. Els agents optimitzen pel senyal de retroalimentacio que els dones, i si aquell senyal es "els tests passen," trobaran el cami mes curt cap al verd – fins i tot si aquell cami es un `time.Sleep`. El teu judici d'enginyeria sobre *per que* alguna cosa funciona importa mes que *si* els tests passen. Aquesta es la part de la feina que encara no s'ha automatitzat. Aprofita-la.

14.5 L'Amnesia de Context

Dues hores dins d'una sessio, tenies una funcionalitat que evolucionava be. Havies dit a l'agent al principi: "No facis servir cap ORM – escrivim SQL en brut en aquest projecte. Es una decisio deliberada." L'agent ho va reconèixer i va escriure consultes netes i fetes a ma per a les primeres tasques.

Despres li vas demanar que afegis un nou endpoint. Noranta minuts de context s'havien acumulat. L'agent va construir l'endpoint – amb Prisma. Fitxer d'esquema complet, migracio, client generat. Codi bonic. Contradient completament la restriccio que havies establert al principi de la sessio i els patrons de cada altre fitxer que havia escrit aquell dia.

Quan ho vas assenyalar, l'agent es va disculpar, va reescriure tot en SQL en brut, i va actuar com si no hagues passat res. No havia *decidit* ignorar la teva restriccio. Simplement l'havia perdut. La finestra de context s'havia omplert amb prou feina intermedia que la instruccio inicial s'havia esvanit en la irrellevancia.

Que fas diferent ara: Les sessions llargues es degraden. Aixo es una propietat fonamental de com funcionen les finestres de context, no un bug que s'arreglara el proper trimestre. Mantin les tasques curtes i enfocades. Commiteja codi funcional en fronteres naturals perque el progres quedi capturat a git, no nomes a la conversa. Comenca sessions noves per a tasques noves. I per a restriccions de tot el projecte com "no ORM" o "no noves dependencies," posa-les en un fitxer `CLAUDE.md` o equivalent que l'agent llegeixi a l'inici. No confiis en que l'agent *recordi* el que vas dir fa dues hores. No ho fara. Escriu-ho.

14.6 L'Allau de Dependencies

Vas demanar un selector de dates. Un simple input on els usuaris puguin seleccionar una data. L'agent va avaluar les opcions i va decidir ser exhaustiu.

Va instal · lar `moment.js` per a la gestio de dates. Despres `@popperjs/core` per posicionar el desplegable. Despres una biblioteca de components

d'interficie completa perque "proporciona primitives de selector de dates accessibles." Despres un preprocessador de CSS perque el sistema de temes de la biblioteca de components ho requeria. Despres dos paquets d'utilitats que el selector de dates de la biblioteca de components necessitava com a dependencies parelles.

Sis noves dependencies. La mida del teu bundle va passar de 180KB a 540KB. El temps de build es va duplicar. Tenies un selector de dates, pero. Era molt maco.

El `<input type="date">` natiu d'HTML hauria estat be. O una sola biblioteca de selector lleugera de 8KB. En lloc d'aixo havies heretat tot un ecosistema perque l'agent va optimitzar per completesa en lloc de minimalitat.

La pitjor part no era la mida del bundle – era la superficie de manteniment. Sis nous paquets vol dir sis coses noves que poden tenir vulnerabilitats de seguretat, sis coses noves que es poden trencar en una actualitzacio, sis nous changelogs per llegir quan Dependabot comenci a obrir PRs. No nomes vas afegir un selector de dates. Vas adoptar sis projectes de codi obert.

Que fas diferent ara: Restringeix el que els agents poden instal · lar. "No noves dependencies sense preguntar-me primer" es una instruccio legitima i sovint *savia*. Quan permetis nous paquets, digues a l'agent les teves restriccions: pressupost de bundle, no paquets amb menys de N descàrregues setmanals, no paquets que arrosseguin mega-dependencies transitives. Els agents no tenen opinions sobre mida de bundle o carrega de manteniment. No mantindran el projecte l'any vinent. *Tu* si. Aixi que tu poses els limits.

14.7 El Fil Comu

Cadascuna d'aquestes histories te la mateixa causa arrel: l'agent estava fent *exactament el que estava dissenyat per fer*, i l'huma no estava proporcionant prou estructura.

El refactoritzador entusiasta estava sent servicial. La biblioteca al · lucinada estava sent creativa. El bucle infinit estava sent persistent. La

resposta incorrecta amb confianca estava sent guiada per tests. L'amnesia de context era una limitacio, no una eleccio. L'allau de dependencies estava sent exhaustiva.

Cap d'aquestes son bugs d'agents. Son bugs de *flux de treball*. L'arreglament mai es "utilitza un agent mes intel · ligent." L'arreglament sempre es el mateix: abast mes estret, millors bucles de retroalimentacio, mes estructura, sessions mes curtes, i un huma que segueix involucrat.

Els agents s'equivoquen. Els humans tambe. La diferencia es que els humans s'equivoquen prou lentament com per adonar-se'n. Els agents s'equivoquen a la velocitat de l'autocompletat, i quan aixeques la vista del cafe, trenta-dos fitxers han canviat.

Mantin-te al bucle. Comprova els diffs. Confia pero verifica. I quan les coses vagin malament – perque ho faran – recorda que el boto d'esborrar-branca-i-comencar-de-nou es l'eina mes infravalorada del teu flux de treball.

14.8 El Manual de Diagnostic

Les histories de guerra son entretingudes. Pero no pots enganxar anecdotes al teu monitor. El que necessites es un enfocament sistematic – una llista de comprovacio per quan l'agent produeix alguna cosa incorrecta, perque puguis diagnosticar la fallada rapidament i arreglar la cosa correcta en lloc de bracejar.

Quan un agent et dona mala sortida, passa per aquestes preguntes en ordre. La majoria de fallades cauen en una de sis categories, i saber en quina categoria ets determina que fas a continuacio.

1. Es un problema d'abast?

Has demanat massa en un sol prompt? El Refactoritzador Entusiasta va passar perque "arregla el z-index" era massa vague – no deia "no toquis res mes." Si l'agent ha canviat fitxers que no esperaves, o ha fet feina que no has demanat, l'abast era massa ample. Ajusta el prompt. Sigues explicit sobre que *no* fer. Les restriccions son mes utils que les instruccions quan tractes amb un agent entusiasta.

2. Es un problema de context?

Tenia l'agent el que necessitava per resoldre el problema real? Recorda la historia del bug de facturacio del Capitol 2 – un enginyer va donar context vague i va obtenir una resposta vaga, l'altre va donar fitxers especifics i traces d'error i va obtenir un arreglament funcional. Si l'agent va resoldre el problema *equivocat*, probablement no podia veure el correcte. Alimenta'l amb els fitxers rellevants, la sortida d'errors, les restriccions que no pot inferir. Els agents no endevinen be. Treballen amb el que els dones.

3. Es un problema de senyal de retroalimentacio?

Els teus tests estan realment verificant la cosa correcta? La Resposta Incorrecta Amb Confianca va passar perque els tests van passar – pero no testejaven sota condicions realistes. El sleep de 500ms "arreglament" era verd a CI i incorrecte a produccio. Si la solucio de l'agent se sent dubtosa pero els tests passen, *els teus tests son el problema*. L'agent va optimitzar pel senyal que li vas donar. Dona-li un senyal millor.

4. Es un problema de capacitat?

Algunes tasques genuinament estan mes enlla del que el model pot fer. Raonament multi-fitxer a traves d'un sistema extens amb dependencies implicites. Problemes de concurrencia subtils que requereixen entendre el comportament en temps d'execucio, no nomes l'estructura del codi. Logica sensible a la seguretat on "plausible" no es prou bo. Si l'agent segueix provant enfocaments diferents i fallant, potser no es capac d'aquesta tasca particular. Aixo no es un problema de prompt – es una limitacio. Reconeix-ho i pren-ho tu. El teu temps es millor invertit fent la feina que enginyant el prompt perfecte per a una tasca que l'agent no pot gestionar.

5. Es un problema de finestra de context?

Les sessions llargues es degraden. Punt. La historia de l'Amnesia de Context ho va demostrar – restriccions establertes aviat a una sessio simplement es perden a mesura que el context s'omple amb feina intermedia. Si l'agent contradiu alguna cosa que va fer correctament mes aviat a la mateixa sessio, o ignora una restriccio que vas establir fa una hora, la finestra de context es la culpable. Comenca una sessio nova. Torna a establir les restriccions rellevants. Dona-li nomes el context que necessita

per a la tasca actual, no el registre arqueologic de tot el que heu discutit avui.

6. Es un bucle?

Tres intents al mateix error es el limit. Si l'agent no ho ha resolt en tres passades, no ho resoldra en trenta. La historia del Bucle Infinit va cremar 200.000 tokens en dinou intents sense progres. Quan vegis el patro – error, arreglament, error diferent, arreglament, error original – trenca el bucle immediatament. Atura l'agent. Llegeix els errors tu. Dona a l'agent un enfocament completament diferent amb un diagnostic fresc, o pren-ho tu completament. El context de l'agent ara esta contaminat amb teories fallides, i mes intents nomes afegiran mes contaminacio.

Imprimeix aquesta llista de comprovacio. Enganxa-la al teu monitor. Les primeres vegades la recorreras conscientment. Despres d'un mes, es torna instint. Despres de tres mesos, detectaras el problema abans que l'agent ni tan sols acabi el seu primer intent.

15 Quan No Utilitzar Agents

Aquest llibre tracta sobre utilitzar agents be. Aquest capitol tracta sobre saber quan no utilitzar-los en absolut.

Si has llegit fins aqui, probablement estas convençut de l'enginyeria agentica. Be. Pero la manera mes rapida de perdre credibilitat – i perdre temps – es buscar un agent quan la feina demana un huma. Els millors enginyers agentics no son els que utilitzen agents mes. Son els que saben *exactament* quan deixar l'agent i fer la feina ells mateixos.

15.1 L'Impost del Sobrecost

Cada interaccio amb un agent te un cost. Escrius el prompt. Esperes la sortida. Revises el que ha produit. Arregles els errors. Tornes a executar si no ha captat la idea. Aixo es sobrecost, i es real.

Per a tasques complexes que et costarien una hora, dedicar dos minuts a prompt i revisio es una ganga. Pero per a tasques que pots fer en trenta segons? El sobrecost fa els agents *mes lents*, no mes rapids.

Renomenar una variable. Arreglar un error tipografic. Ajustar un valor de configuracio. Afegir una linia de log. Aquestes son tasques de memoria muscular. Els teus dits coneixen les pulsacions de tecles. Quan hagis acabat de teclejar un prompt descrivint el que vols, ja ho podries haver fet.

Aixo no es una fallada dels agents. Son matematiques. Les tasques petites tenen retorns petits, i el cost fix de la interaccio amb agents es menja el marge. No deixis que la novetat dels agents et faci caure en la trampa d'utilitzar-los per a tot. Alguna feina es simplement mes rapida a ma.

15.2 Decisions d'Arquitectura Novel·les

Quan estas dissenyant un sistema des de zero – triant entre un monolit i microserveis, decidint el teu model de dades, triant els teus patrons de comunicacio – el *pensament es la feina*. El valor no es al diagrama o al document. El valor es al model mental que construeixes mentre lluites amb els compromisos.

Un agent pot ajudar-te a explorar opcions. Pot llistar els pros i contres de l'event sourcing versus CRUD. Pot esbossar com podria ser una arquitectura particular. Aixo es util com a input.

Pero delegar l'arquitectura mateixa a un agent vol dir que no entens el teu propi sistema. Tindras dificultats per depurar-lo, ampliar-lo, o explicar-lo al teu equip. La feina de l'enginyer senior es prendre les decisions dificils – pesar els compromisos que no tenen respostes netes, decidir quina complexitat val la pena carregar i quina no. Aquell judici ve de fer la feina, no de llegir el resum d'un agent.

Utilitza agents com a sparring partner per a l'arquitectura. No com a arquitecte.

15.3 Codi Critic de Seguretat

Fluxos d'autenticacio. Xifratge. Control d'acces. Validacio d'entrada. Gestio de tokens. Aquestes son arees on "sembla correcte" no es prou bo.

Els bugs de seguretat son diferents dels bugs normals. Una funcio d'ordenacio trencada produeix sortida incorrecta que algu nota. Una comprovacio d'autenticacio trencada no produeix *cap simptoma visible* fins que un atacant la troba. El codi es veu be. Els tests passen. I sis mesos despres estas escrivint un informe d'incident.

Els agents produeixen codi plausible. Aquesta es la seva fortalesa i, en contextos de seguretat, el seu perill. Un defecte subtil en un flux de validacio de JWT, una comprovacio absent en una URL de redireccio, un canal lateral de temps en una comparacio de contrasenyes – aquests son el tipus d'errors que sobreviuen la revisio de codi perque *semblen correctes*.

Escriu el codi critic de seguretat tu. Revisa'l curosament. Aconsegueix un segon parell d'ulls humans. Si utilitzes un agent per esbossar codi de seguretat, tracta aquell esborrany amb mes sospita de la que donaries al primer intent d'un desenvolupador junior, no menys.

15.4 Quan Necessites Aprendre

Estas aprenent un nou llenguatge. Un nou framework. Un nou paradigma. La temptacio es obvia: deixa que l'agent escrigui el codi mentre tu et centres en la imatge general.

Resisteix-la.

Si deixes que l'agent escrigui tot el Rust mentre estas aprenent Rust, no has apres Rust. Has construit alguna cosa que no pots mantenir, depurar, ni ampliar sense l'agent. Has creat una dependencia, no una habilitat.

Hi ha una diferencia crucial entre utilitzar un agent per *explicar* alguna cosa i utilitzar-lo per *fer* alguna cosa. Preguntar "per que es produeix aquest error del borrow checker?" construeix comprensio. Preguntar "arregla aquest error del borrow checker" no.

Quan l'objectiu es aprendre, frena. Escriu el codi tu. Comet els errors tu. Utilitza agents com a tutors, no com a escriptors fantasma. La comprensio que construeixes lluitant a traves de les parts dificils es tota la qüestio.

15.5 Decisions Emocionalment Carregades

No totes les decisions d'enginyeria son tecniques. Algunes de les mes dificils son humanes.

Deprecar una API de la qual un soci depen. Dir a un stakeholder que la seva peticio de funcionalitat no entrara. Resistir un termini que saps que no es realista. Decidir retirar un producte que encara te usuaris.

Aquestes decisions requereixen empatia. Requereixen llegir l'ambient, entendre la politica, pesar el cost huma al costat del cost tecnic. Requereixen *responsabilitat* – algu que sigui propietari de la decisio i les seves consequencies.

Un agent pot redactar el correu. Pot ajudar-te a pensar en els punts de conversa. Pero la decisio mateixa, i la conversa que la lliura, han de venir d'un huma. La gent mereix rebre males noticies d'una persona, no d'algu que ha copiat i enganxat la sortida d'un agent.

15.6 Quan la Base de Codi Es Massa Desordenada

Els agents amplifiquen el que ja hi ha. En una base de codi neta i ben estructurada amb convencions fortes, els agents produeixen codi net i ben estructurat. Recullen els patrons i els segueixen.

En un desastre? Produeixen mes desastre.

Si la teva base de codi te nomenclatura inconsistent, dependencies enredades, cap frontera de moduls clara, i tres maneres diferents de fer el mateix – un agent recollira *tots* aquells patrons. Podria combinar les pitjors parts de cadascun. No sap quins patrons son intencionals i quins son deute tecnic. Simplement veu el que hi ha i produeix mes del mateix.

De vegades el moviment correcte es netejar abans de portar agents. Refactoritza el modul. Estableix la convencio. Esborra el codi mort. Fes de la base de codi un lloc on un agent pugui fer bona feina. Aixo es feina poc glamurosa i manual. Pero es la base que fa possible tota la resta.

Pensa-ho com un taller. No dones eines electriques a algu en un taller desordenat i desorganitzat. Primer neteges, *despres* portes les eines.

15.7 Treballar amb Codi Legacy

Aquella metafora del taller es maca. Pero no tothom te el luxe de netejar primer. Alguns de nosaltres mantenim monolits Java de 500.000 linies que es van iniciar el 2014 i han passat per tres migracions de framework, dues adquisicions, i un breu periode on algu va pensar que la configuracio XML era una bona idea. El consell de "refactoritza abans de portar agents" es solid en teoria i risible a la practica quan estas mirant una base de codi que trigaria anys a refactoritzar.

Aixi que *com* fas servir agents en codi legacy? Amb cura. I amb una estrategia especifica.

Comenca pels tests. Abans d'apuntar un agent a codi legacy, escriu tests de caracteritzacio – tests que capturen el que el codi *fa actualment*, no el que *hauria* de fer. No son aspiracionals. Son descriptius. Diuen "quan crides aquesta funcio amb aquestes entrades, aixo es el que surt, i aquest es el contracte actual tant si algu ho pretenia com si no."

Els tests de caracteritzacio donen a l'agent una xarxa de seguretat. Sense ells, l'agent canviara comportament que no enten, i no ho sabras fins que produccio t'ho digui. Amb ells, qualsevol canvi de comportament apareix com una fallada de test *abans* que el codi surti de la teva maquina. Aixo es no negociable. Codi legacy sense tests es codi legacy que no pots deixar que els agents toquin amb seguretat.

Acota despiadadament. En una base de codi legacy, l'agent no pot entendre tot el sistema. No intentis que ho faci. Apunta'l a un fitxer, una funcio, un bug. Dona-li el context immediat – la signatura de la funcio, els cridadors, el comportament esperat – i res mes. El codi legacy esta ple de suposicions implicites, efectes secundaris no documentats, i peculiaritats que aguanten pes. Com menys vegi l'agent, menys pot malinterpretar. Un abast estret amb context clar guanya a un abast ampli amb complexitat arqueologica.

Utilitza agents per a les parts tedioses. Les bases de codi legacy estan plenes de feina mecanica: actualitzar crides d'API deprecades a centenars de fitxers, migrar d'una versio de dependencia a una altra, afegir anotacions de tipus a codi sense tipar, estandarditzar patrons de gestio d'errors, substituir un framework de logging per un altre. Aquestes son tasques *perfectes* per a agents – repetitives, ben definides, facilment verificables per comprovacions automatiques. Deixa que els agents facin la feina dura. Guarda la teva energia per a les parts que requereixen entendre *per que* el codi es com es. Aquesta es la feina que nomes un huma que ha viscut amb el sistema pot fer.

Documenta a mesura que avances. Cada vegada que un agent treballa en un fitxer legacy amb exit, afegeix un breu comentari explicant que fa el codi, o actualitza el `CLAUDE.md` del projecte amb context sobre aquell

modul. Amb el temps, passa alguna cosa interessant: les parts de la base de codi legacy que els agents han tocat es tornen millor documentades que les parts que no. No perque t'hagis proposat documentar el sistema, sino perque fer servir agents *et va forcar* a articular que fa cada peca per poder fer prompts efectivament. Els agents estan fent lentament la base de codi mes llegible – no refactoritzant-la, sino fent-te explicar-la.

15.8 L'Argument de l'Ofici

Hi ha una rao mes per de vegades deixar l'agent, i no es sobre eficiencia o risc. Es sobre l'ofici.

Escriure codi a ma construeix alguna cosa que els agents no et poden donar. Memoria muscular. Reconeixement de patrons que viu als teus dits, no nomes al teu cap. La comprensio profunda i intuitiva que ve d'haver escrit el mateix tipus de funcio dotzenes de vegades. La satisfaccio silenciosa de resoldre un problema dificil a traves del teu propi raonament.

Aquestes coses importen. No perque siguin romantiques, sino perque et fan un millor enginyer. El desenvolupador que ha escrit un parser a ma enten el parsing de manera diferent que un que nomes ha demanat a un agent que en faci un. El desenvolupador que ha depurat un problema de concurrencia a les 2 de la matinada *sent* el perill de l'estat mutable compartit d'una manera que llegir-ne no proporciona.

Els agents son eines. Potents. Pero si els deixes fer tota la feina, les teves propies habilitats s'atrofien. I quan topis amb un problema que l'agent no pot resoldre – i ho faras – necessites que aquelles habilitats segueixin afilades.

Segueix practicant. Escriu codi a ma regularment. No perque sigui mes rapid, sino perque et mante perillos.

15.9 El Judici Que Importa

L'habilitat central de l'enginyeria agentica no es el prompting. No es la configuracio d'eines. No es saber quin model utilitzar per a quina tasca.

Es el *judici*.

Saber quan un agent t'estalviara una hora i quan en malbaratara una. Saber quan confiar en la sortida i quan reescriure-la des de zero. Saber quan delegar i quan endinsar-s'hi tu.

Els millors enginyers agentics utilitzen agents agressivament – pero selectivament. Busquen agents quan el palanquejament es real: refactoritzacions a gran escala, generacio de boilerplate, exploracio de codi, escriptura de tests, documentacio. I deixen l'agent quan la feina requereix pensament huma, responsabilitat humana, o ofici huma.

Aquell judici es el que separa els enginyers agentics dels simples operadors de prompts. Qualsevol pot teclejar un prompt. Saber *quan no fer-ho* es l'habilitat mes dificil.

16 Equips Agentics

El software sempre ha estat un esport d'equip. Els agents no canvien aixo. Pero canvien la forma de l'equip, el ritme de la feina, i les habilitats que importen. Si lideres un equip – o hi treballes – necessites pensar-hi ara, no despres.

16.1 El Multiplicador de 10x Es Real, Pero Es Distribueix Diferent

Has sentit l'afirmacio: els agents fan els enginyers 10x mes productius. No es incorrecta. Nomes es engannyosa.

Els agents fan certes tasques *dramaticament* mes rapides. Generar boilerplate, escriure infraestructura de tests, refactoritzar a cent fitxers, migrar versions d'API, cablear endpoints CRUD – passen d'hores a minuts. El guany de velocitat es real i es massiu.

Pero altres tasques a penes canvien. El disseny de sistemes segueix requerint pensar. Depurar una condicio de carrera subtil segueix requerint paciencia, intuicio i comprensio profunda del runtime. Les converses amb stakeholders segueixen trigant el que triguen. L'agent no pot seure a la teva revisio d'arquitectura i fer les preguntes correctes.

El multiplicador de 10x no es un multiplicador pla al llarg del teu dia. Es un grafic d'espigues. Algunes tasques van 50x mes rapides. Algunes van 1x. Unes poques fins i tot es poden frenar si lluites contra l'agent en lloc de fer-ho tu.

Els equips que entenen aixo despleguen agents estrategicament. No donen cada tasca a un agent i esperen magia. Identifiquen les zones d'alt palanquejament – les tasques on els agents genuinament comprimeixen els terminis – i centren l'esforc dels agents alla. La resta es queda humana.

16.2 La Revisio de Codi Canvia

Aqui tens que passa quan els agents entren al flux de treball d'un equip: el volum de PRs puja. *Molt.* Un enginyer que solia obrir dos PRs al dia ara n'obre cinc. El codi d'aquells PRs es sintacticament correcte, ben formatat, i passa els tests. I revisar-lo es *esgotador*.

El model antic de revisio de codi – llegir cada linia, buscar errors d'un-per-un, verificar gestio de casos limits – no escala quan la meitat del codi va ser generat en segons. Cremaras els teus revisors en una setmana.

El canvi es de "es correcte?" a "es l'enfocament correcte?" El codi generat per agents normalment es *localment* correcte. Fa el que es va demanar. La questio es si el que es va demanar era la cosa correcta. Necessita existir aquest nou servei, o hauria d'estar aquesta logica dins l'existent? Es la frontera d'abstraccio correcta? Fa aquest canvi el sistema mes simple o mes complex?

Els revisors es converteixen en arquitectes. Amplien la visio. Comproven intencio, no implementacio.

Adaptacions practiques que funcionen:

- PRs mes petits i mes enfocats – mes facils per a agents i revisors
- Comprovacions automatiques gestionen la part mecanica (linting, cobertura de tests, verificacio de tipus)
- El temps de revisio esta protegit al calendari, no espremut entre reunions
- L'equip acorda "patrons de confianca" – si un PR segueix un patro conegut i passa CI, te una via de revisio mes rapida

La fatiga de revisio es l'assassi silencis dels equips assistits per agents. Pren-t'ho seriosament.

16.3 La Questio de l'Enginyer Junior

Aixo es el problema mes dificil d'aquest capitol, i no tinc una resposta neta.

Els enginyers juniors tradicionalment aprenien fent la feina que ara els agents fan mes rapid i millor. Escriure aquell primer endpoint CRUD.

Lluitar amb un layout CSS complicat. Esbrinar per que el test falla. Aquestes tasques eren tedioses per als seniors pero *formatives* per als juniors. La lluita era l'educacio.

Si els agents gestionen tot aixo, on passa l'aprenentatge?

El pitjor resultat son juniors que es tornen dependents dels prompts – poden aconseguir que un agent generi codi, pero no poden explicar que fa el codi o depurar-lo quan es trenca. Es salten completament la fase de comprensio. Aixo no es enginyeria; es un flux de treball de copiar i enganxar molt car.

El cami millor es utilitzar agents com a *tutors*, no substituts. Un junior escrivint una migracio de base de dades hauria de demanar a l'agent que *expliqui* la migracio, no nomes que la generi. "Per que has posat una transaccio aqui?" "Que passa si aixo falla a mitges?" "Mostra'm com es veu sense l'ORM." L'agent es converteix en un professor pacient amb temps infinit – alguna cosa que la majoria de seniors no poden oferir.

La programacio en parella amb agents, *supervisada per seniors*, es el model mes prometedor que he vist. El junior dirigeix l'agent. El senior observa, fa preguntes, i intervé quan el junior accepta alguna cosa que no enten. Es mes lent que deixar l'agent fer-ho tot, pero produeix enginyers que realment saben el que fan.

Els equips que es salten aquesta inversio estan prenent en prestec del futur. Els juniors sense supervisio d'avui son els seniors de dema que no poden depurar produccio sense una crossa d'IA.

16.4 Distribucio de Coneixement

Aqui tens un patro que apareix a cada equip que adopta agents de manera desigual: un enginyer es torna fluid amb agents, comenca a lliurar al triple de velocitat que la resta, i en pocs mesos ha tocat cada part de la base de codi. Es converteix en l'unic punt de fallada.

El bus factor cau a u. No perque ningu ho hagi planificat, sino perque velocitat i concentracio de coneixement estan correlacionats. L'enginyer

que lliura mes apren mes. Els altres es queden enrere, no nomes en output sino en *comprensio* del sistema que col · lectivament posseeixen.

Aixo es un problema de gestio, no de tecnologia. L'arreglament es estructural:

- **Fitxers `CLAUDE.md` compartits.** Cada projecte en te un. Tothom hi contribueix. Codifica el coneixement col · lectiu de l'equip, no el d'una sola persona.
- **Fluxos de treball i convencions compartits.** L'equip acorda com utilitza agents – quines eines, quins patrons, quines baranes. Sense configuracions de llop solitari.
- **Rotacio.** Els enginyers fluids amb agents roten a diferents parts de la base de codi. El coneixement s'escampa a traves de la feina, no de la documentacio.
- **Comparticio de sessions d'agents.** Alguns equips han comencat a compartir sessions d'agents interessants – els prompts, les sortides, les decisions. Es una forma de transferencia de coneixement que no existia abans.

L'objectiu no es frenar el teu enginyer mes rapid. Es assegurar-se que el coneixement de l'equip segueix el ritme del output de l'equip.

16.5 Les Convencions Compartides Importen Mes

Vam cobrir les convencions a un capitol anterior. En un context d'equip, el que hi ha en joc es mes alt.

Quan un enginyer individual utilitza agents, les seves convencions afecten una persona. Quan un equip utilitza agents, les convencions afecten *cada sessio d'agent a tot l'equip*. Un projecte ben estructurat amb nomenclatura clara, patrons consistents, i un `CLAUDE.md` mantingut vol dir que els agents de cada enginyer comencen des d'una base solida.

Un projecte desordenat vol dir que cada agent reinventa la roda. Diferents enginyers obtenen sortides diferents. La base de codi deriva. Les revisions es tornen mes dificils perque ara estas revisant no nomes el codi sino l'*estil* del codi, que varia segons l'agent de quin enginyer l'hagi escrit.

Els estandards de codi en un equip agentic no son sobre estetica. Son sobre *efectivitat dels agents*. L'equip que acorda estructura del projecte, convencions de nomenclatura, patrons de testing, i format de documentacio obte millor output de cada sessio d'agent. Es un multiplicador de forca sobre un multiplicador de forca.

Inverteix el temps. Escriu els estandards. Imposa'ls a CI. Es retorna a cada PR.

16.6 El Nou Standup

Com es la coordinacio diaria quan cada enginyer esta executant multiples sessions d'agents en paral · lel?

L'standup antic: "Ahir vaig treballar en la refactoritzacio d'autenticacio. Avui continuare. Sense blocadors."

El nou standup: "Tinc tres agents executant-se. La refactoritzacio d'autenticacio ha aterrat i esta en revisio. La migracio d'API esta al 80% – l'agent s'ha encallat amb el parsing XML legacy, aixi que estic agafant-ho manualment. Acabo de llancar una tercera sessio per generar tests d'integracio per al modul de facturacio."

La granularitat canvia. La feina es mou mes rapid, aixi que la coordinacio necessita seguir el ritme. Un enginyer podria *comencar i acabar* una tasca entre standups. La sincronitzacio diaria es converteix menys en actualitzacions de progres i mes en alineament d'intencio – assegurant-se que tres enginyers no estan enviant agents al mateix problema des d'angles diferents.

Alguns equips estan passant a standups asincrons amb check-ins mes frequents. Altres utilitzen dashboards compartits que fan seguiment de sessions d'agents actives. La resposta correcta depen de l'equip, pero l'antic ritme de "una actualitzacio per persona per dia" sovint no es suficient.

16.7 Compliment i Rastre d'Auditoria

Si un agent escriu codi que causa un incident de produccio, qui es el responsable? L'enginyer que va fer el prompt? El revisor que va aprovar el PR? El lider d'equip que va decidir adoptar fluxos de treball agentics? Aixo no es una pregunta filosofica que debats davant unes cerveses. Es una questio legal i de compliment, i les industries regulades necessiten respostes clares *abans* que passi l'incident, no durant el postmortem.

La bona noticia es que la resposta no es realment tan complicada. Les eines i processos ja existeixen. Nomes necessites ser explicit sobre ells.

Git es el teu rastre d'auditoria. Cada commit generat per un agent hauria de ser atribuible. El missatge de commit hauria d'indicar que va ser assistit per agent – una etiqueta `Co-Authored-By`, un prefix, qualsevol convencio que adopti el teu equip. El PR hauria de mostrar qui l'ha revisat. L'aprovacio de fusio es la signatura. Aixo ja es com funcionen la majoria d'equips; la clau es fer-ho *consistent*. Atribucio ad-hoc – de vegades etiquetar, de vegades no – es pitjor que cap sistema, perque crea la impressio d'un proces sense la fiabilitat d'un.

El revisor n'es responsable. La resposta practica per a la majoria d'organitzacions es senzilla: l'enginyer que revisa i aprova el PR assumeix la responsabilitat, igual que ho faria per a qualsevol codi de qualsevol font. El codi generat per agents no te un estandard de responsabilitat diferent. Si aproves un PR, estas dient "he revisat aixo i crec que es correcte." L'eina que va generar el codi es irrellevant per a aquella afirmacio. Aixo tambe vol dir que les revisions de codi generat per agents necessiten ser revisions *reals*, no segells de goma. Si el volum de PRs generats per agents esta fent impossible una revisio exhaustiva, aixo es un problema de flux de treball a resoldre, no un estandard a rebaixar.

Per a industries regulades. Documenta el teu flux de treball agentic com a part de la teva documentacio d'SDLC. Quins models s'utilitzen, quina versio, quines baranes hi ha, quin proces de revisio passa el codi generat per agents abans d'arribar a produccio. Els auditors volen veure un *proces*, no perfeccio. Un proces documentat que inclou "generacio de codi assistida per IA amb revisio humana obligatoria i verificacio de CI" es auditable. Un proces no documentat on els enginyers utilitzen les eines que

volen sense cap enfocament consistent no ho es. Si ets a fintech, sanitat, o qualsevol cosa amb supervisio regulatoria, documenta-ho abans que algu ho demani.

Guarda logs de sessions. Rete logs de sessions d'agents, especialment per a codi que toca sistemes sensibles – facturacio, autenticacio, gestio de dades, qualsevol cosa amb implicacions regulatories. No perque els llegiras rutinariament, sino perque podries necessitar-los durant una revisio d'incidents. "Que va veure l'agent quan va generar aquest codi? Quin prompt va produir aquesta sortida? Amb quin context estava treballant?" Aquestes son preguntes que vols poder respondre sis mesos despres. La majoria d'eines agentiques poden exportar o registrar sessions. Configura la retencio abans de necessitar-la. El cost d'emmagatzematge es trivial comparat amb el cost de no tenir els logs quan compliment ve a trucar.

16.8 La Contractacio Canvia

Si els agents gestionen les parts mecaniques del codi, que necessites realment d'un enginyer?

La velocitat bruta de codi importa menys. L'enginyer que podia escriure una cerca binaria perfecta a la pissarra en tres minuts te una habilitat que els agents han commodititzat. Aixo no es inutil – entendre algoritmes segueix important – pero ja no es el diferenciador.

El que importa mes:

- **Disseny de sistemes.** La capacitat de descompondre un problema en components, definir interficies, i raonar sobre compromisos. Els agents poden implementar un disseny. No poden dir-te quin disseny es correcte per a les teves restriccions.
- **Judici.** Saber quan utilitzar l'agent i quan pensar. Saber quan la sortida de l'agent es incorrecta encara que sembli correcta. Saber quines dreceres prendre i quines protegir.
- **Comunicacio.** La capacitat d'articular intencio clarament – als agents, als companys d'equip, als stakeholders. Pensadors vagues obtenen sortida d'agents vaga. Pensadors precisos obtenen resultats precisos.

- **Descomposicio de problemes.** Dividir una tasca gran en peces de mida d'agent es una habilitat. Esta relacionada amb el disseny de sistemes pero es mes tactica. L'enginyer que pot convertir un epic de Jira en deu prompts d'agents ben acotats superara el que enganxa tot l'epic en una finestra de xat.

L'entrevista que pregunta "implementa aquest algoritme a la pissarra" esta testejant una habilitat que importa menys cada any. L'entrevista que pregunta "aqui tens un sistema amb aquestes restriccions – explica'm com el construiries, quins compromisos faries, i com verificaries que funciona" esta testejant les habilitats que importen *mes* cada any.

Contracta per judici. Sempre pots donar-los un agent per a la resta.

17 Paraules Finals

El meu fill gran té cinc anys. Encara no sap llegir, no realment — deletreja paraules a les capses de cereals i als rètols dels carrers, orgullós de cada síl · laba. Però sap què fa el meu ordinador. M'ha vist parlar-hi, ha vist text aparèixer a la pantalla en resposta, m'ha vist assentir o negar amb el cap i tornar a parlar. Un vespre es va enfilar a la meva falda, va mirar un agent refactoritzant un mòdul en temps real — fitxers obrint-se i tancant-se, tests executant-se, marques verdes apareixent — i va dir: "L'ordinador s'està arreglant sol?"

No tenia una bona resposta. Encara no la tinc, no del tot. Però aquella pregunta m'ha acompanyat durant cada capítol d'aquest llibre. Perquè el que vaig adonar-me, assegut allà amb ell, és que no estava escrivint un llibre sobre eines. Estava escrivint un llibre sobre què significa ser enginyer en el moment exacte en què la definició d'enginyeria s'està reescrivint — i sobre què ens emportem cap al que vingui després.

Aquest capítol no és un resum. No necessites que et recapituli quinze capítols que acabes de llegir. Això és el que vull dir-te directament, d'enginyer a enginyer, abans que ens acomiadem.

17.1 El Que Crec

Crec que l'ofici no s'està morint. S'està *comprimint*. Una dècada de boilerplate, fontaneria i cerimònia s'està col · lapsant en intenció i judici. El que queda quan treus el teclejar és allò que sempre va ser la feina de veritat: saber què construir i saber quan està bé.

Crec que els agents són amplificadors, no substituts. Dona un agent a un enginyer mediocre i obtindràs programari mediocre a una velocitat aterridora. Dona'n un a un gran enginyer i obtindràs quelcom que, francament, fa que els últims vint anys de desenvolupament de programari semblin com si estiguéssim construint autopistes amb pales.

Crec que els enginyers que prosperaran seran els que dominin les coses *avorrides* — context, guardrails, testing, convencions, pensament clar — mentre tothom persegueix l'últim anunci del nou model llampant. Les eines canvien cada trimestre. El judici es compon al llarg d'una carrera.

Crec que no ens estan substituint. Ens estan ascendint. De mecanògrafs a capitans. De programadors a *enginyers*, en el sentit més ampli de la paraula. La pregunta és si acceptes l'ascens.

Crec que aquest canvi és *bo*. No fàcil. No indolor. Però bo. Perquè la part de la nostra feina que s'està automatitzant mai va ser la part que estimàvem. Ningú es va fer enginyer perquè somniava amb escriure parsers de JSON repetitius. Ens vam fer enginyers perquè volíem construir coses que importen. Ara podem.

17.2 En Què M'he Equivocat

Seré honest amb tu: vaig canviar d'opinió almenys tres vegades mentre escrivia aquest llibre.

Quan vaig començar el capítol sobre sandboxes, pensava que l'aïllament amb contenidors era excessiu per a la majoria de fluxos de treball. Quan el vaig acabar, després que un agent fes un rm -rf d'un directori que m'importava un dimarts a la tarda, creia que el sandboxing era innegociable. Aquella experiència va acabar al capítol. La convicció darrere és teixit cicatricial.

Originalment vaig escriure el capítol de multi-agent amb la suposició que orquestrar cinc o sis agents simultàniament era l'estat final natural — una planta de fàbrica de treballadors autònoms. He fet marxa enrere. La sobrecàrrega de coordinació és real, els modes de fallada es multipliquen, i he descobert que dos o tres agents ben dirigits superen sis de no supervisats gairebé sempre. El capítol reflecteix on he acabat, però potser acabaré en un lloc diferent en sis mesos.

També vaig ser, durant un temps, massa desdenyós amb els models locals. Vaig escriure un esborrany inicial que bàsicament deia "simplement fes servir les APIs comercials." Llavors vaig passar un cap de setmana executant un model local afinat en una base de codi amb restriccions

propietàries i vaig adonar-me que hi ha tot un món de casos d'ús on local no és només viable sinó *necessari*. El capítol sobre models locals versus comercials existeix perquè m'equivocava i vaig haver de corregir-me.

Parts d'aquest llibre probablement ja són errònies de maneres que encara no puc veure. El paisatge es mou així de ràpid. Però les eines específiques mai van ser el punt. Si t'he ajudat a construir un model mental — una manera de pensar sobre autonomia, confiança i estructura — llavors el llibre ha fet la seva feina, fins i tot quan cada exemple de codi estigui obsolet.

17.3 La Metàfora de la Tripulació, Per Última Vegada

Vaig créixer a Dinamarca, prop de l'aigua. Si has navegat per aigües escandinaves, coneixes la llum a finals d'estiu — baixa i daurada, el tipus de llum que fa que el mar sembli coure martellejat. Recordo una travessia, un vaixell petit, quatre de nosaltres. El vent va girar fort i de sobte tots ens movíem sense parlar. Una persona al floc, una a l'escota major, una trimant, una al timó. Sense ordres. Només confiança construïda al llarg de dotzenes de navegades anteriors.

Així és com se sent l'enginyeria agèntica en un bon dia. Tu al timó, els agents trimant i ajustant, la feina fluint perquè has invertit les hores per construir context compartit — a través de convencions, a través de guardrails, a través de suites de test que atrapen errors abans que importin. No necessites cridar instruccions. El sistema *sap*.

I llavors hi ha els dies dolents. Els dies que el vent cau i un agent al·lucina una API que no existeix, o reescriu un mòdul que no li has demanat tocar, o passa tots els tests perquè ha eliminat els que fallaven. Aquells dies, estàs traient aigua i maleint. Això també és navegar.

Però hauria de ser honest sobre una cosa, perquè aquest llibre no funciona si no ho sóc.

La metàfora d'una *tripulació* implica lleialtat. Continuïtat. Companys de bord amb qui has navegat abans, que coneixen els teus hàbits, que

anticipen la següent ordre. És una imatge bonica. Però tampoc és com treballo realment.

La majoria de dies, el que realment faig és llançar un agent, donar-li una feina, agafar el resultat i llançar-lo per la borda. Llavors en llançar un altre. No recorden la sessió anterior. No saben què vaig demanar a l'agent anterior. Cadascun arriba fresc, fa la seva feina i desapareix. És menys una tripulació lleial i més com contractar estibadors a cada port — els informes, els mires treballar, els pagues i al següent port ho tornes a fer.

I està bé. Així és com funcionaven la majoria de tripulacions reals al llarg de la història marítima. El vaixell era la continuïtat. El capità era la continuïtat. Les cartes, el diari de bord, l'aparell — això persistia entre viatges. La tripulació sovint s'assemblava per a una sola travessia i es dissolia a la destinació. El que ho feia funcionar no era que els mariners coneguessin el capità. Era que el capità coneixia el *vaixell* — i tenia sistemes prou bons perquè qualsevol mariner competent pogués pujar a bord i ser útil.

Això és la teva base de codi. Això són les teves convencions, les teves suites de test, els teus fitxers CLAUDE.md, els teus guardrails. Són el vaixell. Cada nou agent que llances és un nou membre de la tripulació pujant a bord d'un vaixell ben aparellat. No necessiten conèixer la teva història. Necessiten conèixer el vaixell. I si has construït bé el vaixell, seran productius en minuts.

Així que sí — llança'ls per la borda. Llança'n de nous. Això no és un fracàs de la metàfora. És la metàfora funcionant exactament com estava previst. La tripulació és prescindible. El vaixell no.

Tu ets el capità. Sempre has estat el capità. La tripulació acaba d'arribar — i continuaran arribant, frescos i preparats, cada vegada que els necessitis.

17.4 Gràcies

Gràcies per llegir aquest llibre. Ho dic de veritat. Has canviat el teu temps i atenció per les meves paraules, i no m'ho prenc a la lleugera. Espero haver-m'ho guanyat.

Gràcies a la comunitat — els enginyers en fòrums, en servidors de Discord, en repos de codi obert — que estan compartint els seus experiments, els seus fracassos, les seves idees guanyades amb esforç. Aquest llibre ha estat modelat per centenars de converses que no vaig tenir sol.

I gràcies, suposo, als mateixos agents — que m'han ajudat a escriure codi, depurar problemes i ocasionalment generar prosa tan dolenta que em recordava per què el judici humà encara importa. Sou una bona tripulació. Esteu millorant. I sospito que quan el meu fill sigui prou gran per llegir aquest llibre, sereu quelcom que cap de nosaltres va predir del tot.

17.5 Vés i Construeix Alguna Cosa

Això és el que vull que facis.

Aquesta nit — no demà, no la setmana que ve, *aquesta nit* — obre el teu terminal. Tria un bug que has estat evitant. Aquell que continues movent al final del backlog, el que és molest però no urgent, el que viu en una part de la base de codi que preferiries no tocar. Apunta un agent cap a ell. Dona-li context. Posa un guardrail. Mira què passa.

Potser arregla el bug en quatre minuts i sents el terra moure's sota els teus peus, de la mateixa manera que es va moure sota els meus aquell vespre amb l'script de migració. Potser fa un desastre i aprens alguna cosa sobre com donar millors instruccions. De qualsevol manera, sabràs més del que sabies abans.

Després vés més gran. Configura aïllament amb worktree i executa dos agents en paral · lel. Escriu un fitxer CLAUDE.md per a un projecte que t'importi. Construeix una suite de tests prou bona per ser la teva xarxa de seguretat quan els agents facin commits. Refactoritza aquell mòdul del qual tothom té por — però aquesta vegada, amb una tripulació.

Després vés encara més gran. Introdueix fluxos de treball agèntics al teu equip. Comparteix el que has après — les victòries *i* els desastres. Escriu les teves pròpies històries de guerra. Contribueix al coneixement col · lectiu d'una disciplina que encara s'està inventant.

Perquè això és el que és: una disciplina que s'està inventant, ara mateix, per la gent disposada a fer la feina. No per la gent que escriu posts de blog sobre el futur. No per la gent que espera l'eina perfecta. Per la gent que obre un terminal avui i construeix alguna cosa real amb el que té.

Els enginyers que definiran aquesta era no estan esperant permís. No estan esperant certesa. Estan enviant, trencant coses, aprenent i tornant a enviar — amb una tripulació al seu costat que millora una mica cada dia.

Espero que siguis un d'ells.

Rasmus Bornhøft Schlünsen
Març 2026

Als meus fills –
sou la millor tripulacio que he tingut mai.
Tot aixo era per a vosaltres.
Sempre.