EN FELTHÅNDBOG

Det Agentiske Mandskab

Softwareudvikling i en tid med AI-agenter

Rasmus Bornhøft Schlünsen

Marts 2026

rev 2

Forord

Det var en tirsdag aften, engang sidste år. Mine børn sov, og jeg sad og stirrede på et migrationsscript, som jeg havde udskudt hele ugen — den slags kedelige, tabel-for-tabel omrokeringer, der æder en hel dag, hvis man er omhyggelig, og smadrer produktion, hvis man ikke er. På et indfald beskrev jeg problemet for en agent. Skemaet her, begrænsningerne der, pas på den fremmednøgle. Så trykkede jeg enter og gik ud for at lave te.

Da jeg kom tilbage, var scriptet færdigt. Ikke et udkast. *Færdigt.* Korrekte edge cases, rollback-logik, kommentarer jeg selv ville have skrevet. Jeg sad der længe, teen blev kold, og jeg følte to ting på én gang: ægte ærefrygt — og en stille, krybende angst.

For den migration? Det var *min* ting. Jeg var personen på holdet, der kunne holde hele skemaet i hovedet, som vidste hvilke joins der var forbandede, som kunne skrive den omhyggelige SQL i hånden. Femten års muskelhukommelse, og en LLM havde lige matchet det på fire minutter, mens jeg kogte vand.

Jeg vil være ærlig med dig: jeg sov ikke godt den nat. Jeg lå i sengen og kørte den samme løkke, som alle ingeniører jeg kender har kørt. *Hvad skal jeg så bruges til nu? Hvad sker der med det håndværk, jeg har brugt halvdelen af mit liv på at bygge op? Er jeg ved at træne min egen afløser?*

Det tog mig måneder — og en masse byggeri, fejlslag og genopbygning med de her værktøjer — at finde svaret. Og svaret overraskede mig. Håndværket er ikke ved at dø. Det *skifter ham.* Den ydre skal — tastetrykkene, syntaksen, boilerplaten — den del falder af. Men dyret indenunder? Den del, der ved *hvad* man skal bygge og *hvorfor*, der kan lugte en dårlig abstraktion tre filer væk, der kan holde et helt system i hovedet og mærke, hvor det er skrøbeligt? Den del er mere levende end nogensinde.

Vi bliver ikke erstattet. Vi bliver forfremmet. Fra maskinskrivere til tænkere. Fra at skrive kode til at dirigere den — orkestrere, reviewe, forme. De evner, der gjorde dig til en god ingeniør — systemtænkning, smag,

dømmekraft, instinktet for enkelhed — det er *hele spillet* nu, ikke bare baggrundsstøjen.

Men ingen gav os en manual til den her overgang. Den er rodet og ubehagelig og til tider ydmygende. Jeg skrev denne bog, fordi jeg lever midt i det, og jeg har en fornemmelse af, at du også gør. Disse sider er alt, hvad jeg har lært om at arbejde *med* agenterne i stedet for imod dem — eller endnu værre, lade som om de ikke eksisterer.

Hvis du nogensinde har set en AI skrive kode, der lignede din egen, og mærket maven synke, så er denne bog til dig. Bliv ved med at læse. Det bliver bedre — og mærkeligere — end du tror.

Rasmus Bornhøft Schlünsen
Marts 2026

Indhold

1 Introduktion

Sidste år så jeg en junior-udvikler på mit hold — to års erfaring, stadig nervøs til code reviews — shippe et komplet API-endpoint på femogfyrre minutter. Datamodel, validering, fejlhåndtering, tests, dokumentation. Koden var ren. Testene var grundige. PR'en gik igennem review i første forsøg.

Det ville have taget mig en time at lave det samme stykke arbejde. Og jeg har gjort det her i tyve år.

Hun tastede ikke det meste af det selv. Hun beskrev hvad hun havde brug for, pegede en agent mod codebasen og styrede den i mål. Hendes færdighed lå ikke i at skrive koden — den lå i at vide hvad hun skulle bede om, genkende hvornår outputtet var godt, og fange den ene edge case agenten missede. Hun *engineerede*. Bare ikke på den måde jeg lærte det.

Jeg tog hjem den aften og sad med et ubehageligt spørgsmål: hvis afstanden mellem tyve års erfaring og to års erfaring lige er blevet meget kortere, hvad er det så præcis jeg bidrager med?

Svaret, indså jeg til sidst, er alt det der ikke er at taste. Men det tog måneder at nå dertil, en masse fejltagelser, og denne bog.

1.1 Grunden Flytter Sig

I tyve år har det at være softwareingeniør betydet én ting: du åbner en editor, du skriver kode, du shipper den. Værktøjerne skiftede — fra Vim til VS Code, fra SVN til Git, fra bare metal til Kubernetes — men det grundlæggende loop forblev det samme. Dig, et tastatur og et problem.

Det loop er ved at gå i stykker. Og det går hurtigt.

AI-agenter autocompleter ikke bare din kode. De læser hele din codebase, ræsonnerer om arkitektur, laver ændringer på tværs af dusinvis af filer, kører dine tests og itererer på fejl — alt sammen uden at du rører tasta-

turet. De erstatter ikke editoren. De erstatter *workflowet*. Ingeniøren der plejede at bruge 80% af sin tid på at taste, bruger nu 80% af sin tid på at tænke, reviewe og styre.

Nogle ingeniører trives med dette skift. De shipper mere, med højere kvalitet, og de vil fortælle dig at de nyder deres arbejde mere end de har gjort i årevis. Andre er frustrerede, skeptiske eller stille og roligt skræmte over at det håndværk de brugte et årti på at mestre fordamper under dem.

Begge reaktioner er rationelle. Sandheden ligger et sted i den ubehagelige midte.

1.2 Hvorfor Denne Bog

Fordi ingen gav os en drejebog.

Værktøjerne kom hurtigt — Copilot, så Claude, så agenter der autonomt kan køre opgaver fra start til slut — og vi finder alle ud af det i realtid. Jeg ledte efter den bog der kunne fortælle mig hvordan man faktisk arbejder med de her ting. Ikke marketingpitchet. Ikke den akademiske artikel. Ikke Twitter-tråden fra nogen der prøvede det i tyve minutter. Jeg ville have den ærlige ingeniørguide — skrevet af nogen der shipper produktionskode og har set agenter gøre geniale ting og katastrofalt dumme ting i lige stor udstrækning.

Den bog fandtes ikke. Så jeg skrev den.

Jeg skrev den fordi jeg selv var fortabt. Jeg var den senior-ingeniør der ikke kunne finde ud af hvorfor agenten blev ved med at omskrive hele mit komponentbibliotek når jeg bad den om at fikse en knapfarve. Jeg var ham der brændte 500.000 tokens af på en opgave der burde have taget ti minutter, fordi jeg ikke vidste hvordan man sætter grænser. Jeg lavede alle fejlene i denne bog før jeg lærte at undgå dem.

Dette er den guide jeg ville ønske nogen havde givet mig.

1.3 Hvem Den Her Er Til

Du er softwareingeniør. Du har shippet rigtige ting. Du ved hvordan en produktionsincident føles klokken to om natten. Du er ikke bange for terminalen.

Men på det seneste føles noget anderledes. Måske har du prøvet AI-kodningsværktøjer og fundet dem imponerende men kaotiske — som at parprogrammere med nogen der er utroligt hurtig men ikke har nogen forståelse af scope. Måske har du set en udvikler med to års erfaring shippe en hel feature på en eftermiddag med agentassistance, og det fik dig til at føle noget du ikke havde forventet. Måske er du begejstret men ved ikke hvor du skal starte. Måske er du skeptisk og vil have nogen der overbeviser dig med substans, ikke hype.

Denne bog er til dig. Den forudsætter at du kan kode. Den forudsætter at du har været med et stykke tid. Den møder dig hvor du er.

1.4 Sådan Læser Du Denne Bog

Det her er ikke en referencemanual. Det er en rejse, og den er struktureret som en.

Vi starter med at forstå hvad der faktisk ændrer sig og hvorfor — skiftet i hvordan software bliver bygget, ikke bare værktøjerne men *tænkningen*. Derefter dykker vi ned i hvad agenter faktisk er, afklædt marketingsproget, så du har en mental model der holder når værktøjerne skifter næste kvartal.

Derfra får vi snavs på hænderne. Du lærer hvordan agenter fungerer i terminalen, hvordan man sætter guardrails op så de ikke ødelægger din codebase, hvordan Git-workflows ændrer sig når agenter committer kode, og hvorfor sandboxing ikke er valgfrit. Vi graver ned i testing som det feedback loop der gør agenter *pålidelige* i stedet for bare hurtige, og konventioner som det hemmelige våben de fleste overser.

Så går vi dybere — lokale versus kommercielle modeller, prompt engineering som en reel disciplin, og multi-agent orchestration. Vi ser på krigshistorier: rigtige fejl, rigtige lektioner, rigtigt arvæv. Vi snakker om

hvornår man *ikke* skal bruge agenter, for at vide hvornår man skal lægge værktøjet fra sig er lige så vigtigt som at vide hvordan man bruger det. Og vi slutter med hvordan teams adopterer det her uden at implodere.

Til sidst vil du ikke bare vide hvordan man bruger AI-agenter. Du vil vide hvordan man *tænker* om dem — hvilket er den færdighed der overlever når den nuværende generation af værktøjer er forældet.

1.5 Hvad Denne Bog Ikke Er

Lad mig spare dig noget tid.

Det her er ikke en prompt-kogebog. Du finder ikke "50 ChatGPT-prompts til udviklere" her. Prompting er vigtigt, og vi dækker det, men at copy-paste prompts uden at forstå systemet nedenunder er en opskrift på dyr frustration.

Det her er ikke et AI-hypemanifest. Jeg er ikke her for at fortælle dig at agenter erstatter alle programmører inden næste tirsdag. Det gør de ikke. Gabet mellem demo og produktion er lige så bredt som det altid har været, og nogen skal stadig passe på det gab.

Det her er heller ikke en undergangfortælling. "AI kommer efter dit job"-framingen er doven og for det meste forkert. Det der kommer er en fundamental ændring i *hvordan* jobbet fungerer, og det er en helt anden samtale.

Det her er en ingeniørbog. Til ingeniører. Skrevet af nogen der bruger sine dage på at skrive kode med agenter og har git-historikken til at bevise det. Vi bliver praktiske, ærlige og konkrete. Hvis det lyder som din slags ting, så vend siden.

2 Kontekst

Sidste måned kom der en bug fra en kunde: betalinger fejlede lydløst for brugere med ikke-ASCII-tegn i deres faktureringsadresse. En tricky en — den slags der lever i sømmen mellem din frontend-validering og din payment gateways tegnkodning.

En ingeniør på mit hold snuppede ticketten først. Han åbnede sin agent og tastede: "Der er en bug med betalinger for internationale brugere. Kan du kigge på det?" Agenten læste beredvilligt betalingsmodulet igennem, lavede nogle plausible gæt om Unicode-håndtering og producerede en patch der normaliserede al input til ASCII. Det ville have fjernet alle accenter, alle umlauts, alle tegn udenfor det engelske alfabet fra alle brugeres faktureringsadresser. Et fix der teknisk set var værre end buggen.

En time senere tog en anden ingeniør den samme ticket efter det første forsøg blev afvist i review. Hun indsatte kundens error log, det fejlende Sentry-trace, den specifikke payment gateway-svarkode, den relevante sektion af gatewayens tegnkodningsdokumentation og de tre filer involveret i faktureringspipen. Hendes agent diagnosticerede problemet på under to minutter: en UTF-8-streng blev sendt gennem en funktion der antog Latin-1-kodning før den ramte gatewayens API. Fixet var fire linjer. Det blev shippet den eftermiddag.

Modellen var den samme. Agenten var den samme. Buggen var den samme. Det der var forskelligt var hvad hver ingeniør lagde foran agenten før de bad den om at arbejde. Den ene gav den en vag beskrivelse og lod den gætte. Den anden gav den alt hvad den havde brug for til at *se* problemet.

Den forskel — hvad agenten kan se — er hvad dette kapitel handler om.

Den vigtigste færdighed i agentisk ingeniørarbejde er ikke prompting. Det er kontekststyring.

En AI-agent er kun så god som det den kan se. Giv den en vag instruktion og et blankt lærred, og den vil hallucinere selvsikkert. Giv den de rigtige

filer, de rigtige begrænsninger, det rigtige billede af systemet — og den vil gøre ting der føles som magi. Forskellen er ikke modellen. Det er dig.

Traditionelt ingeniørarbejde havde en version af dette også. En senioringeniør skrev ikke bare bedre kode — de holdt mere af systemet i hovedet. De vidste hvilke filer der var vigtige, hvor dragerne boede, hvilke abstraktioner der var bærende og hvilke der var dekorative. Den mentale model var konteksten, og den levede udelukkende i ingeniørens hjerne.

Nu skal du *eksternalisere* den. Dine agenter kan ikke læse dine tanker. De læser filer, environment variables, error logs og hvad end du lægger foran dem. Håndværket i agentisk ingeniørarbejde er at lære hvad man skal fremhæve, hvornår, og hvordan.

2.1 Kontekstvinduet Er Din Arbejdsbænk

Tænk på kontekstvinduet som en fysisk arbejdsbænk. Den har begrænset plads. Du kan ikke dumpe hele din codebase på den og forvente gode resultater. I stedet lægger du de dele ud der er vigtige for *denne* opgave: de relevante kildefiler, den fejlende test, skemaet, måske et stykke dokumentation.

En god agentisk ingeniør kuraterer kontekst ligesom en god kirurg lægger instrumenter frem. Intet unødvendigt. Alt inden for rækkevidde.

Det betyder at man udvikler instinkter for spørgsmål som:

- Hvad har agenten brug for at se for at forstå denne opgave?
- Hvad vil forvirre den hvis jeg inkluderer det?
- Er den kontekst jeg giver *aktuel*, eller fodrer jeg den med forældet information?

2.2 Kontekstvinduets Skat

Hver token du putter i et kontekstvindue koster dig to gange: én gang i penge, én gang i opmærksomhed.

Pengedelen er ligetil. API-kald prissættes efter tokenantal. Dump hele din codebase i konteksten og du brænder kroner af på hver interaktion. Men

den mere lumske omkostning er opmærksomhedsforringelse. Sprogmodeller behandler ikke alle tokens ens — der er et veldokumenteret fænomen hvor information midt i en lang kontekst får mindre vægt end information i starten eller slutningen. Jo mere du propper ind, jo mere sandsynligt er det at modellen overser det der faktisk er vigtigt.

Jeg lærte det på den hårde måde. Tidligt troede jeg mere kontekst altid var bedre. Under arbejde på en tricky database-migration fodrede jeg agenten med alle migrationsfiler vi nogensinde havde skrevet — tre års skemaændringer, hundredvis af filer. Min logik var sund: agenten havde brug for at forstå hele historikken for at skrive den næste migration korrekt. Resultatet var en migration der duplikerede en kolonne der allerede eksisterede, fordi den relevante tidligere migration var begravet midt i en enorm kontekst og modellen reelt tabte overblikket over den.

Næste forsøg gav jeg den kun det aktuelle skema, de tre seneste migrationer og et sammendrag på ét afsnit af den relevante historie. Agenten naglede det.

Det er kontekstvinduets skat i praksis. Der er et sweet spot mellem for lidt og for meget, og at finde det er en færdighed man udvikler gennem øvelse.

For lidt kontekst producerer hallucination. Agenten har ikke nok information, så den udfylder hullerne med plausibelt klingende opfindelser. Du beder den om at fikse en funktion uden at vise den filen, og den opfinder et API der ikke eksisterer. Du beder den om at skrive en test uden at vise den dit testframework, og den vælger Jest når du bruger Vitest.

For meget kontekst producerer forvirring og spild. Agenten har svaret begravet et sted i bunken, men den kan ikke finde det — eller værre, den finder modstridende information på tværs af forskellige filer og vælger den forkerte. Du betaler også for hver token af støj du inkluderede.

Sweet spottet er *kurateret* kontekst. Ikke alt hvad agenten muligvis kunne have brug for, men alt hvad den faktisk har brug for til denne specifikke opgave, lagt klart frem. Tænk på det mindre som at fylde et arkivskab og mere som at briefe en kollega før et møde. Du ville ikke give dem hvert eneste dokument virksomheden nogensinde har produceret. Du ville

give dem de tre ting de behøver at læse og et ét-minuts sammendrag af baggrunden.

En praktisk tommelfingerregel: hvis du er ved at paste noget ind i konteksten, spørg dig selv — vil agenten træffe en anderledes (bedre) beslutning fordi den så dette? Hvis svaret er nej, lad være med at inkludere det.

2.3 Lokalt, Sandboxet, Fjernt: Flyt Dine Agenter Rundt

Kontekst handler ikke kun om tekst i en prompt. Det handler om *hvor* din agent opererer og hvad den har adgang til.

2.3.1 Den Lokale Maskine

Det simpleste setup: agenten kører på din maskine, læser dine filer, udfører dine kommandoer. Det er her de fleste starter — værktøjer som Claude Code der opererer direkte i dit projektbibliotek.

Fordelen er umiddelbarhed. Agenten ser hvad du ser. Den kan læse din kode, køre dine tests, tjekke din git-historik. Risikoen er også åbenlys: det er din maskine, dine credentials, din produktionskonfiguration der ligger lige der i `~/.env`.

2.3.2 Sandboxede Miljøer

En mere disciplineret tilgang er at give agenten en sandbox — en container, en VM, et worktree. Den får en kopi af koden men ikke dine nøgler. Den kan ødelægge ting uden at ødelægge *dine* ting.

Det betyder mere end de fleste indser. Når du lader en agent iterere frit — køre kode, installere pakker, modificere filer — vil du have at den gør det i et rum hvor en fejl er billig. En sandboxet agent er en frygtløs agent, og en frygtløs agent er en produktiv en.

Worktrees er et undervurderet værktøj her. Git worktrees lader dig spinne en isoleret kopi af dit repo op på sekunder. Agenten arbejder i sin egen branch, sit eget bibliotek. Hvis resultatet er godt, merger du det. Hvis ikke, sletter du worktree'et og går videre. Intet rod.

2.3.3 Fjernudforskning

Her bliver det interessant. En dygtig agentisk ingeniør peger ikke bare agenter mod lokale filer — de lærer agenter at *udforske* fjernsystemer.

SSH ind på en staging-server for at undersøge logs. Forespørg en database for at forstå formen på rigtige data. Curl et API-endpoint for at se hvad det faktisk returnerer, ikke hvad dokumentationen påstår det returnerer. Hent container-logs fra en kørende service.

Agenten bliver din spejder. Du peger den mod et system og siger: "gå ud og kig dig omkring og fortæl mig hvad du finder." Men du er nødt til at sætte det op. Agenten har brug for credentials (scopede og midlertidige), netværksadgang og klare grænser for hvad den må røre ved.

Det er en vurderingssag — hvor meget adgang man skal give, til hvilke systemer, med hvilke guardrails. For lidt og agenten er ubrugelig. For meget og du er én dårlig prompt fra en produktionsincident. Den agentiske ingeniør lærer at kalibrere dette over tid.

2.4 Fodring af Kontekst Med Omtanke

De bedste agentiske ingeniører udvikler vaner omkring kontekst:

Start med fejlen. Beskriv ikke buggen — vis agenten stack tracet, den fejlende test-output, loglinjen. Rå kontekst slår omskrevet kontekst hver gang.

Vis, fortæl ikke. I stedet for at forklare dit databaseskema i prosa, giv agenten migrationsfilerne eller ORM-modellerne. I stedet for at beskrive API-kontrakten, giv den OpenAPI-specifikationen eller et curl-svar.

Beskær aggressivt. Hvis du debugger et renderingsproblem, behøver agenten ikke at se din authentication-middleware. Hver irrelevant fil i konteksten er støj der forringer signalet.

Brug filsystemet som kontekst. Et velorganiseret projekt *er* kontekst. Meningsfulde filnavne, klar mappestruktur, en god README — disse er ikke kun til mennesker længere. Dine agenter læser dem også.

Giv filstier, ikke skattejagter. Når du ved hvilke filer der er relevante, sig det eksplicit. "Buggen er i `src/payments/gateway.ts`, specifikt

`encodeAddress`-funktionen på linje 142" er uendeligt bedre end "der er en bug et sted i betalingskoden." Hvert minut agenten bruger på at lede efter den rigtige fil er et minut den ikke bruger på det faktiske problem — og den brænder tokens hele tiden.

Brug git blame til at forklare *hvorfor*. Kode fortæller agenten *hvad* der eksisterer. Git-historik fortæller den *hvorfor*. Når du beder en agent om at modificere et stykke kode der har et ikke-oplagt design, peg den mod den relevante commit-besked eller pull request. "Denne funktion ser mærkelig ud men den blev skrevet sådan på grund af 1247 — se commit-beskeden ved `abc123`" giver agenten den begrundelse den har brug for til at lave ændringer uden at bryde den originale intention.

Copy-paste over omskrivning. Det er værd at gentage fordi det er den mest almindelige fejl jeg ser. Ingeniører beskriver en fejl med deres egne ord i stedet for at paste den faktiske fejl ind. "Buildet fejler med en TypeScript-fejl om typer" versus at paste det præcise compiler-output med filsti, linjenummer og fejlkode. Det første giver agenten en vag retning. Det andet giver den et specifikt mål. Paste altid det rå output. Lad agenten stå for fortolkningen.

Lag din kontekst. Til komplekse opgaver, dump ikke alt på én gang. Start med det overordnede billede — hvad systemet gør, hvad du forsøger at ændre, hvorfor. Giv derefter de specifikke filer. Giv derefter fejlen eller testfejlen. Det spejler hvordan du ville briefe en menneskelig kollega, og det virker af samme grund: det opbygger en mental model før man dykker ned i detaljer.

2.5 Kontekst På Tværs Af Sessioner

Her er et problem ingen advarer dig om: kontekstvinduer er flygtige. Når en session slutter, forsvinder alt agenten lærte — hver fil den læste, hver beslutning den tog, hver blindgyde den udforskede. Næste session starter fra nul.

Til korte opgaver er det ligegyldigt. Du paster fejlen ind, agenten fikser den, færdig. Men rigtigt ingeniørarbejde strækker sig over dage, nogle gange uger. En feature der rører tolv filer på tværs af tre services. En

refaktorering der skal ske i etaper. En debugging-session hvor de første to timer indsnævrer problemet og den sidste halve time fikser det — bortset fra at du var nødt til at lukke din laptop ind imellem.

Hvis du ikke planlægger for sessionsgrænser, vil du spilde enorme mængder tid på at genetablere kontekst agenten allerede havde. Jeg har set ingeniører bruge de første femten minutter af hver session på at genforklare hvad de fortalte agenten i går. Det er ikke ingeniørarbejde. Det er babysitting.

Løsningen er at gøre kontekst *holdbar* — at gemme den et sted hvor næste session kan samle den op.

Lad codebasen være kontinuitetslaget. Den mest pålidelige form for kontekst på tværs af sessioner er koden selv. Hvis agenten lavede fremskridt i går, bør det fremskridt være committet. Gode commit-beskeder bliver brødkrummestien: "Refaktorerede payment gateway til at separere kodningsstepper — næste skridt er at tilføje tests for ikke-ASCII-input." Næste session starter med at læse den seneste git log, og agenten har et klart billede af hvor tingene står.

Brug CLAUDE.md-filer (eller tilsvarende). Mange agentværktøjer understøtter en kontekstfil på projektniveau — en markdown-fil i roden af dit repo der beskriver projektets arkitektur, konventioner og aktuelle tilstand. Denne fil persisterer på tværs af sessioner fordi den lever i filsystemet. Opdater den efterhånden som projektet udvikler sig. Inkluder ting som: hvad de vigtigste komponenter er, hvilke mønstre man skal følge, hvad der er i stykker lige nu, hvad holdet arbejder på. Det er et briefingdokument som hver ny session automatisk læser.

Skriv sessionsresumeer. Når du afslutter en kompleks session, brug tres sekunder på at lade agenten opsummere hvad den opnåede, hvad der mangler, og hvad den lærte om codebasen. Gem det resumé et sted — en kommentar på ticketten, en note i projektet, selv en tekstfil i repoet. Næste session starter med at læse det resumé, og du har bevaret timers akkumuleret forståelse i nogle få afsnit.

Commit tidligt og ofte. Det er dækket i Git-kapitlet, men det er værd at gentage her fordi det fundamentalt er en kontekststyringsstrategi. Hvert commit er et checkpoint som fremtidige sessioner kan referere til. En

session der slutter med ucommittede ændringer er en session hvis kontekst er fanget i et terminalvindue der måske ikke eksisterer i morgen.

De ingeniører der håndterer langvarigt agentisk arbejde godt er dem der behandler sessionsgrænser som en førsteklasses bekymring. De lukker ikke bare laptopen — de lukker loopet.

2.6 Kontekst Som Arkitektur

Her er den dybere indsigt: efterhånden som du bliver bedre til agentisk ingeniørarbejde, begynder du at designe dine systemer *til* kontekst. Du skriver klarere commit-beskeder fordi agenter læser dem. Du holder funktioner små fordi agenter arbejder bedre med fokuserede filer. Du vedligeholder opdateret dokumentation fordi agenter behandler det som sandhed.

Det her er ikke en mindre justering. Det ændrer hvordan du tænker om kodestruktur på alle niveauer.

Små funktioner er kontekstvenlige. En funktion på 400 linjer kræver at agenten holder hele molevitten i arbejdshukommelsen for at lave en ændring sikkert. En funktion på 30 linjer der gør én ting er noget agenten kan forstå fuldstændigt, modificere med selvtillid og verificere hurtigt. Det gamle råd — "en funktion bør gøre én ting" — var altid god ingeniørkunst. Nu er det også god kontekststyring. Hver gang du udtrækker en velnævnt funktion fra en større, skaber du en meningsenhed som en agent kan arbejde med uafhængigt.

Filnavngivning er navigation. Når en agent skal finde koden der håndterer brugerautentificering, starter den med at kigge på filnavne. `auth.ts` er et signal. `utils.ts` er et sort hul. `handleStuff.js` er en blindgyde. Disciplinen med at navngive filer klart — `user-authentication.ts`, `payment-gateway.ts`, `rate-limiter.middleware.ts` — er ikke længere bare en høflighed overfor fremtidige udviklere. Det er et indeks som agenter bruger til at finde vej gennem din codebase uden at læse hver eneste fil.

Mappestruktur er arkitekturdokumentation. En flad mappe med tres filer fortæller agenten ingenting om hvordan systemet er organiseret. Et klart hierarki — `src/api/`, `src/services/`, `src/models/`, `src/`

`middleware/` — fortæller den alt. Agenten kan udlede arkitekturen fra mappestrukturen alene, uden at læse en eneste linje dokumentation. Jeg har set agenter navigere en velstruktureret monorepo med 10.000 filer mere effektivt end et dårligt struktureret projekt med 200, udelukkende fordi mappelayoutet gjorde systemet læsbart.

Monorepos vs. multirepos: en kontekst-afvejning. I en monorepo kan agenten se alt — API'et, frontenden, de delte biblioteker, infrastrukturkonfigurationen. Det er kraftfuldt til opgaver der krydser grænser. Men det betyder også at agenten kan se *for meget*, og trækker irrelevant kode ind fra urelaterede services. I et multirepo-setup er hvert repo naturligt scopet — agenten ser kun den service den arbejder på. Men opgaver på tværs af services bliver sværere fordi agenten ikke nemt kan referere til den anden side af en API-kontrakt. Ingen af tilgangene er universelt bedre. Pointen er at din repo-strategi er en kontekstbeslutning, uanset om du tænker på det sådan eller ej.

Typer er kontekst. En stærkt typet codebase giver en agent noget som en dynamisk typet ikke gør: en maskinlæsbar beskrivelse af hver funktions kontrakt. Agenten kan se på en funktionssignatur og vide præcis hvad der går ind og hvad der kommer ud, uden at læse implementeringen. TypeScript, Rust, Go — disse sprog bærer strukturel kontekst i deres typesystemer. Python og JavaScript lader agenten gætte medmindre du har skrevet grundige docstrings eller type hints. Det her er ikke et argument for ét sprog frem for et andet. Det er en observation om at typesystemer gør dobbelt tjeneste i den agentiske æra: de fanger bugs *og* de kommunikerer intention.

Dokumentation er sandhed (uanset om den er korrekt eller ej). Agenter behandler din README, dine API-docs, dine inline-kommentarer som autoritative. Hvis din dokumentation siger at API'et returnerer et `user_id`-felt men det faktiske svar returnerer `userId`, vil agenten skrive kode mod dokumentationen og producere en bug. Forældet dokumentation var altid et irritationsmoment. Med agenter er det en aktiv kilde til defekter. Standarden for dokumentationsnøjagtighed går op — ikke fordi agenter har brug for bedre docs end mennesker, men fordi agenter vil følge dårlige docs mere trofast end et menneske ville.

Den måde du strukturerer din kode, dine repos, din infrastruktur — det hele bliver en del af den kontekst du giver dit mandskab. De ingeniører der forstår dette tidligt vil bygge systemer der ikke bare er vedligeholdelige af mennesker, men *navigerbare* af agenter. Og over tid akkumulerer denne navigerbarhed — hver ny agentsession drager fordel af hver strukturel beslutning du tog før den.

3 Hvad Er en Agent?

Ordet "agent" bliver kastet rundt i flæng. Det bruges om alt fra en chatbot der svarer på spørgsmål til et system der autonomt deployer kode til produktion. Før vi går videre, lad os være præcise om hvad vi mener — for distinktionen er vigtig for hvordan du arbejder med dem.

3.1 Spektrummet

Ikke alle AI-værktøjer er agenter. I den ene ende foreslår **autocomplete** de næste par tokens mens du taster — reaktivt, én linje ad gangen, ingen tænkning involveret. **En copilot** ser mere kontekst og genererer større blokke, men den er stadig passiv: du spørger, den svarer. Skiftet sker med **værktøjsbrugende agenter**. En agent genererer ikke bare tekst — den *handler*. Den læser filer, skriver filer, kører kommandoer, inspicerer resultater, og gør det afgørende i et loop: prøv, observer, justér, prøv igen. I den anden ende tager **autonome agenter** et overordnet mål, planlægger deres egen tilgang og leverer et resultat med minimal menneskelig interaktion.

Det meste praktiske agentiske ingeniørarbejde i dag foregår i den værktøjsbrugende zone. Du giver agenten en opgave, den har adgang til værktøjer, og den arbejder iterativt. Du er i loopet — reviewende, vejledende, godkendende — men agenten laver det tunge løft.

3.2 Hvad Gør Noget "Agentisk"

Tre evner adskiller en agent fra en fancy chatbot:

Planlægning. En agent bryder et mål ned i trin. "Tilføj autentificering til denne app" bliver til en række handlinger — læs codebasen, vælg frameworket, opret middleware, opdater routes, tilføj tests, verificer. En chatbot giver dig en kodeblok. En agent giver dig en proces.

Værktøjsbrug. En agent interagerer med verden — læser dine filer, kører dine tests, undersøger fejloutput. Hvert tool call giver ny information der former den næste beslutning. Dette feedback loop er det der gør agenter kraftfulde: de genererer ikke kode i et vakuum, de genererer kode og *verificerer* den.

Iteration. En agent kan prøve, fejle og prøve igen. Skriv en funktion, kør testene, se en fejl, læs fejlen, justér, kør igen. Handl, observer, justér. En chatbot giver dig ét skud. En agent giver dig en cyklus.

3.3 Agenter Er Ikke Magi

Det er vigtigt at være klar i øjnene om hvad agenter er og hvad de ikke er.

Agenter er ikke bevidste. De forstår ikke din kode på den måde du gør. De har ikke intuition, smag eller erfaring. Hvad de har er evnen til at processere store mængder tekst, genkende mønstre og generere plausible næste skridt — meget hurtigt, meget utrætteligt, og i et omfang der ville udtømme ethvert menneske.

De hallucinerer. De laver selvsikre fejl. De løser nogle gange det forkerte problem på smuk vis. De kan skrive kode der passerer alle tests men fuldstændig misser pointen. De er geniale praktikanter med uendelig energi og ingen dømmekraft.

Det er derfor *ingeniøren* er vigtig. Agenten bidrager med hastighed og bredde. Du bidrager med retning, dømmekraft og smag. Kombinationen er mere kraftfuld end nogen af delene alene.

3.4 Når Agenter Fejler

De vil fejle. At forstå *hvordan* de fejler hjælper dig med at bygge bedre workflows.

Scope creep. Du beder om et bugfix, agenten refaktorerer tre filer og opdaterer build-systemet. Agenter er ivrige, og den iver strækker sig ud over hvad du bad om. Små, fokuserede opgaver og branch-isolering er dit forsvar.

Hallucinerede API'er. Agenten kalder funktioner eller biblioteker der ikke eksisterer — eller eksisterer i en anden version. At køre tests fanger dette. Agenten kan ikke hallucinere sig forbi en testsuite.

Overselvsikkerhed. Agenten siger den er færdig, og det ser færdigt ud, men der er en subtil bug der kun viser sig under specifikke betingelser. Review diffs. Stol ikke blindt på agentens output.

Kontekstab. På lange opgaver mister agenten overblikket over tidligere beslutninger — modsiger sig selv, omskriver kode den allerede skrev, glemmer begrænsninger. Små commits og klar kontekststyring er modgiften.

Hver fejltilstand har en modforanstaltning, og de modforanstaltninger er kapitlerne i denne bog: kontekst, guardrails, git, sandboxes, testing, konventioner. Principperne er ikke teoretiske — de er direkte svar på hvordan agenter fejler i praksis.

3.5 Den Rigtige Mentale Model

Tænk ikke på agenter som værktøjer. Tænk ikke på dem som erstatninger. Tænk på dem som samarbejdspartnere med et meget specifikt sæt styrker og svagheder.

De er hurtige hvor du er langsom. De er tålmodige hvor du er utålmodig. De kan holde mere tekst i arbejdshukommelsen end du kan. De bliver aldrig trætte, aldrig frustrerede, har aldrig en dårlig dag.

Men de ved ikke hvad der er vigtigt. De ved ikke hvad brugeren faktisk har brug for. De ved ikke hvilken teknisk gæld der er acceptabel og hvilken der er en tikkende bombe. De ved ikke hvornår man skal skubbe tilbage på et krav. De ved ikke hvornår specifikationen er forkert.

Det er dit job. Og det vil det altid være.

4 Hvad Jeg Lærte Af At Bygge Mit Eget Agentværktøj

Her er hvad ingen fortæller dig når du begynder at arbejde med AI-agenter: det svære er ikke at få én agent til at gøre noget nyttigt. Det svære er at håndtere kaosset når du har tre af dem kørende på én gang. Fire terminalvinduer åbne, agenter der arbejder på forskellige opgaver, én der crashede lydløst for tyve minutter siden og du har ikke opdaget det. Agenterne har det fint. *Du* er flaskehalsen.

Jeg brugte hundrede timer på at bygge mit eget værktøj til at løse dette — Clovr Code Terminal, en browserbaseret kontrolflade til at køre multiple agentsessioner. De timer lærte mig mere om agentisk ingeniørarbejde end nogen mængde læsning kunne have gjort. Dette kapitel destillerer de principper der kom ud af det. CCT er casestudiet, ikke pointen.

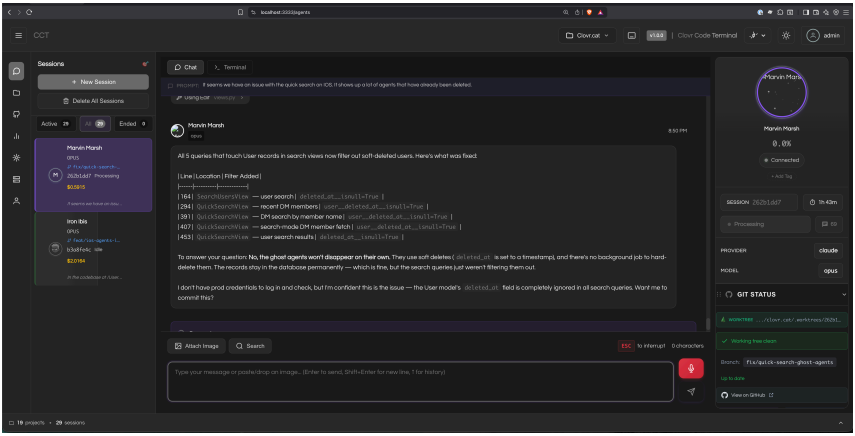
4.1 Agentisk Arbejde Er Iboende Parallelt

Single-agent workflows er en krykke. Ikke altid — nogle gange er én agent på én opgave præcis det rigtige. Men den virkelige styrke ved agentisk ingeniørarbejde viser sig når du kører multiple agenter parallelt, hver fokuseret på en forskellig del af problemet.

Det er kontraintuitivt. De fleste af os er vokset op i en verden hvor *vi* er den enkelte udførelsestråd. Skriv kode, så tests, så docs. Sekventielt. Agenter bryder den model — de kan alle arbejde samtidigt, men kun hvis du kan holde styr på dem.

Hvis dit workflow ikke giver dig synlighed over parallelt arbejde, vil du enten serialisere alt (og spilde agenternes potentiale) eller køre ting parallelt og miste overblikket (og spilde din egen tid på at rydde op i rodet). Uanset hvilket værktøj du bruger — tmux-paneler, multiple

Cursor-projekter, selv en post-it der tracker hvad der kører hvor — løs synlighedsproblemet først. Agenterne styrer ikke sig selv.



Figur 1: CCT's hoveddashboard — multiple agentsessioner kørende parallelt, med realtidsstatus og omkostningssporing.

4.2 Agenter Har Brug For Struktureret Kontekstoverlevering

Du har en agent der lige er færdig med at planlægge en feature. Den kender kravene, filstrukturen, edge cases. Nu vil du have en anden agent til at implementere det der blev planlagt. Hvordan overfører du den viden?

Den naive tilgang er copy-paste — grib planen fra agent éts output, paste den ind i agent tos prompt. Det virker, lige akkurat. Du mister nuancer, glemmer ting og bliver et menneskeligt clipboard, hvilket er præcis den slags lavværdiarbejde agenter skal eliminere.

Den bedre tilgang er strukturerede overleveringer: et defineret format til at sende kontekst fra én agent til en anden. Skriv en overleveringsskabelon. Få agenter til at opsummere deres arbejde i et konsistent format før de afslutter. Fodér det resumé til den næste agent. Det er "mandskab"-metaforen gjort operationel — agenter der samarbejder ikke gennem delt hukommelse, men gennem struktureret kommunikation.

Jeg har brugt dette mønster til at kæde tre agenter i sekvens: én til at planlægge, én til at designe, én til at implementere. Hver læser den forrige agents overlevering. Resultatet er konsekvent bedre end en enkelt agent der forsøger at gøre alt, fordi hver agent arbejder inden for en fokuseret kontekst i stedet for en udflydende en.

4.3 Tillid Skal Være Konfigurerbar

Hvert tool call en agent laver — hver bash-kommando, hver filskrivning, hver netværksforespørgsel — er en tillidsbeslutning. At køre agenter uden guardrails er hurtigt og skræmmende. Én af mine kørte `rm -rf` på et bibliotek jeg brød mig om. (Det var i et worktree, så ingen reel skade. Lektien lærte jeg alligevel.) Den modsatte ekstrem — at godkende hver eneste operation manuelt — gør agenter ubrugelige. Du bruger al din tid på at klikke "tillad" på `git status` og `ls`.

Det rigtige svar er et konfigurerbart tillidsspektrum. Always-allow-regler for sikre kommandoer, manuel godkendelse for sensitive operationer og fuld-auto-tilstand til hurtig prototyping i disposable miljøer. Over tid bliver din tilladelseskonfiguration et levende dokument over dit tillidsforhold med agenterne.

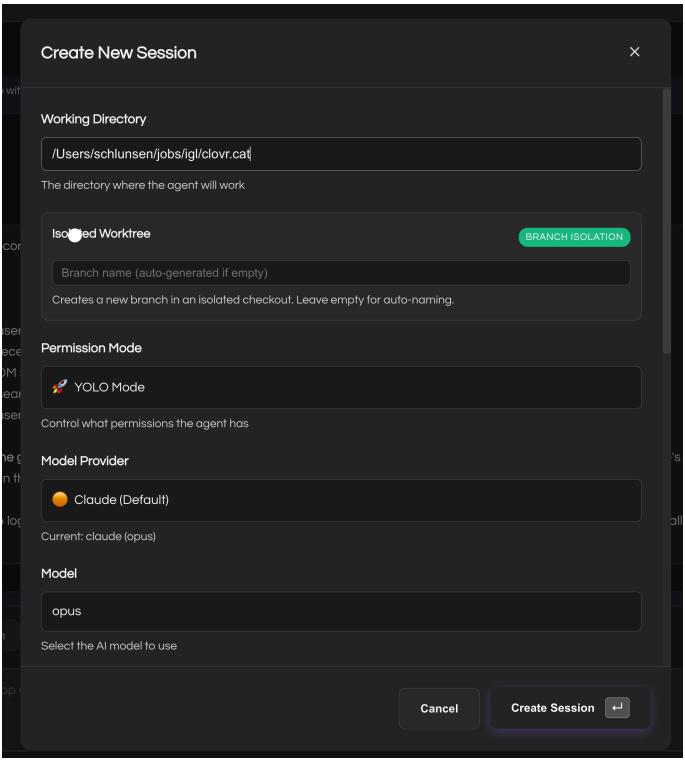
Ethvert agentisk workflow har brug for en måde at tune tillid på. Hvis dit værktøj kun tilbyder "tillad alt" eller "godkend alt," vil du svinge mellem angst og frustration. Tilladelseslaget er ikke overhead — det er det der gør vedvarende agentisk arbejde muligt.

4.4 Versionskontrol Er Agentinfrastruktur

Hver gang en agent lavede rod, var mit første instinkt `git diff` og `git stash`. Versionskontrol var allerede mit sikkerhedsnet. At gøre det til en førsteklasses del af værktøjet formaliserede bare det jeg allerede gjorde manuelt.

Princippet er simpelt: *lad aldrig en agent arbejde på din main-branch.* Giv den en branch. Giv den et worktree. Giv den en container. Uanset hvilken isoleringsmekanisme du foretrækker, brug den. Gode resultater

bliver merged. Dårlige resultater bliver slettet. Intet rod, ingen risiko, intet drama.



Figur 2: Oprettelse af en ny session med worktree-isolering, tilladelsesmode og modelvalg — hvert princip fra denne bog bagt ind i en enkelt dialog.

Hvis du bruger Claude Code i en terminal, giver et simpelt shell-script der opretter et worktree og starter en session dig firs procent af denne fordel. Værktøjet er ligegyldigt. Isoleringen er det der tæller.

4.5 Du Behøver Ikke At Bygge Dit Eget

Jeg vil være direkte: *byg ikke din egen agentiske IDE.* Jeg gjorde det fordi jeg havde specifikke kløer der skulle klås og fordi det at bygge det lærte

mig ting jeg ville skrive om. Men jeg kunne have været produktiv med Claude Code i en terminal og et par gode shell-scripts.

Principperne i dette kapitel — parallel synlighed, strukturerede overleveringer, konfigurerbar tillid, versionskontrol som infrastruktur — kan implementeres med ethvert værktøj:

- **Parallel synlighed:** tmux-paneler, en simpel logfil, selv en post-it der tracker hvad der kører hvor.
- **Strukturerede overleveringer:** en markdown-skabelon som agenter udfylder når de er færdige. Kopiér den til den næste agents prompt.
- **Konfigurerbar tillid:** Claude Codes tilladelsesflags, `.claude/settings.json`, eller bare at køre agenter i en begrænset sandbox.
- **Git-isolering:** et tre-linjers shell-script der opretter et worktree og starter en session.

Værktøjet er ligegyldigt. Den mentale model er det der tæller.

4.6 Den Egentlige Lektie

Agentisk ingeniørarbejde handler ikke rigtig om agenterne. Det handler om *systemerne omkring agenterne* — synligheden, kontekstoverleveringen, tillidsgrænser, isoleringen, feedback loops. Få dem rigtigt og næsten enhver kapabel model vil producere gode resultater. Få dem forkert og selv den bedste model vil skabe dyrt kaos.

Resten af denne bog handler om de systemer.

5 Guardrails

At give en agent magt uden grænser er ikke modigt ingeniørarbejde — det er forsømmelighed. Guardrails er det der gør autonome agenter *brugbare* i rigtigt arbejde. De er forskellen mellem en agent der hjælper dig med at shippe og en der tager dit staging-miljø ned klokken tre om natten.

Og alligevel er guardrails ikke et løst problem. De er noget levende — noget du tuner, tester og diskuterer. Få dem forkert i den ene retning og din agent er farlig. Få dem forkert i den anden retning og din agent er ubrugelig. Håndværket ligger i at finde grænsen.

5.1 Tillidsgradient

Ikke hver opgave fortjener det samme niveau af autonomi. At læse filer? Lav risiko, lad den køre. At skrive kode i en feature branch? Medium risiko, tjek diffen. At køre databasemigrationer? Høj risiko, kræv eksplicit godkendelse.

Tænk på det som en mixerpult. Hver kategori af handling har sin egen skyder: fillæsning, filskrivning, shell-kommandoer, netværksadgang, git-operationer. Du skubber hver skyder til det niveau af autonomi du er komfortabel med. Nogle skydere forbliver lave for altid. Andre skubber du op efterhånden som tilliden vokser.

Fejlen de fleste laver er at behandle gradienten som binær — enten kan agenten gøre noget eller den kan ikke. Virkeligheden er mere nuanceret. En agent kan have lov til at køre `npm test` uden at spørge, men `npm install` kræver en prompt. Begge er shell-kommandoer. Risikoprofilen er helt anderledes.

Og gradienten skifter over tid. På dag ét kører din agent i en stram sandbox. Hver shell-kommando godkendes. Hver filskrivning reviewes. Du ved endnu ikke hvor den er genial og hvor den er skrøbelig, så du overvåger alt.

Efter en uge viser mønstre sig. Agenten er fejlfri til at skrive unit tests. Den er solid til refaktorering. Den laver af og til tvivlsomme valg om afhængighedshåndtering. Nu afspejler dine skydere det: tests og refaktorering kører frit, afhængighedsændringer reviewes.

Efter en måned har du set hundrede opgaver gennemført med succes. Du løsner skyderne yderligere. Agenten committer til feature branches på egen hånd. Den kører den fulde testsuite uden at spørge. Efter tre måneder er det som en betroet kollega — du giver den en opgave, tjekker tilbage efter en time og reviewer PR'en. Guardrails er der stadig, men de er usynlige for de 95% af arbejdet der er rutine.

Det er den centrale indsigt: *guardrails bør være knap mærkbare for hverdagsarbejde, og absolut rigide for ekstraordinære situationer.* Agenten bør flyde gennem sine normale opgaver uden friktion, og ramme en hård mur i det øjeblik den forsøger noget usædvanligt. Den mur er der hvor du dukker op.

De ingeniører der aldrig justerer skyderne ender med at opgive agentiske workflows helt. De ingeniører der skubber dem for hurtigt brænder sig. Sweet spottet er en stabil, evidensbaseret stramning.

5.2 Tilladelsesscoping

Princippet er det samme som i sikkerhed: mindste privilegium. En agent der arbejder på din frontend har ikke brug for SSH-adgang til din databaseserver. En agent der skriver unit tests har ikke brug for dine AWS-credentials.

I praksis betyder det:

- Scopede API-nøgler med udløbstider
- Miljøspecifikke credentials (del aldrig prod-nøgler med en dev-agent)
- Læseadgang som standard, skriveadgang som undtagelse
- Netværksisolering hvor muligt — hvis agenten ikke har brug for internet, giv den ikke internet

Værktøjer som Claude Code har allerede indbyggede tilladelsessystemer — allow/deny-lister for kommandoer, filadgangskontroller, godkendelses-

prompts for destruktive operationer. Brug dem. Godkend ikke blindt alt fordi det er hurtigere at klikke "ja."

En konkret allowlist kan starte sådan her:

```
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
```

Det er en dag-ét-konfiguration. Den er konservativ. Agenten kan læse, teste og udforske — men den kan ikke slette, deploye eller nå netværket. Du vil mærke friktionen med det samme. Agenten vil bede om tilladelse til at køre `npm install` når den har brug for en afhængighed. Den vil spørge før den opretter en fil. Det er meningen.

Efter en uge har du set den arbejde. Du stoler på dens vurdering af filoprettelse. Du tilføjer `touch` og `mkdir` til allowlisten. Efter en måned lader du den køre `npm install` uden at spørge — men kun i projektbiblioteket, ikke globalt. Efter tre måneder lader du den pushe til feature branches men aldrig til `main`.

Allowlisten *vokser* med din erfaring. Den er en log af tillidsbeslutninger, og at læse en ingeniørs allowlist fortæller dig præcis hvor meget agentisk erfaring de har.

5.3 Godkendelsesporte

Nogle handlinger bør altid kræve et menneske i loopet. Ikke fordi agenten ikke kan gøre dem, men fordi *konsekvenserne* af at få dem forkert er for høje.

Gode kandidater til godkendelsesporte:

- Enhver operation der rører produktionsdata
- Sletning af filer eller branches
- Installation af nye afhængigheder
- Netværksforespørgsler til eksterne services
- Ethvert git push
- Ændring af CI/CD-konfiguration
- Ændring af environment variables eller secrets

Målet er ikke at bremse agenten. Målet er at skabe naturlige checkpoints hvor du, ingeniøren, kan verificere at agentens kurs stadig matcher din intention. Et hurtigt blik på en diff tager fem sekunder. At komme sig efter en dårlig deploy tager timer.

Den bedste port er en du næsten aldrig afviser. Hvis du konstant afviser godkendelser, er din agent forkonfigureret eller kommunikerer dårligt — og du bør fikse årsagen, ikke blive ved med at klikke "nej."

5.4 Når Guardrails Fejler

Det gør de. En agent vil misforstå en begrænsning, finde en kreativ omvej til en limitation, eller støde på en edge case dine guardrails ikke forventede. Det er normalt.

Her er et virkeligt scenarie. En ingeniør havde `rm` og `rm -rf` på deny-listen — fornuftigt nok. Agenten havde brug for at fortryde nogle ændringer til et sæt filer. Den kunne ikke slette dem. Så den kørte `git checkout -- .` som *var* på allowlisten, fordi at checke filer ud fra git lyder uskadeligt. Resultatet? Alle ucommittede ændringer i arbejdsbiblioteket — inklusive ingeniørens eget igangværende arbejde på andre filer — blev slettet. Agenten løste sit snævre problem og skabte et meget større.

Lektien er ikke at `git checkout` bør nægtes. Det er at guardrails er *forsvar i dybden*, ikke en enkelt mur. Du har brug for multiple lag:

- **Allowlisten** fanger de åbenlyse farlige kommandoer.
- **Sandboxen** (et worktree, en container) begrænser eksplosionsradius.
- **Commit-historikken** lader dig genskabe når noget slipper igennem.
- **Din egen review** fanger de ting som ingen automatiseret regel ville flagge.

Intet enkelt lag er tilstrækkeligt. En agent der er blokeret fra at køre `rm` vil finde en anden måde at slette data på hvis det er hvad den mener opgaven kræver. Den er ikke ondsindet — den er *opfindsom*. Den samme kreativitet der gør agenter nyttige er det der gør enkeltlags-guardrails utilstrækkelige.

Svaret er ikke at fjerne autonomi — det er at forbedre guardrails og tilføje lag. Hver fejl er et signal. Behandl det som en bug: forstå hvad der skete, tilføj et tjek og gå videre. Over tid bliver din guardrail-konfiguration en afspejling af hårdt tilkæmpet erfaring, ikke ulig hvordan en `.gitignore` vokser med et projekt.

De bedste agentiske ingeniører frygter ikke agentfejl. De bygger systemer hvor fejl fanges tidligt, inddæmmes hurtigt og læres af automatisk.

5.5 Omkostningen Ved For Mange Guardrails

Der er en fejltilstand der ligner forsigtighed men ikke er det. Du konfigurerer din agent med godkendelsesporte på alt — hver filskrivning, hver shell-kommando, hver git-operation. Femten minutter inde i en opgave har du klikket "ja" fyrre gange og du læser dem ikke længere.

Det er faren. Overbegrænsede agenter producerer to resultater, begge dårlige. Enten giver ingeniøren op og stopper med at bruge agenter, og konkluderer at de "ikke er klar endnu." Eller — værre — godkendelsestrætheden træner dem til at klikke "ja" refleksivt. Nu har du guardrails der *føles* sikre men giver nul reel beskyttelse.

Færdigheden ligger i at finde sweet spottet. Du vil have guardrails stramme nok til at fange ægte fejl, og løse nok til at agenten kan *flyde*.

En god tommelfingerregel: hvis du godkender den samme type handling mere end fem gange i en session, bør den nok være på allowlisten.

En anden tommelfingerregel: track hvor ofte dine godkendelser faktisk afviser noget. Hvis du har godkendt fem hundrede handlinger og afvist tre, er dine guardrails for aggressive for de handlingstyper. Hvis du har godkendt halvtreds og afvist ti, er de velkalibrerede — de ti afvisninger er dem der tæller.

Målet er en agent der arbejder som en dygtig entreprenør. Den dukker op, gør arbejdet og tjekker ind med dig ved meningsfulde milepæle. Ikke efter hvert hammerslag.

5.6 Miljøspecifikke Guardrails

Din udviklingslaptop, din staging-server og dit produktionsmiljø er tre helt forskellige risikoprofiler. Dine guardrails bør afspejle det.

Lokal udvikling er der hvor du giver agenter mest frihed. Agenten kan installere pakker, køre vilkårlige kommandoer, modificere enhver fil og udføre tests — fordi worst case er at du kører `git reset` og starter forfra. Din lokale maskine er en legeplads. Lad agenten lege.

Selv her er der grænser. Agenten bør ikke have adgang til produktionscredentials. Den bør ikke kunne pushe til `main`. Den bør ikke kunne publicere pakker. Men inden for rammerne af "lokal udvikling på en feature branch," lad den køre hurtigt.

Staging strammer skruerne. Agenten kan deploye til staging — men med godkendelse. Den kan læse staging-logs og forespørge staging-databaser — men ikke modificere data. Den kan køre integrationstests mod staging-services — men ikke rekonfigurere de services. Staging er hvor du verificerer at agentens arbejde overlever kontakten med et rigtigt miljø, og guardrails afspejler de højere indsatser.

Produktion er et helt andet dyr. Det mest ærlige råd: din produktionsagent bør være read-only, hvis den overhovedet eksisterer. Lad den forespørge logs. Lad den læse metrikker. Lad den undersøge incidents ved at hente data. Men i det øjeblik den skal *ændre* noget i produktion — en

konfigurationsværdi, en databaserecord, en kørende service — er det en menneskelig beslutning, punktum.

Nogle teams tillader agenter at eksekvere forhåndsgodkendte runbooks i produktion: genstart en service, skaler replicas op, rul tilbage til en known-good deploy. Det er stramt scopede, veltestede operationer med klare rollback-veje. Det er rimeligt. Men "lad agenten finde ud af hvordan man fikser produktionsincidentet" er ikke en guardrail-konfiguration — det er en bøn.

Mønstret er simpelt: jo tættere du er på rigtige brugere og rigtige data, jo strammere bliver guardrails. Din lokale agent er en samarbejdspartner. Din staging-agent er en superviseret arbejder. Din produktionsagent er en read-only observatør.

Sæt det op én gang, og det bliver usynligt. Agenten tilpasser sin adfærd baseret på det miljø den opererer i. Den kører hurtigt lokalt, tjekker ind på staging og går hands-off i produktion. Når dit hold er enige om disse grænser, behøver de sjældent at blive genbesøgt — og når de gør, er det fordi noget gik galt i produktion, hvilket er præcis hvornår du vil gentænke guardrails alligevel.

6 Git Som Agentinfrastruktur

Du kender allerede Git. Du har committet, branchet og merged i årevis. Men i agentisk ingeniørarbejde er Git ikke bare versionskontrol — det er rygraden i hele dit workflow. Det er din fortrydelsesknap, dit parallelle eksekverings-framework, din review-grænseflade og dit dokumentationssystem på én gang.

De fleste ingeniører bruger måske 20% af hvad Git tilbyder. Agentisk ingeniørarbejde kræver de andre 80%.

6.1 Små Commits, Altid

Den vigtigste Git-vane for agentisk ingeniørarbejde: commit småt, commit ofte.

Når en agent laver ændringer, vil du kunne reviewe hvert logisk skridt uafhængigt. Et commit der siger "refaktorerede autentificering, opdaterede tests, fiksede navbaren og ændrede databaseskemaet" er umuligt at reviewe og umuligt at rulle delvist tilbage. Fire separate commits — hvert der gør én ting — giver dig kirurgisk kontrol.

Det betyder mere med agenter end med menneskelige udviklere, fordi agenter bevæger sig hurtigt. En agent kan lave tyve filændringer på tredive sekunder. Hvis de ændringer er bundlet i ét commit, og noget går i stykker, sidder du og vikler et rod fra hinanden. Hvis de er i fem rene commits, reverterer du det ene der brød tingene og beholder resten.

Træn dig selv — og dine agenter — til at committe ved naturlige grænser:

- Efter hver logisk ændring, ikke efter hver session
- Før du skifter til en anden bekymring
- Efter tests passerer, for at fange en known-good tilstand
- Før du forsøger noget risikabelt, for at oprette et gendannelsespunkt

6.2 Lad Agenter Skrive Dine Commit-Beskeder

Det her er en af de nemmeste gevinster i agentisk ingeniørarbejde, og det er næsten pinligt simpelt: lad agenten skrive commit-beskeden.

Tænk over det. Agenten har lige lavet ændringerne. Den ved præcis hvad den modificerede, hvorfor den modificerede det, og hvad intentionen var. Den har den fulde diff i konteksten. Den vil skrive en mere præcis, mere beskrivende commit-besked end du ville — fordi du ville opsummere fra hukommelsen, og agenten opsummerer fra fakta.

En typisk menneskelig commit-besked klokken 23: "fix auth bug"

En typisk agent commit-besked: "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"

Den anden er nyttig seks måneder fra nu når nogen — menneske eller agent — prøver at forstå hvorfor den kode eksisterer. Den første er støj.

Gør det til en vane. Når agenten fuldfører en opgave, bed den om at committe med en beskrivende besked. Eller konfigurér dit workflow så det sker automatisk. Kvaliteten af din git-historik vil forbedre sig fra den ene dag til den anden.

6.3 Branches Som Opgavegrænser

Hver agentopgave får sin egen branch. Det er ikke til forhandling.

Branchen tjener flere formål:

- **Isolering.** Agentens ændringer påvirker ikke din main branch før du eksplicit merger dem.
- **Review-scope.** Når du er færdig, reviewer du en enkelt PR — diffen mellem branchen og main. Det er et workflow enhver ingeniør allerede kender.
- **Rollback.** Hvis det hele er forkert, sletter du branchen. Rent. Ingen kirurgi nødvendig.
- **Parallelt arbejde.** Multiple agenter på multiple branches, der arbejder samtidigt, uden at træde hinanden over tæerne.

Navngiv dine branches beskrivende: `agent/refactor-auth-middleware`, `agent/add-user-tests`, `agent/fix-sidebar-rendering`. Når du kigger på din branchliste, bør du se et manifest over alt hvad dine agenter arbejder på.

6.4 Worktrees Til Parallelle Agenter

Branches alene er ikke nok til ægte parallelt arbejde. Hvis to agenter er på forskellige branches men deler det samme arbejdsbibliotek, vil de kæmpe om filsystemet. Git worktrees løser dette.

Et worktree er en separat checkout af dit repo — et andet bibliotek, på en anden branch, der deler den samme `.git`-historik. At oprette et tager sekunder:

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

Nu har du to biblioteker: din hoved-checkout og worktree'et. Hver agent får sit eget worktree, sin egen branch, sit eget filsystem. De kan begge køre tests, modificere filer og bygge — samtidigt, uden konflikter.

Når arbejdet er gjort:

- Godt resultat → merge branchen, fjern worktree'et
- Dårligt resultat → fjern worktree'et, slet branchen
- Behov for at iterere → behold worktree'et, fortsæt samtalen

Det er den billigste sandbox du kan bygge. Ingen containere, ingen VM'er, ingen cloud-ressourcer. Bare Git.

6.5 Review Af Agentarbejde Gennem Diffs

Din primære grænseflade til at reviewe agentarbejde er ikke at læse kode — det er at læse diffs.

Det er et subtilt men vigtigt skift. Når du selv skriver kode, reviewer du den mens du skriver. Når en agent skriver kode, reviewer du den bagefter. Og den mest effektive måde at gøre det på er gennem diffen mod din base branch.

Udvikl dine diff-læsningsfærdigheder:

- **Start med testændringerne.** Hvis agenten skrev eller modificerede tests, læs dem først. De fortæller dig hvad agenten mener koden bør gøre. Hvis testene matcher din intention, er implementeringen sandsynligvis fin.
- **Kig efter scope creep.** Ændrede agenten filer du ikke forventede? Urelaterede formateringsændringer? Ekstra afhængigheder? Det er røde flag.
- **Tjek grænserne.** Funktionssignaturer, API-kontrakter, databaseskemaer — ændringer af grænseflader har uforholdsmæssig stor påvirkning. Review dem omhyggeligt.
- **Stol men verificer.** Hvis diffen er stor, læs ikke hver linje. Spot-check de kritiske stier, sørg for at testene er meningsfulde, og kør suiten selv.

Målet er ikke at læse hver linje agenten skrev — det modvirker formålet. Målet er at verificere at agentens ændringer matcher din intention og ikke introducerer problemer. Diffs gør det hurtigt.

6.6 Git-Historik Som Dokumentation

Her er den indsigt de fleste ingeniører misser: agenter læser din git-historik. Når en agent forsøger at forstå hvorfor kode eksisterer, hvordan en feature udviklede sig, eller hvilken tilgang der blev prøvet før, kigger den på `git log` og `git blame`.

Det betyder at din commit-historik er dokumentation. Ikke den slags du skriver i en wiki — den slags der er indlejret i koden selv, permanent, med perfekt proveniens.

Gode commit-beskeder akkumulerer over tid. Seks måneder fra nu, når en agent arbejder på din codebase, vil den læse de beskeder for at forstå kontekst. Forskellen mellem en historik af "fix bug" og "fix race condition in session cleanup" er forskellen mellem en agent der forstår din codebase og en der gætter.

Det gælder også for PR-beskrivelser, branchnavne og merge commit-beskeder. Hver stump tekst du vedhæfter din Git-historik er kontekst for fremtidige agenter. Skriv derefter.

6.7 Git-Workflowet For Agentisk Ingeniørarbejde

Sat sammen ser workflowet sådan ud:

1. Opret en branch til opgaven
2. Opret eventuelt et worktree til isolering
3. Peg agenten mod worktree'et
4. Lad den arbejde — committe ved naturlige grænser
5. Agenten skriver beskrivende commit-beskeder
6. Review diffen mod main
7. Merge hvis godt, kassér hvis ikke

Dette workflow er hurtigt, sikkert og skalerer til multiple parallelle agenter. Det bruger Git-features der har eksisteret i årevis — branches, worktrees, diffs — men kombinerer dem på en måde der er formålsbygget til agentisk ingeniørarbejde.

Den bedste del: du kender allerede alt dette. Du har brugt Git i årevis. Agentisk ingeniørarbejde kræver ikke nye værktøjer — det kræver at du bruger dine eksisterende værktøjer mere bevidst.

7 Sandboxes

En sandbox er en gave du giver din agent: friheden til at tage fejl.

Når en agent opererer i en sandbox, kan den prøve ting uden konsekvenser. Installer en mærkelig afhængighed. Omskriv et modul fra bunden. Kør et script der måske crasher. Hvis det virker, fedt — du trækker resultatet ud. Hvis ikke, smider du sandboxen væk. Ingen oprydning, ingen rollback, ingen skade.

Det er ikke bare en sikkerhedsforanstaltning. Det ændrer fundamentalt hvor produktiv en agent kan være.

7.1 Frygten

Uden en sandbox bærer hver agenthandling risiko. Agenten tøver (eller rettere, du tøver med at lade den handle). Du tilføjer flere begrænsninger, flere godkendelsesporte, flere guardrails — og snart er agenten knap mere nyttig end en fancy autocomplete.

Sandboxes løser dette ved at gøre *omkostningen ved fejl* i bund og grund nul. Og når fejl er billigt, er eksperimentering gratis. En agent i en sandbox kan prøve tre forskellige tilgange til et problem, køre dem alle og lade dig vælge vinderen. Det er et workflow der er umuligt når hver handling er irreversibel.

7.2 Git Worktrees

Til kodefokuseret arbejde er git worktrees den letteste sandbox du kan bygge. Et worktree giver dig en fuld kopi af dit repo i et separat bibliotek, på sin egen branch, på sekunder.

Workflowet ser sådan ud:
1. Opret et worktree til opgaven

2. Peg agenten mod det
3. Lad den arbejde — commits, filændringer, testkørsler, hvad end den behøver
4. Review resultatet
5. Merge hvis godt, slet worktree'et hvis ikke

I praksis ser det sådan ud:

```
# Opret et isoleret arbejdsrum til agenten
git worktree add ../myapp-refactor agent/refactor-auth
cd ../myapp-refactor

# Agenten arbejder her — fuldstændig isoleret
# Når den er færdig, review og merge:
cd ../myapp
git merge agent/refactor-auth

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

Ingen containere at bygge, ingen VM'er at boote. Bare git. Det er særligt kraftfuldt når du kører multiple agenter parallelt — hver får sit eget worktree, sin egen branch, sit eget isolerede arbejdsrum.

Jeg har et shell-alias til dette fordi jeg bruger det så ofte:

```
# I .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 Containere

Til opgaver der rækker ud over kode — installation af systempakker, kørsel af services, test af infrastruktuændringer — er containere den naturlige sandbox. Docker giver dig et reproducerbart, isoleret miljø du kan spinne op på sekunder og ødelægge lige så hurtigt.

Nøglen er at gøre dit projekt containervenligt. En god `Dockerfile` og `docker-compose.yml` er ikke bare til deployment længere — de er agentinfrastruktur. Når dit projekt kan boote i en container med én kommando, har du givet enhver agent evnen til at arbejde i et rent rum.

En minimal agentvenlig Dockerfile ser sådan ud:

```
FROM node:22-slim
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
# Agenten kan køre tests, lint, build — alt isoleret
CMD ["npm", "test"]
```

Mønstret er: hurtig at bygge, nem at destruere, indeholder alt agenten behøver for at verificere sit eget arbejde. Hvis din container tager femten minutter at bygge, vil agenten ikke iterere hurtigt nok til at være nyttig. Optimér for rebuild-hastighed — lag dine afhængigheder, brug slanke base images, cache aggressivt.

7.4 Efemere Miljøer

De mest sofistikerede sandboxes er cloudbaserede efemere miljøer — kortlivede preview-deployments der spinnes op per branch eller per opgave. Services som Railway, Fly.io eller cloud dev environments giver dig en fuld kørende stack der er fuldstændig til at smide væk.

Det er vigtigt for integrationstest. En agent kan deploye sine ændringer til et efemert miljø, køre end-to-end tests mod rigtig infrastruktur, verificere at alt virker, og så beslutter du om det skal promoveres til staging. Agenten rører aldrig dine rigtige miljøer.

Økonomien er overbevisende. Et preview-miljø der kører i to timer mens agenten arbejder koster ører. Den produktionsincident det forhindrer koster tusindvis. Matematikken favoriserer altid mere sandboxing, ikke mindre.

7.5 Sandbox-Spektrummet

Der er et spektrum af sandboxing, og det rigtige valg afhænger af opgaven:

Worktrees — til rene kodeændringer. Hurtigst at oprette og destruere. Ingen isolering ud over filsystemet. Godt til: refaktoreringer, feature branches, testskrivning. Sekunder at sætte op.

Containere — til kode plus miljø. Isoleret filsystem, netværk og processer. Godt til: afhængighedsændringer, systemniveau-arbejde, alt der kan forurene din lokale maskine. Minutter at sætte op.

Efemere cloud-miljøer — til full-stack-verifikation. Rigtig infrastruktur, rigtige databaser, rigtig netværkstopologi. Godt til: integrationstest, deployment-verifikation, multi-service-ændringer. Minutter at sætte op, men koster rigtige penge.

VM'er — til maksimal isolering. Separat kerne, separat alt. Godt til: sikkerhedsfølsomt arbejde, upålidelige agenter, infrastrukturautomatisering. Minutter til timer at sætte op.

Start med worktrees. Bevæg dig op ad spektrummet når opgaven kræver det. Det meste dagligdags agentiske arbejde har aldrig brug for mere end et worktree.

7.6 Sandbox-Tankegangen

Den dybere lektie handler ikke om værktøjer — det handler om at designe dit workflow omkring disponibilitet. Hvis dit udviklingsmiljø tager en time at sætte op, er sandboxes upraktiske. Hvis det tager tredive sekunder, er de naturlige.

Investér i hurtig, reproducerbar opsætning. Investér i infrastructure as code. Investér i at gøre dit projekt nemt at boote fra bunden. De investeringer betaler sig selv tilbage hver gang en agent har brug for et sikkert rum at arbejde i — hvilket, efterhånden som du bliver bedre til agentisk ingeniørarbejde, er konstant.

En nyttig lakmustest: kan en ny udvikler (eller en ny agent) gå fra `git clone` til kørende tests på under to minutter? Hvis ikke, har du sandbox-

gæld. Hvert minut af opsætningsfriktion er et minut der modvirker sandboxing — og usandboxede agenter er agenter der arbejder uden net.

8 Testing Som Feedback Loop

To codebases. Samme agent. Samme opgave: "Tilføj en rate limiter til API'et der returnerer 429 efter 100 requests per minut per bruger."

I den første codebase er der ingen tests. Agenten læser route handlers, vælger et middleware-indsætningspunkt, skriver rate-limiting-logikken og… stopper. Den har ingen måde at vide om det virkede. Den kan ikke starte serveren og sende 101 requests for at se hvad der sker. Den kan ikke tjekke om eksisterende endpoints stadig svarer korrekt. Den producerer en diff, siger "Jeg har tilføjet rate limiting," og håber på det bedste. Du reviewer koden, kniber øjnene sammen, synes den ser rimelig ud, merger den, og opdager i produktion tre dage senere at middlewaren var mountet i forkert rækkefølge og aldrig blev eksekveret. Hver request sejlede igennem. Rate limiteren var dekoration.

I den anden codebase er der en testsuite. Agenten skriver rate-limiting-middlewaren, kører så testene. To eksisterende tests fejler — en health check-test der ikke forventede middleware-headers, og en auth-test hvor test-setupet lavede 150 hurtige requests og nu bliver throttlet. Agenten læser fejlene, justerer middlewaren til at skippe health-endpointet, opdaterer test-fixturen til at nulstille rate-counteren mellem tests, og kører igen. Grønt. Så skriver den tre nye tests: én der verificerer at den 101. request returnerer 429, én der verificerer at counteren nulstilles efter et minut, og én der verificerer at forskellige brugere har uafhængige limits. Alle passerer.

Samme agent. Samme opgave. Nat og dag i resultater. Forskellen var testsuiten.

I traditionel udvikling verificerer tests at din kode virker. I agentisk ingeniørarbejde gør tests noget mere fundamentalt: de fortæller agenten om den lykkedes. Det ændrer alt ved hvordan du tænker om testing.

Der er noget foruroligende ved det i starten. Din testsuite — den du skrev for at verificere *din* kode — bliver den spec en agent implementerer mod.

De tests du svedte over er ikke bare kvalitetssikring længere. De er definitionen af hvad "korrekt" betyder. Det er en mærkelig følelse: din tidligere indsats der bliver fundamentet for et workflow der ikke eksisterede da du skrev det. Men det er også, hvis du lader det være, dybt bekræftende. Alle de timer brugt på at skrive grundige tests? De var ikke bare god praksis. De var *investering* — og afkastet kommer nu.

8.1 Agentens Øjne

En agent kan ikke kigge på en UI og fortælle om den ser rigtig ud. Den kan ikke mærke om et API-svar er "hurtigt nok." Den kan ikke intuitivt vide om en refaktorering bevarede den subtile adfærd brugere afhænger af. Hvad den *kan* gøre er at køre din testsuite og læse resultaterne.

Tests bliver agentens primære feedback-mekanisme. Grønt betyder "fortsæt." Rødt betyder "prøv igen." Ingen tests betyder at agenten flyver blindt — gætter om dens ændringer virker, uden mulighed for at verificere.

Det er derfor utestede codebases er svære at arbejde med agentisk. Det er ikke bare et kvalitetsproblem — det er et informationsproblem. Uden tests har agenten intet signal. Det er som at bede nogen om at hænge en billedramme op med bind for øjnene. De rammer måske rigtigt, men det ville du ikke satse på.

Og kvaliteten af det signal er enormt vigtig. En test der siger `FAIL: expected 429, got 200` er handlingsbar. Agenten ved præcis hvad der gik galt og kan ræsonnere om hvorfor. En test der siger `FAIL: assertion error` uden kontekst er knap bedre end stilhed. Klarheden af dit testoutput er klarheden af agentens syn.

8.2 TDD Får Ny Betydning

Test-Driven Development var altid en god idé. Med agenter bliver det en superkraft.

Workflowet er simpelt: skriv testen først, giv den så til agenten og sig "få denne til at passe." Agenten har nu et klart, utvetydigt succeskriterium.

Den kan iterere — skrive kode, køre testen, læse fejlen, justere, gentage — i et stramt loop der tager sekunder per cyklus.

Her er hvad det ser ud i praksis. Lad os sige du har brug for en funktion der parser en varighedsstreng som `"2h30m"` til totale sekunder. Du skriver testen:

```
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")
```

Så fortæller du agenten: "Få `test_parse_duration` til at passe."

Agentens første forsøg håndterer måske timer og minutter men glemmer sekunder. Den kører testen: `FAIL: parse_duration("45s") returned 0, expected 45`. Klart signal. Den tilføjer sekundhåndtering, kører igen: `FAIL: parse_duration("abc") did not raise ValueError`. Endnu et klart signal. Den tilføjer inputvalidering. Grønt. Færdig.

Hver cyklus tog sekunder. Agenten behøvede aldrig at spørge dig hvad "korrekt" betyder — testene definerede det. Og fordi testen dækker edge cases du tænkte over på forhånd, håndterer implementeringen dem fra starten, ikke som bugs opdaget senere.

Det er fundamentalt anderledes end at bede en agent om at "bygge en varighedsparser." Det er vagt. Hvilket format? Hvilke edge cases? Hvad skal ske med dårligt input? Men "få disse syv assertions til at passe" er præcist. Agenten ved præcis hvordan succes ser ud, og den kan måle sin egen fremgang mod det.

Den agentiske ingeniør lærer at udtrykke intention gennem tests. Hver test er en kontrakt. Hver assertion er et krav. Jo bedre dine tests, jo bedre performer dine agenter. Du stopper med at tænke på testskrivning som overhead og begynder at tænke på det som at *programmere agenten* — du specificerer adfærd i det mest utvetydige sprog der findes: kode der enten passerer eller ikke gør.

8.3 Hvad Gør en God Agentisk Test

Ikke alle tests er lige nyttige for agenter. De tests der tjener mennesker godt under udvikling kan aktivt vildlede en agent. En god agentisk test har specifikke egenskaber, og det er værd at være bevidst om dem.

Hurtig. En agent itererer i et loop. Hvis hver cyklus tager ti minutter, prøver agenten seks tilgange per time. Hvis hver cyklus tager ti sekunder, prøver den 360. Hastighed er ikke bare bekvemmelighed — det er forskellen mellem en agent der konvergerer mod en løsning og en der timer ud. Kør den fulde suite hvis den er hurtig; kør kun de relevante tests hvis den ikke er. Under alle omstændigheder skal feedback loopet være stramt.

Deterministisk. En flaky test er værre end ingen test for en agent. Når en test fejler tilfældigt, trækker et menneske på skuldrene og kører igen. En agent ser en fejl og forsøger at fikse den. Den ændrer fungerende kode for at jagte et spøgelse. Så passerer den flaky test — ikke fordi agentens ændring var korrekt, men fordi de tilfældige stjerner stod rigtigt. Nu har du en meningsløs kodeændring der ligner et fix men ikke er det. Agenten er blevet belønnet for at gøre noget unyttigt. Hvis du har flaky tests, sæt dem i karantæne før du giver agenten adgang til suiten.

Isoleret. Tests der afhænger af eksekveringsrækkefølge, delt tilstand eller eksterne services skaber forvirrende fejl. Agenten ændrer funktion A og test B fejler — ikke på grund af en reel afhængighed, men fordi test B afhænger af tilstand som test A satte op. Agenten vil bruge cyklusser på at forsøge at forstå et forhold mellem A og B der ikke eksisterer i koden, kun i test-harnessen. Isolér dine tests. Hver enkelt bør sætte sin egen verden op og rive den ned.

Klare fejlbeskeder. `AssertionError: False is not True` fortæller agenten ingenting. `Expected user.status to be 'active' after calling activate(), but got 'pending'` fortæller den præcis hvad der gik galt. Gode assertion-beskeder er gratis dokumentation. Brug dem. Custom fejlbeskeder i dine assertions er den billigste investering du kan gøre i agentisk produktivitet.

Fokuseret på adfærd, ikke implementering. En test der asserter den interne struktur af en returværdi bryder når agenten refaktorerer. En test der asserter *adfærden* — "givet dette input, får jeg dette output"

— overlever refaktoreringer og giver agenten frihed til at finde bedre løsninger. Hvis dine tests begrænser implementeringen for stramt, kan agenten ikke forbedre den.

Lakmustesten: hvis du kun viste testfilen til en kompetent ingeniør uden anden kontekst, kunne de så skrive en korrekt implementering? Hvis ja, vil de tests fungere godt for en agent. Hvis nej — hvis testene er for vage, for koblede eller for flaky til at tjene som en pålidelig specifikation — fix testene før du giver dem til agenten.

8.4 Dækningsspørgsmålet

"Hvor meget testdækning har jeg brug for til effektivt agentisk arbejde?"

Instinktet er at sige 100%. Fuld dækning. Hver linje, hver branch, hver sti. Men det er hverken praktisk eller nødvendigt. Hvad du faktisk har brug for er *målrettet* dækning: nok til at agenten kan verificere den specifikke ting den ændrer.

Tænk på det sådan. Hvis du beder en agent om at modificere betalingsbehandlingsmodulet, har du brug for solid testdækning af betalingsbehandling. Du har ikke nødvendigvis brug for fuld dækning af e-mail-skabelonsystemet, admin-dashboardet eller CSV-eksportfunktionen. Agenten har brug for tests omkring den kode den rører, plus nok integrationstests til at bekræfte at den ikke har brudt grænsefladerne mellem systemer.

Det fører til en praktisk strategi: *dæk de varme stier først.* Se på hvor du faktisk vil pege agenter. Din kerne forretningslogik. Dine API-kontrakter. Dine datatransformationer. Det er de områder der behøver rigoristiske tests — ikke på grund af abstrakte kvalitetsmål, men fordi det er de områder hvor agenter vil arbejde.

Dækningsmetrikker kan endda være vildledende. En codebase med 90% linjedækning men ingen tests på betalingsflowet er værre, til agentiske formål, end en codebase med 40% dækning der grundigt tester betalinger, auth og API-laget. Agenten har ikke brug for et dækningsbadge. Den har brug for tests hvor arbejdet sker.

Når det er sagt, har integrationstests uforholdsmæssig stor værdi. En unit test fortæller agenten "denne funktion virker isoleret." En integrationstest fortæller agenten "disse komponenter virker sammen." Når en agent ændrer en funktion, fanger unit tests lokale regressioner og integrationstests fanger ripplevirkninger. Begge tæller, men hvis du starter fra nul, giver integrationstests mere for indsatsen fordi de verificerer *sømmene* — de steder hvor ting har tendens til at gå i stykker.

Én ting mere: glem ikke negative tests. Tests der verificerer at dit system *afviser* dårligt input er kritiske for agenter. Uden dem kan en agent "forenkle" din valideringslogik, få alle positive tests til at passe, og efterlade dig med et system der accepterer skrald. Hvis du har en valideringsregel, test begge sider af den.

8.5 Hastighed Betyder Noget

Når en agent itererer i et testloop, påvirker hastigheden af din testsuite direkte produktiviteten. En testsuite der tager ti minutter at køre betyder at agenten venter ti minutter mellem forsøg. En testsuite der tager ti sekunder betyder at agenten kan prøve dusinvis af tilgange i den tid det tager dig at hente kaffe.

Det skaber et stærkt incitament til at:

- Holde unit tests hurtige og isolerede
- Separere hurtige tests fra langsomme integrationstests
- Bruge watch modes der kun genkører påvirkede tests
- Investere i testinfrastruktur på samme måde som du investerer i CI

Hurtige tests er ikke bare en forbedring af udvikleroplevelsen længere. De er agentinfrastruktur.

Praktisk betyder det at du vil have en lagdelt teststrategi. En hurtig unit suite der kører på sekunder — agentens indre loop. En medium integrationssuite der kører på et minut — agenten kører denne før den kalder opgaven færdig. Og en langsom end-to-end suite der kører i CI — den endelige verifikation før merge.

Fortæl dine agenter hvilken suite de skal bruge. "Kør `pytest tests/unit` efter hver ændring. Kør `pytest tests/integration` når du tror du

er færdig." Det holder det indre loop hurtigt mens det stadig fanger integrationsproblemer før de når dig.

8.6 Testing Ud Over Kode

Agenter skriver ikke kun applikationskode. De skriver infrastrukturkonfigurationer, deployment-scripts, dokumentation og mere. Hver af disse kan — og bør — have en form for automatiseret verifikation.

Type checking er det hurtigste feedback loop du kan give en agent. En typefejl viser sig på millisekunder, før en eneste test kører. I typede sprog, eller i Python med mypy, eller i JavaScript med TypeScript, fanger type checking en enorm klasse af fejl øjeblikkeligt. Agenten omdøber et felt, og type checkeren flagger øjeblikkeligt hvert sted der refererer til det gamle navn. Det er et strammere feedback loop end nogen testsuite kan give.

Linting og formattering fanger en anden klasse af problemer. En fejlkonfigureret ESLint-regel kan virke som en mindre irritation, men for en agent er en lint-fejl et utvetydigt signal. "Dit import er ubrugt." "Denne variabel er erklæret men aldrig læst." Det er små korrektioner der akkumulerer til renere kode med nul indsats fra dig.

Skemavalidering for API-kontrakter sikrer at agentens ændringer ikke bryder grænsefladen mellem services. Hvis du har OpenAPI-specs, JSON Schema-definitioner eller GraphQL-typedefinitioner, validér mod dem. En agent der ændrer en response-payload vil øjeblikkeligt lære at den brød kontrakten, i stedet for at opdage bruddet når en downstream-service crasher i staging.

Kontrakttest mellem services tager dette videre. Hvis service A afhænger af service B's API, verificerer en kontrakttest at B stadig opfylder hvad A forventer — uden at skulle køre begge services samtidigt. Når en agent modificerer service B, fanger kontrakttestene brydende ændringer som unit tests inden i B ville misse helt.

Konfigurationsfilvalidering er kriminelt underbrugt. En YAML-lint på dine Kubernetes-manifester. En terraform validate på din infrastrukturkode. Et docker-compose config-check. Det tager sekunder at køre og fanger fejl der ellers ville dukke op som mystiske fejl under deployment.

Hvert automatiseret tjek du tilføjer er endnu et signal agenten kan bruge til at selvkorrigere.

Den agentiske ingeniør tænker bredt om testbarhed: ikke "returnerer min funktion den rigtige værdi?" men "kan jeg automatisk verificere at denne ændring er korrekt?"

8.7 Når Tests Vildleder

En dårlig testsuite er værre end ingen testsuite. Det er ubehageligt men værd at sidde med.

Uden tests ved agenten at den flyver blindt. Den vil være konservativ. Den vil fortælle dig at den ikke kan verificere sine ændringer. Du vil reviewe mere omhyggeligt. Manglen på signal er i det mindste en ærlig mangel på signal.

Med dårlige tests flyver agenten med falsk tillid. Den laver en ændring, testene passerer, og den rapporterer succes. Du ser grønt og slapper af i din review. Men testene verificerede faktisk ikke den rigtige ting.

Det sker på flere forudsigelige måder.

Tests der tester mocken, ikke koden. Når din test mocker databasen, HTTP-klienten, filsystemet og køen — og så asserter at den mockede funktion blev kaldt med de rigtige argumenter — tester du din testopsætning, ikke din applikationslogik. En agent kan få den "rigtige" kode til at gøre hvad som helst og testen vil stadig passe, så længe mock-forventningerne er opfyldt. De tests giver et grønt signal der ikke betyder noget.

Tests der er for løse. `assert response.status_code == 200` fortæller dig at endpointet ikke crashede. Det fortæller dig ikke at svaret indeholder de rigtige data. En agent kunne returnere en tom body, den forkerte brugers data, eller et svar der mangler halvdelen af sine felter, og den assertion ville stadig passe. Specificitet i assertions er specificitet i feedbacksignalet.

Tests der duplikerer implementeringen. Hvis din test i bund og grund genimplementerer funktionen under test og tjekker at begge returnerer det samme, verificerer den ingenting. Agenten kan ændre

implementeringen og testen sammen — og vil, hvis den mener de bør matche. Du ender med kode og tests der er enige med hinanden men ikke med virkeligheden.

Det hænger direkte sammen med "Det Selvsikre Forkerte Svar"-mønstret fra krigshistoriekapitlet. Agenten der tilføjede en `time.Sleep(500)` for at fikse en race condition? Testene passerede. De passerede fordi testmiljøet havde lav concurrency hvor 500ms altid var nok. Testene var *teknisk korrekte* men *praktisk vildledende*. De gav et grønt signal for et fix der ville fejle under produktionsbelastning.

Forsvaret er ligetil men kræver disciplin. Review dine tests med samme grundighed som du reviewer din kode. Spørg: "Hvis agenten introducerede en subtil bug, ville denne test fange den?" Hvis svaret er nej, er testen møbler — den får rummet til at se beboet ud men gør faktisk ingenting.

8.8 Den Dydige Cyklus

Når du sætter alt dette sammen — gode tests, hurtig feedback, sandboxede miljøer og en agent i loopet — klikker noget.

Agenten tager en opgave op. Den læser de eksisterende tests for at forstå forventet adfærd. Den laver en ændring. Den kører den hurtige testsuite — ti sekunder. To fejl. Den læser fejlbeskederne, forstår problemet, justerer. Kører igen. Grønt. Den skriver nye tests til den nye adfærd. Kører den fulde integrationssuite — ét minut. Alt grønt. Den committer med en beskrivende besked og giver dig en diff som du ved passerer hvert automatiseret tjek i dit system.

Du reviewer en diff som du ved passerer alle tests. Dit job skifter fra "tjek om dette virker" til "tjek om dette er den rigtige tilgang." Det er en meget bedre brug af din tid, og en meget bedre brug af din ekspertise. Du er ikke længere en menneskelig testkører. Du er en arkitekt der reviewer designs.

Og her er den akkumulerende effekt. Hver gang en agent arbejder på din codebase og testsuiten hjælper den med at lykkes, har du bevist værdien af de tests. Hver gang en manglende test forårsager en bug, mærker du smerten med det samme — og du tilføjer testen. Din testsuite vokser

præcis de steder der tæller, styret af reel feedback fra rigtigt agentisk arbejde.

Det er den dydige cyklus i agentisk ingeniørarbejde: bedre tests fører til mere autonome agenter, som fører til hurtigere iteration, som giver dig mere tid til at skrive bedre tests. Hver omgang af cyklussen gør den næste hurtigere.

De ingeniører der får mest ud af agentiske værktøjer er ikke dem med de klogeste prompts eller de mest kraftfulde modeller. De er dem med de bedste testsuiter. En veltestet codebase er en kraftmultiplikator der akkumulerer med hver opgave du giver til en agent. En utestet codebase er en skat der gør hver opgave langsommere og mere risikabel.

Hvis du tager én ting med fra dette kapitel, lad det være dette: før du optimerer dine prompts, før du eksperimenterer med nye modeller, før du bygger avanceret orchestration — gå og skriv tests. Skriv dem til den kode dine agenter vil røre. Gør dem hurtige, deterministiske, isolerede og specifikke. Den investering vil betale sig mere end noget andet i denne bog.

Din testsuite er ikke overhead. Den er fundamentet alt andet er bygget på.

9 Konvention Over Konfiguration

Jeg så den samme agent producere kode der var en senioransættelse værdig i ét projekt og uvedligeholdelig skrald i et andet — på den samme eftermiddag, på den samme maskine, med den samme model. Forskellen var ikke prompten. Det var codebasen.

To projekter. Samme tech stack — TypeScript, React, PostgreSQL. Samme holdstørrelse. Samme agentværktøj.

Projekt A har et strengt mappelayout. Hvert API-endpoint følger det samme mønster: en handler-fil, en skemafil, en testfil, navngivet identisk. Databaselaget bruger et konsistent repository-mønster. Der er en `CLAUDE.md` i roden der beskriver arkitekturen på to sider. Når en ingeniør peger en agent mod en ticket — "tilføj et nyt endpoint til brugernotifikationer" — læser agenten de eksisterende endpoints, følger mønstret og producerer en pull request der ser ud som om et menneske på holdet skrev den. Reviewet tager tre minutter.

Projekt B er den anden slags. Codebasen er vokset organisk over to år. Nogle endpoints er i `routes/`, nogle i `api/`, nogle i `handlers/`. Halvdelen af databaseforespørgslerne bruger en ORM, den anden halvdel bruger rå SQL. Der er ingen konsistent fejlhåndtering — nogle funktioner thrower, nogle returnerer fejlobjekter, nogle returnerer null. Agenten kigger på denne codebase og gør sit bedste, men "sit bedste" betyder at vælge det mønster den så senest. Den resulterende kode virker, teknisk set, men matcher ikke noget omkring den. Reviewet tager tredive minutter, det meste brugt på "sådan gør vi det ikke her."

Forskellen mellem disse projekter er ikke talent. Det er ikke værktøj. Det er konvention.

Der er et gammelt princip i software: konvention over konfiguration. Gør standarden til det rigtige. Reducér antallet af beslutninger der skal tages. Når alle følger de samme mønstre, forklarer koden sig selv.

Det princip var altid nyttigt for menneskelige teams. For agentisk ingeniørarbejde er det essentielt.

Hvis det lyder som om jeg fortæller dig at gøre det glamourløse arbejde — skrive dokumentation, håndhæve navnekonventioner, vedligeholde projektstruktur — så er det præcis det jeg gør. Og jeg ved hvordan det føles. Du blev ikke ingeniør for at skrive stilguider. Men det er et af de øjeblikke hvor håndværket beder dig om at bekymre dig om noget der plejede at føles som overhead, fordi indsatsen har ændret sig. Konvention plejede at være en høflighed mod dit fremtidige jeg. Nu er det operativsystemet dine agenter kører på.

9.1 Hvorfor Agenter Elsker Konvention

En agent der navigerer i en ukendt codebase gør det samme som en nyansat: den leder efter mønstre. Hvor bor tests? Hvordan er filer navngivet? Hvad er importkonventionen? Hvor er konfigurationen?

Hvis dit projekt følger stærke konventioner, opfanger agenten mønstrene hurtigt og producerer kode der passer ind. Hvis hver fil er en snefnug — forskellige navne, forskellig struktur, forskellige stile — famler agenten. Den ved ikke hvilket mønster den skal følge, så den opfinder sit eget, og resultatet føles fremmed.

Konvention er *implicit kontekst*. Det er information agenten absorberer fra strukturen af din kode uden at du behøver at forklare det. Når dine testfiler altid bor ved siden af de kildefiler de tester, navngivet `foo.test.ts` ved siden af `foo.ts`, behøver agenten ikke at få at vide hvor den skal putte en ny test. Den læser mappen, ser mønstret og følger det. Når dine API-handlers alle eksporterer den samme form — en handler-funktion, et skema, et sæt middleware — producerer agenten en ny handler der eksporterer præcis den samme form.

Det er derfor meningsfulde frameworks altid har været produktive, og hvorfor de er *endnu mere* produktive i den agentiske æra. Rails, Next.js,

Laravel — de påtvinger en struktur. Den struktur er ikke bare for mennesker. Det er et sprog agenten taler flydende.

9.2 CLAUDE.md Dybt Dyk

En voksende konvention i agentisk ingeniørarbejde er `CLAUDE.md`-filen: et dokument i roden af dit projekt der fortæller agenten hvad den behøver at vide. Ikke en README til mennesker. En briefing til agenter.

Det er en af de ting med højest løftestangseffekt du kan gøre for dit agentiske workflow, og de fleste teams springer det enten over eller skriver et par vage linjer og kalder det gjort. Lad os snakke om hvad en god en faktisk ser ud.

En stærk `CLAUDE.md` har fem sektioner:

Projektoversigt. To eller tre sætninger. Hvad gør den her ting, hvad er tech stacken, hvad er deployment-målet. En agent der ved at den arbejder på "en B2B SaaS faktureringsplatform bygget med Go og PostgreSQL, deployet til Kubernetes" træffer fundamentalt anderledes beslutninger end en der gætter.

Build- og testkommandoer. Hver kommando agenten kan have brug for, listet eksplicit. Ikke "tjek Makefilen" — de faktiske kommandoer. Agenter læser dokumentation bogstaveligt. Hvis din `CLAUDE.md` siger `make test`, vil agenten køre `make test`. Hvis den siger "kør testene" uden at specificere hvordan, vil agenten gætte, og den gætter måske forkert.

Arkitekturbeslutninger. De ting der ikke er åbenlyse fra koden. Hvorfor du valgte en monorepo. Hvorfor auth-servicen er separat. Hvorfor du bruger event sourcing til ordrepipen men simpel CRUD til brugeradministration. Det er de beslutninger der former hvert stykke ny kode, og en agent der ikke kender dem vil bryde dem konstant.

Konventioner. Din stil. Hvordan du navngiver ting. Hvordan du håndterer fejl. Hvordan din importrækkefølge ser ud. Om du foretrækker tidlige returns eller nestede conditionals. De ting der får kode til at føles som om den hører til i *dette* projekt.

Almindelige faldgruber. Hvor ligene er begravet. Databasemigrationen der altid skal køres før seed. Den environment variable der ikke er i `.env.example` men er påkrævet for betalingsflowet. Testen der er flaky på CI men ikke lokalt. Ethvert projekt har disse — skriv dem ned.

Her er hvad en rigtig `CLAUDE.md` ser ud for et mellemstort projekt:

```
# 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
```

Det er ikke langt. Det tog måske tredive minutter at skrive. Men hver agentsession der læser denne fil starter med mere kontekst end de fleste menneskelige udviklere får i deres første uge.

Den vigtigste disciplin: hold den opdateret. En forældet `CLAUDE.md` er værre end slet ingen, fordi agenten vil stole på den. Når du ændrer en konvention, opdater filen. Når du tilføjer en ny service, tilføj den til arkitektursektionen. Når nogen opdager en ny faldgrube, dokumenter den. Behandl den som kode — den lever i versionskontrol, den bliver reviewed i PR'er, den er en del af projektet.

Nogle teams går videre: de putter `CLAUDE.md`-filer i undermapper også. En `packages/api/CLAUDE.md` der dækker API-specifikke mønstre. En `packages/web/CLAUDE.md` der dokumenterer komponentbibliotekets konventioner. Jo dybere agenten går ind i projektet, jo mere specifik kontekst den får. Det er som et løg af dokumentation — bred kontekst ved roden, specifik kontekst efterhånden som du dykker ned.

9.3 Konventioner Som Agenthukommelse

Konventioner er det tætteste agenter kommer på langtidshukommelse. Når du ser det, ændrer det hvordan du tænker om dem alle.

En agents kontekstvindue nulstilles hver session. Den husker ikke hvad den gjorde i går. Den husker ikke arkitekturdiskussionen du havde i sidste uge. Den husker ikke at de sidste tre gange den brugte `fetch` direkte, bad du den om at bruge API-client wrapperen i stedet.

Men din projektstruktur persisterer. Dine filnavne persisterer. Din `CLAUDE.md` persisterer. Dine linter-regler persisterer. Dine testmønstre persisterer. Alt du koder ind i formen af dit projekt er der hver gang agenten åbner sine øjne.

Det reframer konventioner fuldstændigt. De handler ikke bare om konsistens for mennesker. De er *ekstern hukommelse* for agenter. Hver konvention du etablerer er en lektion agenten ikke behøver at genlære.

Tænk over hvad der sker uden konventioner. Session ét: agenten opretter et nyt endpoint og lægger fejlhåndteringen inline. Du retter det i review: "Vi bruger AppError-klassen." Session to: agenten opretter endnu et endpoint og laver den samme fejl, fordi den ikke husker session ét. Session tre: det samme. Du har den samme samtale igen og igen.

Tilføj nu én konvention — et dokumenteret fejlhåndteringsmønster, håndhævet af en linter-regel — og problemet forsvinder permanent. Agenten læser den eksisterende kode, ser mønstret, følger det. Linteren fanger enhver afvigelse. Lektien er kodet ind i selve projektet, ikke i nogens hukommelse.

Det er derfor de mest effektive agentiske teams er besat af konventioner der virker kedelige. Konsistent importrækkefølge. Streng filnavngivning. Standard funktionssignaturer. Det er ikke æstetiske præferencer — det er hukommelse. Det er holdets akkumulerede visdom, gemt i et format der overlever kontekstvindue-nulstillinger.

Analogien jeg bliver ved med at vende tilbage til: konventioner er for agenter hvad institutionel viden er for organisationer. En virksomhed hvor alt lever i én persons hoved er skrøbelig. En virksomhed med stærke processer og dokumentation er resilient. Det samme gælder for codebases. Et projekt hvor "hvordan vi gør tingene" kun lever i seniorudviklerens hukommelse er skrøbeligt. Et projekt hvor det er kodet ind i strukturen, værktøjerne og dokumentationen er resilient — for mennesker og agenter i lige mål.

9.4 Praktiske Konventioner Der Hjælper Agenter

Konsistent filnavngivning. Hvis dine API-routes lever i `routes/`, dine modeller i `models/`, og dine tests i `__tests__/` — kan agenten navigere dit projekt uden et kort. Navngiv filer efter hvad de indeholder. Hold det kedeligt.

Formatters og linters. Værktøjer som Prettier, ESLint, `gofmt` eller Ruff er ikke bare til kodestil — de er guardrails der sikrer at agentgenereret kode matcher dit projekts standarder automatisk. Kør dem ved gem, kør dem i CI, gør dem ufravigelige.

Standard projektlayout. Uanset om det er Go standard-layoutet, Rails-konventioner eller dit teams egen struktur — vælg ét og hold dig til det. En `justfile` eller `Makefile` i roden der lister standardkommandoerne (`build`, `test`, `lint`, `dev`) giver agenter et indgangspunkt til ethvert projekt.

Små, fokuserede filer. Agenter arbejder bedre med filer under et par hundrede linjer. Når en fil indeholder én bekymring — én komponent, ét modul, ét sæt relaterede funktioner — kan agenten læse den, forstå den og modificere den uden at vade igennem urelateret kode.

Beskrivende commit-beskeder. Agenter læser git-historik. Et commit der siger "fix bug" lærer ingenting. Et commit der siger "fix race condition in session cleanup when WebSocket disconnects during auth" giver agenten kontekst om hvad der var vigtigt, hvad der var skrøbeligt, og hvordan holdet tænker om problemer.

Konsistent fejlhåndtering. Vælg et mønster og håndhæv det overalt. Custom fejlklasser med fejlkoder. En central fejlhåndterer. En standard fejlsvar-form. Når agenten støder på en fejlsituation i ny kode, bør der være nul tvetydighed om hvordan den skal håndteres. Hvis dit projekt bruger tre forskellige fejlhåndteringstilgange i tre forskellige filer, vil agenten producere en fjerde.

Standard API-svarformater. Hvert endpoint returnerer den samme form: `{ data, error, meta }` eller hvad du vælger. Statuskoder følger de samme regler overalt. Paginering virker på samme måde på hvert liste-endpoint. Det er den slags konsistens agenter excellerer i at vedligeholde — men kun hvis konventionen er klar fra den eksisterende kode.

Logningskonventioner. Struktureret logning med konsistente felter. Hver logindgang inkluderer et request-ID, et tidsstempel og et alvorlighedsniveau. Hver fejllog inkluderer fejlkoden og et stack trace. Når agenten tilføjer logning til ny kode — og det bør den — skal loggen se identisk ud med alle andre logs i systemet.

Mappestruktur der fortæller en historie. En agent bør kunne `ls` topniveauet af dit projekt og forstå arkitekturen. Her er hvad et velstruktureret projekt ser ud fra en agents perspektiv:

```
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
```

Hvert mappenavn er et substantiv. Hver fil inden i indeholder præcis hvad mappenavnet lover. Der er intet sted at blive forvirret om hvor ny kode bør gå. En agent der læser denne struktur ved øjeblikkeligt: "Jeg skal tilføje en ny databaseforespørgsel, den hører til i `repositories/`. Jeg skal lave et nyt endpoint, det starter i `routes/`."

Sammenlign det med et projekt hvor databaseforespørgsler er spredt ud over handler-filer, forretningslogik lever i hjælpefunktioner, og der er tre forskellige mapper der alle ser ud til at indeholde "helpers." Agenten vil putte kode *et* sted, men det er nok ikke det *rigtige* sted.

9.5 Konventionsskatten

Lad os være ærlige om omkostningen. At etablere konventioner tager tid, og det føles som overhead — særligt i starten.

At skrive en `CLAUDE.md` tager en eftermiddag. At sætte linters og formatters op tager en dag. At refaktorere en inkonsistent codebase til at følge ét enkelt mønster tager en uge, måske mere. At håndhæve konventioner i code review betyder at bremse PR'er der "virker fint" men ikke følger standarden.

Der er en reel fristelse til at springe alt dette over. Koden virker. Testene passerer. Hvorfor bruge tid på navnekonventioner når der er features der skal shippes?

Det ærlige svar: hvis du er en solotvikler der bygger en prototype til at smide væk, er konventionsskatten nok ikke det værd. Kør hurtigt, ship det, smid det væk.

Men i det øjeblik et andet par øjne rører din codebase — menneskelige *eller* agent — begynder konventioner at betale sig selv. Og afkastet akkumulerer.

Første gang du etablerer en konvention koster det dig en time. Hver efterfølgende gang en agent følger den konvention i stedet for at spørge dig hvordan den skal håndtere det, sparer du fem minutter. Lav regnestykket: efter tolv agentsessioner har konventionen betalt sig selv. Efter hundrede sessioner har du sparet *timer*.

Det akkumulerer på tværs af holdet. Fem ingeniører, der hver kører multiple agentsessioner per dag, der alle drager fordel af de samme konventioner. Den initiale investering af én ingeniørs eftermiddag sparer holdet hundredvis af timer over et år.

De teams der investerer i konventioner tidligt ser langsommere ud i starten. De bruger tid på "kedelige" ting — linterkonfigurationer, projektstruktur, dokumentation. Men tre måneder inde producerer deres agenter kode der kræver minimal review. Seks måneder inde shipper de dobbelt så hurtigt som holdet der sprang konventionsarbejdet over. Et år inde er gabet pinligt.

Det er den samme dynamik som teknisk gæld, bare vendt om. Konvention er teknisk *formue* — og ligesom finansiel formue, jo tidligere du begynder at investere, jo mere dramatisk er renters rente.

Den største fejl jeg ser er teams der venter til deres codebase er et rod før de forsøger at etablere konventioner. Det er det dyreste tidspunkt at gøre det. Det billigste tidspunkt er ved starten af et projekt — men det næstbilligste tidspunkt er i dag.

9.6 Den Akkumulerende Effekt

Hver konvention du etablerer, hver standard du håndhæver, hvert stykke dokumentation du vedligeholder — det akkumulerer alt sammen. Hvert gør agenter en smule mere effektive, en smule mere autonome, en smule mere tilbøjelige til at producere kode der passer til dit projekt som en handske.

Men akkumuleringen er ikke bare additiv. Konventioner interagerer. En konsistent filnavnekonvention *plus* en standard mappestruktur *plus* en `CLAUDE.md` der beskriver arkitekturen — tilsammen giver disse agenten en mental model af hele projektet. Den ved hvor den finder ting, hvad den skal kalde dem, og hvordan de passer sammen. Fjern bare én af de tre, og agentens effektivitet falder uforholdsmæssigt.

Det er derfor halvhjertede konventioner er næsten lige så dårlige som ingen konventioner. Et projekt med konsistent filnavngivning men kaotisk mappestruktur sender blandede signaler. Agenten ser orden i én dimension og kaos i en anden, og den ved ikke hvilket signal den skal stole på.

De ingeniører der investerer i konvention tidligt har ikke bare renere codebases. De har codebases der er klar til den agentiske fremtid. Deres agenter producerer bedre resultater. Deres reviews er hurtigere. Deres iterationscyklusser er strammere.

Konvention er ikke glamourøst arbejde. Det er det mindst spændende kapitel i denne bog. Men det er det der får alle andre kapitler til at fungere. Sandboxene, teststrategierne, multi-agent orchestrationen — det hele fungerer bedre når den underliggende codebase er konsistent, dokumenteret og konventionel.

Din codebase er det miljø dine agenter lever i. Gør det læseligt. Gør det forudsigeligt. Gør det kedeligt. Agenterne vil takke dig ved at skrive kode der ser ud som om den hører til.

10 Lokale LLM'er vs. Kommercielle LLM'er

En ven af mig — senioringeniør i en Series B-startup — pingede mig en fredag eftermiddag. "Jeg har lige fået vores første rigtige API-regning. Toogtyve hundrede dollars. For *marts*." Han havde kørt Claude Code på tværs af sit hold på otte ingeniører, som alle itererede på features, debuggede, refaktorerede. Ingen havde sat tokenbudgetter. Ingen holdt øje med måleren. Agenterne arbejdede fremragende, og regningen var til at få vand i øjnene af.

Samme uge talte jeg med en ingeniør i et sundhedsselskab i München. Hun kørte Llama 70B på en lokal GPU-server fordi deres patientdatapipeline ikke *måtte* røre eksterne API'er. Ikke "burde ikke" — *måtte ikke*. Deres compliance-team havde gjort det klart skriftligt. Hun fik acceptable resultater til fokuserede opgaver, men hver gang hun havde brug for kompleks multi-fil-ræsonnering, faldt modellen fra hinanden og hun endte med at lave arbejdet manuelt.

Disse to historier er bogstøtterne for det samme spørgsmål: hvor kører modellen? Det lyder som en infrastrukturbeslutning. Det er i virkeligheden en beslutning om tillid, omkostninger, kapacitet og — i stigende grad — hvordan du arkitekterer hele dit agentiske workflow.

10.1 Kommercielle LLM'er: Frontlinjen

Kommercielle modeller — Claude, GPT, Gemini — er hvor frontlinjen er. Hvis du laver seriøst agentisk ingeniørarbejde, har du sandsynligvis brugt det meste af din tid her. Der er gode grunde til det.

10.1.1 Kontekstvinduer

Kontekst er alt for agenter. En agent læser ikke bare din prompt — den læser filer, tool-outputs, fejlbeskeder, testresultater og sin egen tidligere

ræsonnering. En enkelt kompleks opgave kan nemt fylde 50.000 tokens af kontekst, og en multi-step debugging-session kan blæse forbi 100.000.

Frontier kommercielle modeller tilbyder kontekstvinduer på 128K tokens og derover. Det betyder enormt meget for agentisk arbejde fordi agenten har brug for at holde din codebase i hovedet. Når konteksten løber tør, begynder agenten at glemme tidligere filer den læste, tidligere beslutninger den tog, tidligere fejl den så. Den degraderer fra en kapabel ingeniør til nogen med hukommelsestab.

Lokale modeller topper typisk ved 8K-32K kontekst i praksis, selvom de teknisk understøtter mere på papiret. Kvaliteten af opmærksomheden degraderer når du presser mod grænsen. En kommerciel model ved 100K kontekst ræsonnerer stadig godt. En lokal model ved 32K kontekst mister ofte tråden.

10.1.2 Tool Use-Kvalitet

Agenter lever og dør af tool use. At læse filer, skrive kode, køre kommandoer, søge i codebases — det er ikke valgfrie ekstra. Det er kerneloopet. Og kvaliteten af tool use varierer dramatisk mellem modeller.

Frontier kommercielle modeller er specifikt trænet og tunet til tool calling. De formaterer argumenter korrekt, de kæder tool calls logisk, de genopretter elegant når et tool call fejler. De ved hvornår de skal læse en fil før de redigerer den. De ved hvornår de skal køre tests efter de har lavet ændringer.

Mindre modeller — inklusive de fleste lokale modeller — er mindre pålidelige her. De hallucinerer filstier, glemmer at sende påkrævede argumenter, kalder tools i forkert rækkefølge eller sidder fast i loops der kalder det samme fejlende tool igen og igen. Forskellen er ikke subtil. På en kompleks opgave med ti eller femten tool calls, kan en frontier-model lykkes 90% af gangene hvor en lokal model lykkes 40%.

10.1.3 Instruktionsfølge

Når du fortæller en frontier-model "modificer kun filer i `src/auth/`-mappen" eller "ændr ikke det offentlige API, kun den interne implementering," lytter den generelt. Den følger system prompts, respekterer begrænsninger og holder sig inden for grænser.

Mindre modeller driver. De starter godt, og ignorerer gradvist dine begrænsninger efterhånden som samtalen bliver længere og konteksten fyldes op. Det er et reelt problem for agentisk arbejde, hvor præcision tæller. En agent der "for det meste" følger instruktioner vil af og til omskrive en fil du ikke bad den om at røre, eller refaktorere noget du eksplicit sagde den skulle lade være. I et sandboxet miljø er det bare irriterende. Uden sandbox er det farligt.

10.1.4 Valg Af Den Rigtige Kommercielle Model

Ikke alle kommercielle modeller er udskiftelige, selv på frontlinjen. Her er hvad jeg har fundet i praksis:

Til kompleks multi-fil refaktorering og arkitekturarbejde — brug den mest kapable model du har råd til. Det er her ræsonneringskvalitet betyder mest. Prisforskellen mellem en mid-tier og top-tier model er et par dollars per opgave. Kvalitetsforskellen er ofte forskellen mellem en ren diff og et rod du er nødt til at lave om manuelt.

Til fokuserede single-fil-opgaver — at skrive tests, implementere en veldefineret funktion, fikse en tydelig bug — performer mid-tier-modeller næsten lige så godt som top-tier, til en brøkdel af prisen. Opgaven er scopet nok til at modellen ikke behøver at jonglere mange bekymringer på én gang.

Til højvolumen, lavkompleksitetsarbejde — at generere boilerplate, formatere kode, skrive commit-beskeder — er den billigste model der kan følge instruktioner det rigtige valg. Du kører disse opgaver hundredvis af gange. Per-token-besparelsen akkumulerer.

Den fejl jeg ser oftest er at bruge en enkelt model til alt. Det er som at køre en Formel 1-bil i supermarkedet. Match modellen til opgaven.

10.2 Lokale LLM'er: Det Fulde Billede

At køre en model lokalt — Llama, Mistral, Qwen, DeepSeek, Codestral, eller nogen af de open-weight modeller via Ollama, llama.cpp eller vLLM — giver dig noget kommercielle API'er ikke kan: komplet datasuverænitet og nul marginalomkostning per token.

Din kode forlader aldrig din maskine. Du kan fodre den med proprietær kildekode, interne dokumenter, produktionslogs, kundedata, API-nøgler du ved et uheld efterlod i en konfiguration — intet af det krydser en netværksgrænse. Og når modellen er downloadet, er hver inference gratis. Kør den tusind gange, kør den hele natten, ingen sender dig en regning.

Men lad os være ærlige om oplevelsen, fordi markedsføringen omkring lokale modeller ofte ikke er det.

10.2.1 Hardware-Virkeligheden

Hardwarespørgsmålet er det første alle spørger om, og svaret er mindre glamourøst end "kør AI lokalt!"-blogindlæggene antyder.

MacBook Pro med Apple Silicon (M2/M3/M4 Pro eller Max, 32-64GB RAM). Det er sweet spottet for individuelle udviklere. Du kan køre en 7B-14B parameter model komfortabelt, og en 32B model hvis du har RAM'en. Inference vil være mærkbart langsommere end en kommerciel API — måske 15-30 tokens per sekund for en 14B model, sammenlignet med 80+ fra en kommerciel API. En 70B model vil teknisk køre på en 64GB maskine men den vil være langsom nok til at teste din tålmodighed. Du vil vente 10-20 sekunder på svar der tager 2 sekunder fra en API.

Dedikeret GPU-server (NVIDIA RTX 4090 eller A100). Det er her lokal inference bliver oprigtigt hurtig. En 4090 med 24GB VRAM kan køre en 14B model ved næsten-kommercielle hastigheder. En A100 med 80GB kan køre en 70B model komfortabelt. Men nu vedligeholder du hardware. Du håndterer CUDA-drivere, VRAM-styring og den lejlighedsvise "hvorfor skriger min GPU-blæser klokken 3 om natten"-hændelse.

Forbrugerhardware (16GB MacBook Air, ældre maskiner, ingen diskret GPU). Du er begrænset til 7B-modeller, og oplevelsen vil være middelmådig. Inference er langsom, modellerne er for små til seriøst agentisk arbejde, og du vil bruge mere tid på at vente end på at arbejde. Jeg ville ikke anbefale dette til andet end eksperimentering.

Det ærlige sammendrag: lokale modeller er praktiske for udviklere med en nyere MacBook Pro eller en dedikeret GPU. Under den hardware-tærskel vil du have en frustrerende oplevelse. Over den vil du have et genuint nyttigt værktøj — bare et andet værktøj end en kommerciel API.

10.2.2 Hvor Lokale Modeller Skinner

Lokale modeller er ikke bare "dårligere kommercielle modeller." Der er workflows hvor de genuint giver mere mening:

Højfrekvente, lavrisiko-opgaver. At generere docstrings, skrive commit-beskeder, oprette boilerplate-kode, formatere data. Disse opgaver har ikke brug for en genimodel — de har brug for en hurtig, gratis model. At køre dem lokalt betyder at du kan fyre-og-glemme uden at tænke på omkostninger.

Følsomme codebases. Vi dækker dette mere i privatlivssektionen, men det er en legitim og voksende use case. Hvis din kode ikke kan forlade dit netværk, er lokale modeller den eneste mulighed, og "godt nok" er uendeligt bedre end "ikke tilgængeligt."

Offline-udvikling. I et fly, i et tog, i en bunker — din lokale model virker uden WiFi. Det lyder underordnet indtil du er på en tolv-timers flyvetur og prøver at debugge noget.

Eksperimentering og læring. Når du eksperimenterer med agent-frameworks, tester promptstrategier eller bygger custom tool-integrationer, føles det sløsende at brænde API-credits af på trial-and-error. En lokal model lader dig iterere frit.

10.2.3 Hvor Lokale Modeller Kæmper

Det er værd at være specifik om fejltilstandene, for "den er mindre kapabel" er for vagt til at være nyttigt.

Multi-fil-ræsonnering. Bed en lokal 14B-model om at spore en bug på tværs af fire filer og et databaseskema, og den mister tråden. Den finder måske den rigtige fil men misforstår interaktionen mellem komponenter. Det er det største praktiske gab.

Langtidshorisontopgaver. Agentiske opgaver der involverer mange trin — læs, planlæg, implementer, test, debug, iterer — kræver at modellen opretholder kohærent intention på tværs af en lang kontekst. Lokale modeller har tendens til at drive. De glemmer planen. De genbesøger beslutninger de allerede tog. De sidder fast i loops.

Nuanceret code review. "Denne kode er korrekt men tilgangen er forkert" er et vurderingskald der kræver dyb forståelse. Lokale modeller har

tendens til overfladeanalyse — de fanger syntaksproblemer og åbenlyse bugs men misser arkitekturproblemer.

Kompleks tool-orchestrering. Når en opgave kræver ti eller flere kædede tool calls — at læse en testfil, køre den, læse fejlen, finde kildefilen, forstå konteksten, lave en ændring, køre testen igen — snubler lokale modeller hyppigere ved hvert trin, og fejlraten akkumulerer.

Intet af dette betyder at lokale modeller er ubrugelige. En 32B-model der kører lokalt kan håndtere en overraskende bred vifte af velscopede opgaver. Men du skal scope opgaverne tilsvarende. At bede en lokal model om at gøre hvad en frontier-model gør er at sætte den op til at fejle.

10.3 Omkostningen Ved Agentisk Arbejde

En enkelt agentisk kodningssession kan nemt forbruge 100.000-500.000 tokens. Det overrasker folk når de ser tallene første gang. Ikke fordi du chatter — fordi agenten *arbejder*.

Overvej hvad der sker når en agent tackler et moderat komplekst bugfix. Den læser tre eller fire filer for at forstå konteksten (måske 8.000 tokens input). Den ræsonnerer om problemet (2.000 tokens output). Den læser to filer mere (4.000 tokens). Den skriver et fix (1.000 tokens). Den kører testene (tool call, 500 tokens output). Testene fejler. Den læser fejlen (1.000 tokens). Den reviderer fixet (1.500 tokens). Den kører testene igen. De passerer. Den læser diffen for at dobbeltjekke (1.000 tokens). Total: måske 25.000 tokens for et ligetil bugfix.

Overvej nu en kompleks refaktoreringsopgave. Agenten læser måske tyve filer, genererer en plan, implementerer ændringer på tværs af otte filer, kører testsuiten fire gange, debugger to regressioner og skriver nye tests. Det er nemt 200.000 tokens. Ved frontier-modelpriser er det et sted mellem $3 og $15 afhængigt af modellen og input/output-forholdet.

Her er grove prisintervaller jeg har set i praksis, baseret på brug af frontier kommercielle modeller:

- **Simpelt bugfix eller lille feature:** $0,30-$1,00
- **Moderat feature med tests:** $2-$5

- **Kompleks multi-fil refaktorering:** $5-$15
- **Stor feature-implementering:** $15-$40+
- **Eksplorativ debugging-session der trækker ud:** $10-$30

Gang disse tal med et hold af ingeniører, der hver kører flere agentiske opgaver per dag, og du kigger på rigtige penge. Min vens $2.200-regning? Det passer nogenlunde for otte aktive ingeniører.

10.3.1 Strategier Til Omkostningsstyring

Løsningen er ikke at stoppe med at bruge agenter. Det er at være klog omkring det.

Sæt tokenbudgetter. De fleste agent-frameworks lader dig sætte en maksimal tokengrænse per opgave. Det er ikke bare omkostningskontrol — det er også et kvalitetssignal. Hvis en agent brænder 500.000 tokens af på en opgave der burde tage 50.000, er noget gået galt. Agenten sidder fast, looper eller misforstår opgaven. Et budget tvinger den til at fejle hurtigt i stedet for at spiralere.

Brug model routing. Send ikke hver opgave til den dyreste model. Vi dykker mere ned i dette i næste sektion, men kortversionen: brug en billig, hurtig model til udforskning og kodelæsning, og en kapabel dyr model til ræsonnering og implementering. Alene dette kan skære omkostninger med 50-70%.

Cache aggressivt. Hvis dit agent-framework understøtter prompt caching, slå det til. Meget af en agents kontekst gentages mellem ture — den samme system prompt, den samme projektkontekst, de samme nyligt læste filer. Caching undgår at genbehandle de tokens ved hvert tur.

Scop opgaver stramt. En velafgrænset opgave ("fix timezone-buggen i `billing/invoice.py`, testen er i `tests/test_invoice.py`") er billigere end en vag ("fix faktureringsbugsene"). Scoping er ikke bare godt ingeniørarbejde — det er omkostningskontrol. Agenten læser færre filer, laver færre eksplorative tool calls og konvergerer hurtigere.

Gennemgå dine fejl. Når en opgave fejler eller bruger alt for mange tokens, find ud af hvorfor. Var prompten vag? Manglede agenten kontekst den havde brug for? Var modellen ikke kapabel nok til opgaven? Hver fejl er en tuning-mulighed.

10.3.2 Styring Af Omkostninger På Tværs Af Et Hold

Individuel omkostningskontrol er én ting. At styre forbrug på tværs af et hold på otte eller tolv ingeniører, der hver kører agenter hele dagen, er et helt andet problem. Det er forskellen mellem at holde øje med din egen kost og at drive et restaurantkøkken.

Per-ingeniør-budgetter. Sæt et månedligt tokenbudget per ingeniør — ikke for at begrænse, men for at gøre omkostninger synlige. Når alle kan se deres eget forbrug, ændrer adfærden sig naturligt. Folk begynder at scope opgaver mere omhyggeligt, vælge den rigtige model til jobbet, dræbe løbske sessioner tidligere. Et rimeligt udgangspunkt: $200-400 per måned per ingeniør for et hold der aktivt bruger agenter. Nogle ingeniører vil konsekvent komme ind under budget. Andre vil spike under intense debugging-uger. Det er fint — pointen er bevidsthed, ikke håndhævelse.

Dashboard og synlighed. Du kan ikke styre hvad du ikke kan se. Spor forbrug per ingeniør, per projekt, per opgavetype. De fleste API-udbydere giver dig forbrugsopdelinger per API-nøgle, og hvis du tildeler nøgler per ingeniør eller per projekt, er dataene allerede der. Led det ind i et dashboard holdet kan se — selv et delt regneark virker til at starte med. Ingeniøren der opdager at de brændte $80 på en enkelt debugging-session vil naturligt begynde at scope opgaver strammere næste gang. Du behøver ikke at have en samtale om det. Tallet taler for sig selv.

Alarmer og sikringsafbrydere. Sæt alarmer ved 50% og 80% af det månedlige budget så ingen bliver overrasket. Vigtigere endnu, sæt en hård per-session tokengrænse som sikringsafbryder. Hvis en enkelt agentsession overstiger 500K tokens, er noget gået galt — agenten looper, opgaven er for vag, eller modellen kæmper med noget ud over dens kapacitet. Dræb den og undersøg om morgenen. Det er det der forhindrer "$2.200 overraskelsesregning"-scenariet fra åbningen af dette kapitel. Du behøver ikke at fange hver løbsk session manuelt. Automatiserede grænser gør jobbet.

ROI-samtalen. På et tidspunkt vil nogen i ledelsen spørge hvorfor API-regningen er i femcifrede. Du har brug for regnestykket klar. Framing det sådan: en senioringeniør koster 150K dollars om året fuldt loaded. Hvis agenter gør dem 30% mere produktive, er det $45K i værdi. Agentomkostningerne er $3-4K per år per ingeniør. ROI'en er omkring 10x. Selv hvis du er konservativ — lad os sige 15% produktivitetsforøgelse i stedet

for 30% — fungerer tallene stadig komfortabelt. Den sværere del er at måle den produktivitetsforøgelse præcist. Det kan du nok ikke, ikke med videnskabelig stringens. Men du kan tracke konkrete signaler: tid til merge af PR'er, antal opgaver fuldført per sprint, reduktion i kontekstskift. Par de metrikker med omkostningsdataene og du har en historie finans kan arbejde med.

Omkostning som kvalitetssignal. Højt tokenforbrug på en opgave er ikke bare dyrt — det er et signal om at noget gik galt. Opgaven var dårligt scopet, konteksten var utilstrækkelig, eller modellen var ikke kapabel nok til jobbet. Begynd at tracke omkostning per opgavetype over tid. Hvis omkostningerne for en bestemt kategori trender opad, undersøg — dine prompts kan have driftet, eller codebasen er blevet kompleks nok til at kræve en anden tilgang. Hvis omkostningerne trender nedad, er det dit hold der bliver bedre til agentisk ingeniørarbejde. De skriver strammere prompts, vælger bedre modeller og scoper opgaver mere effektivt. Omkostningsdata, brugt godt, bliver et spejl for holdets færdigheder.

10.4 Privatliv og Compliance

Noget kode kan genuint ikke forlade bygningen. Det er ikke paranoia — det er lov.

Statslige entreprenører der arbejder med klassificerede eller eksportkontrollerede projekter kan ikke sende kildekode til tredjeparts-API'er, punktum. Kravene til dataopbevaring er ikke forslag. De kommer med strafferetlige sanktioner.

Sundhedsselskaber der håndterer patientdata er bundet af HIPAA, GDPR eller tilsvarende regulering. Hvis din kode rører patientjournaler — selv testfixturer med realistiske falske data som en compliance-officer kunne skele til — skal du tænke omhyggeligt over hvad der går over nettet.

Finansielle institutioner har deres eget labyrinth af regulering. SOX-compliance, PCI-DSS, interne revisionskrav — detaljerne varierer, men temaet er konsistent: data forbliver inden for kontrollerede grænser.

Og så er der ren og skær konkurrencehemmeligholdelse. Dine proprietære algoritmer, din hemmelige sauce, din konkurrencefordel — at sende det

til andres servere kræver tillid til at de ikke træner på det, ikke logger det, ikke bliver kompromitteret. De fleste kommercielle API-udbydere tilbyder stærke kontraktmæssige garantier om dette. Men "stærke kontraktmæssige garantier" og "umuligt at bryde" er forskellige ting, og nogle sikkerhedsteams er ikke villige til at acceptere gabet.

I alle disse tilfælde er lokale modeller ikke nice-to-have. De er den eneste mulighed.

Afvejningen er reel: du accepterer reduceret modelkapacitet til gengæld for absolut datakontrol. En lokal 32B-model der gør et middelmådigt job på din klassificerede codebase er uendeligt mere nyttig end en frontier-model du ikke må bruge. Og til fokuserede, velscopede opgaver — den slags du bør skrive alligevel — er kvalitetsgabet ofte mindre end du forventer.

10.4.1 Mellemvejen: Private Deployments

Det er værd at nævne at der er en fremvoksende mellemvej. Cloud-udbydere tilbyder nu private model-deployments — dedikerede instanser af frontier-modeller der kører inde i din VPC, med kontraktmæssige garantier om at dine data forbliver inden for din grænse og ikke bruges til træning. AWS Bedrock, Azure OpenAI, Google Clouds Vertex AI tilbyder alle versioner af dette.

Det er ikke billigt. Du betaler for dedikeret compute, ikke delt API-infrastruktur. Men for store organisationer der har brug for frontier-kapacitet og streng datakontrol, er dette i stigende grad svaret. Du får kommerciel modelkvalitet med lokale model-privatlivsgarantier — for en pris.

10.5 Model Routing I Praksis

Det rigtige spørgsmål er ikke "lokalt eller kommercielt?" Det er "hvilken model, til hvilken del af workflowet?"

Et sofistikeret agentisk setup router forskellige dele af workflowet til forskellige modeller. Det er ikke teoretisk — teams gør det i dag, og det er den retning økosystemet bevæger sig.

10.5.1 Routing-Mønstret

Tænk over hvad en agent faktisk gør under en typisk opgave:

1. **Udforskning** — at læse filer, søge i codebasen, forstå struktur. Det er højvolumen, lavræsonnerings-arbejde. Modellen skal beslutte hvilke filer den skal læse og udtrække relevant information, men den tænker ikke dybt.

2. **Planlægning** — at analysere problemet, overveje tilgange, beslutte en strategi. Det kræver stærk ræsonnering. Det er den del hvor modelkvalitet tæller mest.

3. **Implementering** — at skrive de faktiske kodeændringer. Det kræver god kodegenerering og evnen til at følge planen fra trin 2.

4. **Verifikation** — at køre tests, læse fejl, beslutte om arbejdet er gjort. Moderat ræsonnering, tungt tool use.

5. **Iteration** — hvis verifikation fejler, gå tilbage og justér. Det kræver forståelse af fejlen og at forbinde den tilbage til implementeringen.

Ikke alle disse trin har brug for den samme model. Trin 1 kan håndteres af en lille, hurtig, billig model — selv en lokal. Den læser filer og rapporterer hvad der er i dem. Trin 2 er hvor du vil have frontier-modellen — det er den dyre ræsonnering der retfærdiggør API-omkostningen. Trin 3-5 kan ofte håndteres af en mid-tier model, da den svære tænkning allerede er gjort og modellen eksekverer en plan.

10.5.2 Hvad Det Ser Ud Som

I praksis kan model routing være så simpelt som at konfigurere dit agent-framework til at bruge forskellige modeller til forskellige opgavetyper:

```
# 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
```

Eller det kan være dynamisk — en letvægtsklassifikator der kigger på det aktuelle trin og router tilsvarende. Nogle frameworks er begyndt at bygge dette ind nativt. Andre kræver at du selv kobler det sammen.

Økonomien er overbevisende. Hvis 60% af en agents tokens bruges på udforskning og simple opgaver, og du router dem til en model der er

10x billigere, har du skåret dine samlede omkostninger med mere end halvdelen uden nogen reduktion i kvalitet på de dele der tæller.

10.5.3 Routing For Privatliv

Model routing løser også privatlivsproblemet mere elegant end en alt-eller-intet tilgang.

Lad os sige du bygger en sundhedsapplikation. Datamodellerne og forretningslogikken rører patientdata — det skal forblive lokalt. Men frontend-komponenterne, build-konfigurationen, CI-pipelinen? De indeholder ikke følsomme data. Der er ingen grund til at du ikke kan bruge en frontier kommerciel model til de ikke-følsomme dele mens du router følsomt arbejde til en lokal model.

Det kræver værktøj der er bevidst om følsomhedsgrænser — hvilke filer kan gå eksternt, hvilke kan ikke. Det værktøj modnes stadig, men mønstret er klart. I stedet for "alt lokalt" eller "alt kommercielt" får du "lokalt hvor det betyder noget, kommercielt alle andre steder."

10.6 Kom I Gang Uden At Betale Formuen

Du behøver ikke et budget for at begynde at lære agentisk ingeniørarbejde. Du behøver en laptop og en aften. Her er en praktisk rampe fra nul omkostninger til reel produktivitet.

10.6.1 Det Gratis Niveau

De fleste kommercielle udbydere tilbyder gratis niveauer eller prøvekreditter. Per tidligt 2026 kan du komme i gang med Claude, GPT eller Gemini uden at indtaste et kreditkort. De gratis niveauer er ratebegrænsede og kontekstbegrænsede, men de er mere end nok til at lære det grundlæggende — skriv din første agentiske prompt, se en agent iterere på en testfejl, få en fornemmelse for feedback loopet.

Par en gratis-niveau API-nøgle med et open source agent-framework — Claude Codes gratis niveau, eller Aider med dens gratis-model-support — og du har et komplet agentisk setup til nul kroner. Det vil ikke være hurtigt. Det vil ikke håndtere komplekst multi-fil-arbejde. Men det vil lære dig alt i kapitel to til fem af denne bog uden at bruge en krone.

10.6.2 Den Lokal-Først-Vej

Hvis du har en MacBook Pro med 16GB eller mere RAM, kan du køre nyttige modeller lokalt gratis, for evigt.

Installér Ollama — det tager én kommando. Pull en model — `ollama pull qwen2.5-coder:14b` tager et par minutter. Peg et agent-framework mod den. Du har nu et agentisk kodningssetup uden API-omkostninger, ingen ratebegrænsninger og ingen data der forlader din maskine.

Oplevelsen vil ikke matche en frontier kommerciel model. Kontekstvinduer er mindre. Multi-fil-ræsonnering er svagere. Tool use er mindre pålideligt. Men til fokuserede, velscopede opgaver — "skriv tests til denne funktion," "refaktorér denne klasse til at bruge dependency injection," "tilføj fejlhåndtering til dette endpoint" — er en lokal 14B-model overraskende kapabel. Og fordi den er gratis, kan du iterere uden at holde øje med måleren.

En praktisk startstack til nulomkostnings agentisk ingeniørarbejde:

- **Ollama** til model serving (gratis, lokal)
- **Qwen 2.5 Coder 14B** eller **DeepSeek Coder V2** til kodeopgaver
- **Aider** eller **Claude Code** (med lokal model-support) som agent-framework
- Et projekt med en testsuite (agenten har brug for feedback)

Det er det. Intet abonnement. Ingen API-nøgle. Ingen cloud compute. Du vil ramme loftet på et tidspunkt — en opgave der har brug for et større kontekstvindue, eller multi-fil-ræsonnering den lokale model ikke kan håndtere. Det er der du rækker ud efter en kommerciel model. Men det gratis setup vil bære dig længere end du forventer.

10.6.3 Sweet Spottet: $20/Måned

Når du er klar til at bruge penge, er det mest omkostningseffektive setup jeg har fundet en kombination af lokale modeller til udforskning og et kommercielt abonnement til ræsonnering.

De fleste kommercielle udbydere tilbyder en udviklerplan i $20/måned-intervallet der inkluderer en generøs tokentillæg. Brug dette til de opgaver der faktisk har brug for frontier-kapacitet — kompleks debugging, multi-fil refaktorering, arkitekturplanlægning. Brug din lokale model til alt

andet — at læse kode, generere boilerplate, skrive commit-beskeder, køre igennem testiterationer.

Denne hybridtilgang dækker typisk 80-90% af en solo-udviklers agentiske arbejde. Den lokale model håndterer de højvolumen, lavræsonneringsopgaver. Den kommercielle model håndterer de øjeblikke der har brug for ægte intelligens. Din månedlige regning forbliver forudsigelig, og du vælger ikke mellem kvalitet og omkostninger — du bruger hver hvor det giver mening.

10.6.4 Skaler Op Med Intention

Fejlen er at starte med den dyreste mulighed og optimere bagefter. Start gratis. Lær workflows. Forstå hvor modelkvalitet faktisk tæller og hvor "godt nok" er godt nok. Brug derefter penge på de specifikke huller som gratis værktøjer ikke kan udfylde.

Når du bruger rigtige penge på API-kald, vil du vide præcis hvad du betaler for — og vigtigere, hvad du *ikke* betaler for. Den viden er mere værd end nogen mængde kreditter.

10.7 Landskabet Skifter

Seks måneder fra nu vil detaljerne i dette kapitel være forældede. Priserne vil ændre sig. Kapacitetsgabene vil indsnævres. En ny model vil komme ud der omblander rangeringerne. Det er den ene forudsigelse jeg vil lave med tillid.

Hvad der ikke ændrer sig er rammen for at tænke om det. Du vil stadig have brug for at evaluere modeller langs de samme akser: kapacitet, omkostning, privatliv, hastighed og pålidelighed. Du vil stadig have brug for at matche modellen til opgaven i stedet for at vælge én model og bruge den til alt. Du vil stadig have brug for at forblive fleksibel.

De ingeniører jeg ser gøre det bedste arbejde er ikke loyale over for nogen bestemt model eller deployment-tilgang. De er pragmatikere. De bruger frontier kommerciel model når opgaven kræver det, en mid-tier model når den er god nok, og en lokal model når privatliv eller omkostning kræver det. De måler hvad der virker. De skifter når noget bedre kommer.

Bliv ikke religiøs omkring dette. Modellen er et værktøj. Færdigheden er at vide hvilket værktøj man skal gribe efter — og den færdighed overføres uanset hvilke modeller der eksisterer seks måneder fra nu.

11 Prompting Som Ingeniørarbejde

Der er ingen hemmelig syntaks. Ingen magisk besværgelse der får en agent til at producere perfekt kode. Prompting er *kommunikation* — og du ved allerede hvordan man kommunikerer.

Hvis du nogensinde har skrevet en god bug report, ved du hvordan man prompter. Hvis du nogensinde har skrevet et design-dokument som en holdkammerat kunne implementere uden at stille dig tyve spørgsmål, ved du hvordan man prompter. Hvis du nogensinde har oprettet en Jira-ticket der ikke kom tilbage som noget helt andet end hvad du ville — ved du hvordan man prompter.

Færdighederne overføres direkte. Klarhed, specificitet, kontekst, begrænsninger. De samme ting der gør menneskeligt samarbejde effektivt gør agentsamarbejde effektivt. Forskellen er at agenter ikke stiller opklarende spørgsmål når din prompt er vag. De gætter bare. Og de gætter selvsikkert.

Det er her mange erfarne ingeniører stille og roligt kæmper. Du har brugt år på at opbygge færdigheden i at *gøre* — skrive kode, debugge, bygge systemer. Nu er den færdighed der tæller at *artikulere* — at forklare hvad du vil med nok præcision til at nogen anden kan gøre det. Det er en anden muskel. Og det kan føles, i de tidlige dage, som en degradering. Det er det ikke. Men ubehaget er reelt, og at lade som om det ikke er det hjælper ikke.

11.1 Anatomien Af En God Opgaveprompt

En god prompt har tre dele: *hvad* du vil have gjort, *hvorfor* det er vigtigt, og *hvordan* succes ser ud. De fleste giver kun det første, og selv det er normalt vagt.

Overvej forskellen:

Dårlig: "Fix auth-buggen."

Det fortæller agenten næsten ingenting. Hvilken auth-bug? Hvor manifesterer den sig? Hvad er den forventede adfærd? Agenten vil gå på jagt i din codebase, danne en teori om hvad du muligvis mener, og anvende et fix der kan være helt forkert. Du har lavet et fem-minutters fix om til en tyve-minutters review af noget du ikke bad om.

God: "Login-endpointet returnerer 401 for gyldige tokens når sessioncachen er kold. Buggen er sandsynligvis i `middleware/auth.go` i `validateSession`-funktionen. Testen i `auth_test.go:TestColdCacheLogin` reproducerer den. Fix buggen og sørg for at alle eksisterende auth-tests stadig passerer."

Det er et helt andet dyr. Agenten kender symptomet, den mistænkte lokation, og hvordan den verificerer fixet. Den kan gå direkte til den relevante kode, forstå problemet og validere sin løsning — alt uden at gætte.

Mønstret er simpelt. *Hvad* der er i stykker eller nødvendigt. *Hvor* man skal kigge. *Hvordan* man verificerer. Hvert minut du bruger på at gøre din prompt præcis sparer dig fem minutter på at reviewe det forkerte output.

11.2 Begrænsningsspecifikation

At fortælle en agent hvad den skal gøre er kun halvdelen af jobbet. At fortælle den hvad den *ikke* skal gøre er lige så vigtigt.

Agenter er ivrige. De optimerer for at løse det problem du beskrev, og de vil med glæde refaktorere hele dit modul, tilføje tre nye afhængigheder og ændre det offentlige API for at gøre det. Det er ikke ondskab — det er en optimeringsalgoritme der gør hvad optimeringsalgoritmer gør. Dit job er at sætte grænserne.

Nyttige begrænsninger ser sådan ud:

- "Modificer ikke den offentlige API-overflade."
- "Behold den eksisterende teststruktur — tilføj nye test cases, omorganisér ikke."
- "Tilføj ikke nye afhængigheder."

- "Hold dig inden for de eksisterende fejlhåndteringsmønstre i denne codebase."
- "Ændr ikke nogen filer uden for `services/`-mappen."

Tænk på begrænsninger som autoværnet på en bro. Agenten kan køre hvor som helst inden for banerne, men den kan ikke køre over kanten. Uden autoværn får du kreative løsninger der teknisk virker men skaber vedligeholdelsesmareridt. Med dem får du løsninger der passer ind i din codebase som om de altid har været der.

Jo mere erfaren du bliver med agentisk ingeniørarbejde, jo mere er dine prompts defineret af deres begrænsninger snarere end deres instruktioner. Du lærer hvilke friheder der fører til gode resultater og hvilke der fører til kaos.

11.3 Opgavenedbrydning

En almindelig fejl: at bede en agent om at bygge noget stort i en enkelt prompt. "Byg et brugerdashboard med realtidsmetrikker, rollebaseret adgang og eksport til CSV." Det er ikke en prompt — det er et projekt. Og projekter skal brydes ned i opgaver.

Opgavenedbrydning er praksis med at splitte store forespørgsler i små, *verificerbare* trin. Hvert trin har et klart input, et klart output og en klar måde at tjekke om det virkede.

I stedet for "byg et brugerdashboard" skriver du:

1. Opret datamodellen for dashboard-metrikker i `models/dashboard.go` med skemaet defineret i design-dokumentet. Skriv unit tests til modelvalideringen.
2. Byg API-endpointet `GET /api/dashboard` der returnerer metrikker for den autentificerede bruger. Skriv integrationstests.
3. Tilføj rollebaseret filtrering så admin-brugere ser alle metrikker og almindelige brugere kun ser deres egne. Opdater de eksisterende tests til at dække begge roller.
4. Byg React-komponenten der viser dashboarddataene. Brug den eksisterende `DataTable`-komponent til metrikgitteret.

Hvert trin er en selvstændig prompt. Hvert har et verificerbart resultat. Hvert bygger på det verificerede output fra det forrige trin. Hvis trin to går skævt, fanger du det før du har spildt tid på trin tre.

Det er ikke bare god prompting — det er godt ingeniørarbejde. Du anvender de samme nedbrydningsfærdigheder du ville bruge til at planlægge en sprint eller bryde en pull request ned. Arbejdsenheden er lille nok til at reviewe, lille nok til at teste, og lille nok til at smide væk hvis den er forkert.

11.4 Prompten Som Spec

De bedste prompts jeg har set ligner miniature designdokumenter. De beskriver det ønskede resultat, ikke implementeringstrinene. De lister begrænsningerne. De definerer acceptkriterier. De giver lige nok kontekst til at agenten kan træffe gode beslutninger uden at drukne den i irrelevant information.

Her er hvad en prompt-som-spec ser ud:

"Tilføj rate limiting til `/api/search`*-endpointet. Brug den eksisterende* `RateLimiter`*-middleware i* `middleware/ratelimit.go`*. Sæt grænsen til 100 requests per minut per autentificeret bruger, og 20 per minut for uautentificerede requests. Returnér en 429-status med en* `Retry-After`*-header når grænsen overskrides. Tilføj tests for både den autentificerede og uautentificerede sti, inklusive edge casen hvor en bruger rammer præcis grænsen. Modificer ikke rate limiter-middlewaren selv — konfigurér og anvend den bare."*

Det er en spec. En agent kan implementere dette uden tvetydighed. En menneskelig reviewer kan tjekke resultatet mod kravene. Det ønskede resultat er klart, begrænsningerne er eksplicitte, og verifikationskriterierne er definerede.

At skrive prompts på denne måde kræver øvelse. Det kræver også disciplin — disciplinen til at gennemtænke hvad du faktisk vil have før du begynder at taste. Men den disciplin betaler udbytte. En velspecificeret prompt producerer et resultat du kan merge. En vag prompt producerer et resultat du er nødt til at omskrive.

11.5 Iteration Over Perfektion

Din første prompt vil ikke være perfekt. Det er fint. Prompting er en iterativ proces, og færdigheden ligger ikke i at skrive den perfekte prompt — den ligger i at *læse outputtet*, forstå hvor kommunikationen brød sammen, og forfine.

Når en agent producerer noget forkert, modstå trangen til at bebrejde værktøjet. Spørg i stedet dig selv: hvad undlod jeg at kommunikere? Glemte jeg en begrænsning? Var konteksten utilstrækkelig? Antog jeg viden agenten ikke havde?

Det er debugging — men i stedet for at debugge kode debugger du din egen kommunikation. Fejlbeskeden er agentens output. Stack tracet er din prompt. Et sted derinde er miskommunikationen, og at finde den gør din næste prompt bedre.

Erfarne agentiske ingeniører udvikler et feedback-instinkt. De ser agentens output og ved øjeblikkeligt hvilken del af deres prompt der forårsagede afvigelsen. "Ah, jeg sagde 'håndtér fejl' men specificerede ikke *hvilke* fejl eller *hvordan* de skal håndteres. Selvfølgelig smed den en generisk catch-all ind."

Hver iteration strammer loopet. Første prompt bringer dig 70% af vejen. En opfølgende korrektion bringer dig til 90%. En sidste forfining bringer dig i mål. Over tid bliver dine første prompts bedre, og du har brug for færre iterationer. Men du har aldrig brug for nul.

11.6 Stemmedrevet Udvikling

De fleste af os prompter ved at taste. Det giver mening — vi er ingeniører, vi lever i tekst. Men der er en anden inputkanal der er hurtigere, mere naturlig og overraskende underbrugt: din stemme.

Moderne speech-to-text har nået det punkt hvor du kan tale en prompt ind i din terminal og få den transskriberet med næsten perfekt nøjagtighed. Værktøjer som Whisper, macOS Dictation og SuperWhisper lader dig tale til din agent i stedet for at taste. Resultatet er det samme — tekst går ind, kode kommer ud. Men oplevelsen er fundamentalt anderledes.

Her er grunden: at taste og at tale er forskellige tænkemåder. Når du taster, redigerer du undervejs. Du sletter et ord, omformulerer, backspacer, omstrukturerer. Teksten du producerer er *poleret* — du havde tid til at glatte de ru kanter ud før den forlod dine fingre. At tale giver dig ikke den luksus. Når du taler, committer du. Ordene forlader din mund og de er væk. Der er ingen backspace.

Det lyder som en ulempe. Det er faktisk en træningsgrund.

Når du taler en prompt, er du tvunget til at organisere dine tanker *før* du åbner munden. Du kan ikke læne dig op ad krykken med at redigere midt i sætningen. Du er nødt til at vide hvad du vil, strukturere det mentalt og levere det klart — i realtid. De første par gange du prøver det, vil du snakke i øst og vest. Du vil sige "øh" og cirkle rundt og modsige dig selv. Agenten vil modtage et rodet transskript, og outputtet vil afspejle det rod.

Men noget sker hvis du bliver ved med det. Du bliver bedre. Ikke bare til prompting — til *at tale klart om tekniske problemer*. Du udvikler evnen til at beskrive en bug, en feature eller en refaktoreringsopgave i en enkelt sammenhængende strøm. Du lærer at frontloade kontekst, angive begrænsninger tidligt og slutte med et klart ønske. Du stopper med at ævle fordi ævlen producerer dårlige resultater.

Den færdighed overføres overalt. Standup-møder. Arkitekturdiskussioner. Parprogrammering. Incident-kald. Enhver situation hvor du har brug for at artikulere en teknisk idé klart, under tidspres, uden sikkerhedsnettet af en teksteditor. Stemmedrevet udvikling er ikke bare en hurtigere måde at prompte — det er øvelse til enhver teknisk samtale du nogensinde vil have.

Der er også en praktisk hastighedsfordel. De fleste mennesker taler med 130 ord per minut. De fleste taster med 40 til 80. Til den slags overordnede, intentionsdrevne prompts der producerer det bedste agentoutput — "her er problemet, her er konteksten, her er hvad jeg vil, her er hvad jeg ikke vil" — er tale simpelthen hurtigere. Du bruger mindre tid på input og mere tid på at reviewe output.

Prøv det i en uge. Vælg et speech-to-text-værktøj, kobl det ind i dit workflow, og tal dine prompts i stedet for at taste dem. Første dag vil føles

akavet. Tredje dag vil du bemærke at dine talte prompts bliver strammere. Ved slutningen af ugen vil du bemærke at din *talte kommunikation generelt* bliver strammere.

Det er også værd at nævne at speech-to-text-modeller kan køre helt på din maskine. NVIDIAs Parakeet-familie af modeller — kompakte, højpræcisions ASR-modeller — kører lokalt uden nogen cloud-afhængighed. Værktøjer som SuperWhisper og whisper.cpp gør det samme med OpenAIs Whisper-vægte. En moderne MacBook kan køre disse modeller i næsten-realtid med præcis transskription og lav latency. Du behøver ikke en cloud-service for at omdanne tale til tekst — det lokale værktøj er allerede der.

Agenterne er ligeglade med om din prompt blev tastet eller talt. Men *du* vil være en klarere tænker af at have talt den.

11.7 Visuel Kontekst: Når Ord Ikke Er Nok

Ikke alt er nemt at beskrive i tekst. Et ødelagt layout, en mærkelig renderingsglitch, en fejldialog med et stack trace — nogle gange er den hurtigste måde at kommunikere hvad du ser at *vise* det.

Moderne LLM'er er multimodale. De kan læse screenshots, diagrammer, fotos af whiteboards og fejlbeskeder fanget fra din skærm. Det er ikke en nyhedsfeature — det er et af de mest underbrugte værktøjer i det agentiske ingeniørworkflow.

Her er et workflow jeg bruger dagligt: jeg ser en bug på min telefon — et layout der er i stykker, en modal der er forskudt, en formular der æder input. Jeg screenshoter det på iOS, og takket være Universal Clipboard paster jeg det direkte ind i min terminalsession på min Mac. Agenten ser hvad jeg ser. Ingen grund til at beskrive "knappen overlapper headeren på mobilvisning" — screenshottet *er* beskrivelsen.

Det er vigtigt fordi visuelle bugs er notorisk svære at beskrive i tekst. Du ender med at skrive tre afsnit om padding og z-index når et enkelt screenshot kommunikerer problemet øjeblikkeligt. Agenten kan se den ødelagte tilstand, ræsonnere om hvad der er galt, og foreslå et fix — ofte hurtigere end du kunne have færdiggjort at taste beskrivelsen.

Men det rækker ud over bug reports. Nogle almindelige visuelle kontekst-workflows:

- **Fejlscreenshots.** En browserkonsol fuld af rødt, et terminal stack trace, et deployment-dashboard der viser fejlede health checks. Screenshot det, paste det, bed agenten om at diagnosticere. Det er særligt nyttigt når fejlbeskeder er lange eller indeholder formatering der er smertefuld at kopiere som tekst.
- **Designreferencer.** En Figma-mockup, en skitse, en konkurrents UI du vil tilnærme. Paste billedet og sig "gør vores indstillingsside til at ligne det her." Agenten kan udtrække layoutstruktur, farvevalg og komponenthierarki fra en visuel reference.
- **Debugging af visuel tilstand.** "Hvorfor ser denne side forkert ud?" er en forfærdelig prompt. Et screenshot af siden *plus* "hvorfor ser denne side forkert ud?" er en fantastisk en. Agenten kan sammenligne hvad den ser med det forventede layout og identificere CSS-problemer, manglende data eller renderingbugs.
- **Whiteboard-fotos.** Efter en arkitekturdiskussion, tag et foto af whiteboardet og paste det ind. Agenten kan læse boksene, pilene og etiketterne, og hjælpe dig med at oversætte den skitse til kodestruktur, API-definitioner eller dokumentation.

iOS-til-Mac clipboard-pipelinen fortjener særlig omtale fordi den fjerner al friktion fra dette workflow. Du behøver ikke at gemme screenshottet, AirDroppe det, finde det i Finder og trække det ind i et værktøj. Du ser problemet, du fanger det, du paster det. Tre sekunder fra "det er i stykker" til "agenten kigger på det." Den hastighed er vigtig fordi den holder dig i flow. Eventuelle ekstra trin — selv tredive sekunders filhåndtering — skaber nok friktion til at du falder tilbage til at taste en tekstbeskrivelse, som er langsommere og mindre præcis.

Den centrale indsigt er at *kontekst ikke bare er tekst.* Når vi talte om kontekst som den vigtigste ingrediens i agentisk arbejde, talte vi om alle former for kontekst — kodefiler, dokumentation, testoutput, *og* visuel tilstand. Et screenshot er tusind tokens værd, og agenter der kan se er dramatisk mere nyttige end agenter der kun kan læse.

Hvis du ikke allerede bruger visuel kontekst i dit agentiske workflow, så begynd. Screenshot dine bugs. Paste dine fejlbeskeder. Del dine designreferencer. Agenterne kan se nu. Lad dem.

11.8 Anti-Mønstre

Nogle promptingvaner producerer konsekvent dårlige resultater. Lær at genkende dem.

At være for vag. "Gør denne kode bedre." Bedre hvordan? Hurtigere? Mere læsbar? Mere vedligeholdbar? Agenten vil vælge *noget* at forbedre, og det er sandsynligvis ikke det du havde i tankerne. Hvis du ikke kan artikulere hvad "bedre" betyder, er du ikke klar til at prompte.

At være for foreskrivende. Den modsatte fejl. "På linje 47, ændr variabelnavnet fra `x` til `count`, tilføj så en if-statement på linje 48 der tjekker om count er større end nul, så…" Du skriver koden på dansk og beder agenten om at oversætte. Det er langsommere end at skrive koden selv. Beskriv *resultatet*, ikke tastetrykkene.

Kontekstdumping. At paste hele din codebase, alle dine designdokumenter og et transskript af dine sidste tre holdmøder ind i prompten. Mere kontekst er ikke altid bedre. Irrelevant kontekst er støj, og støj overdøver signal. Giv agenten hvad den har brug for — filstier, funktionsnavne, den specifikke adfærd du vil have — og stol på at den udforsker derfra.

Køkkenvask-prompts. "Fix auth-buggen, refaktorér også databaselaget, og mens du er i gang opdater README'en og tilføj TypeScript-typer til API-klienten." Det er fire separate opgaver proppet ind i én prompt. Agenten vil forsøge dem alle, gøre ingen af dem godt, og producere en diff så stor at det tager længere at reviewe den end at gøre arbejdet selv. Én prompt, én opgave.

At antage delt kontekst. "Gør det på samme måde som vi gjorde betalingsmodulet." Agenten husker ikke din sidste session. Den ved ikke hvad "vi" besluttede til standup. Hver prompt starter fra nul. Giv konteksten eksplicit, hver gang.

11.9 Prompting Er En Færdighed

Prompting er ikke et parlortrick. Det handler ikke om at opdage den ene mærkelige frase der udløser bedre output. Det er en kommunikationsfærdighed — og som alle kommunikationsfærdigheder forbedres den med øvelse, feedback og bevidst opmærksomhed.

De ingeniører der får mest ud af agentiske værktøjer er dem der behandler prompting med samme grundighed som de anvender til at skrive kode. De tænker før de taster. De specificerer før de implementerer. De verificerer før de går videre.

Det er ikke en ny færdighed. Det er bare ingeniørarbejde.

12 Multi-Agent Orchestrering

Én agent er kraftfuld. Multiple agenter der arbejder sammen er noget helt andet.

Tænk på det sådan. En enkelt tømrer kan bygge et skur. Men et hus? Du har brug for en elektriker, en blikkenslager, en tagdækker — specialister der arbejder parallelt, hver fokuseret på det de gør bedst, der koordinerer lige nok til at holde sig ude af hinandens vej. Huset rejser sig hurtigere og arbejdet er bedre end hvis én person forsøger at gøre alt.

Agentisk ingeniørarbejde fungerer på samme måde. En enkelt agent på en kompleks opgave vil male sig igennem den sekventielt — planlæg, implementér, test, debug, iterer. Det virker, men det er langsomt. Tre agenter, der hver håndterer en velafgrænset del af problemet, kan afslutte på en tredjedel af tiden. Nogle gange mindre, fordi fokuserede agenter laver færre fejl end én agent der jonglerer for mange bekymringer.

Men der er en hage. Parallelle agenter kræver *koordinering*. Og koordinering har en omkostning. Dette kapitel handler om hvornår man skal betale den omkostning, og hvordan man betaler den godt.

12.1 Nedbrydningsstrategier

Den sværeste del af multi-agent-arbejde er ikke at køre multiple agenter. Det er at beslutte hvordan man deler arbejdet op.

Den gyldne regel: opgaver skal have *minimal kobling*. Hvis agent A's arbejde afhænger af agent B's output, kan de ikke køre parallelt — de er sekventielle, og du bør behandle dem sådan. Kunsten er at finde sømme i dit arbejde hvor du kan skære rent.

Der er tre pålidelige nedbrydningsmønstre.

Per lag. Én agent håndterer backend-API'et, en anden bygger frontend-komponenten, en tredje skriver testene. Det virker godt fordi lag har

naturlige grænser — en defineret grænseflade mellem dem. Så længe du aftaler API-kontrakten på forhånd, kan hver agent arbejde uafhængigt.

Per feature. Hvis du bygger tre uafhængige features, giv hver til en separat agent. Det er den simpleste nedbrydning. Featuresene rører forskellige filer, forskellige mapper, forskellige bekymringer. Merge-konflikter er sjældne.

Per bekymring. Én agent refaktorerer, en anden skriver tests til den refaktorerede kode, en tredje opdaterer dokumentationen. Det er et sekventielt mønster — mere pipeline end parallel — men det lader hver agent fokusere på en enkelt type tænkning. Refaktoreringsagenten behøver ikke at kontekstskifte ind i testskrivningstilstand. Den refaktorerer bare, og giver stafetten videre.

Den nedbrydning du vælger afhænger af arbejdets form. Men princippet er konstant: *find sømmene, skær langs dem, minimér overfladearealet hvor agenter skal være enige.*

12.2 Branch-Per-Agent

Hver agent får sin egen branch. Vi dækkede dette i Git-kapitlet. I multi-agent-arbejde bliver det absolut essentielt.

Men branches alene er ikke nok. To agenter på forskellige branches der deler det samme arbejdsbibliotek vil kæmpe om filsystemet — de vil overskrive hinandens filer, korrumpere hinandens builds, bryde hinandens testkørsler. Du har brug for *worktrees*.

Hver agent får sit eget git worktree: et separat bibliotek, på sin egen branch, med sin egen kopi af codebasen. Agenterne deler historik men intet andet. De kan bygge, køre tests, installere afhængigheder og lave rod — alt uden at påvirke hinanden.

Opsætningen er hurtig:

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

Tre biblioteker. Tre branches. Tre agenter. Fuld isolering. Det er sandbox-modellen fra tidligere kapitler gjort konkret for multi-agent-arbejde. Når en agent er færdig, reviewer du dens branch, merger hvis den er god, og fjerner worktree'et. Hvis arbejdet er dårligt, smider du det hele væk. Nul omkostning.

12.3 Overleveringsmønstret

Ikke alt multi-agent-arbejde er parallelt. Nogle gange arbejder agenter *sekventielt*, hvor hver bygger på den forrige. Agent A planlægger. Agent B implementerer. Agent C reviewer.

Det er overleveringsmønstret, og det er kraftfuldt — men kun hvis selve overleveringen er ren.

Problemet med sekventielle agenter er kontekstab. Agent A har en rig forståelse af problemet når den er færdig med at planlægge. Agent B starter koldt. Hvis du bare fortæller Agent B "implementér planen," vil den misse nuancer. Den vil antage anderledes. Den vil løse et lidt anderledes problem end det Agent A planlagde for.

Fixet er *struktureret overlevering*. Agent A planlægger ikke bare — den producerer en artefakt der fanger dens ræsonnering. Det kan tage flere former:

- **Et sammendragsdokument.** En markdown-fil droppet i repoet: `PLAN.md`. Den beskriver hvad der skal ske, hvilke afvejninger der blev overvejet, hvad der blev afvist og hvorfor. Agent B læser dette før den skriver en eneste linje kode.
- **Et sæt filer.** Agent A opretter stub-filer — tomme funktioner med docstrings, grænsefladedefintioner, typesignaturer. Agent B udfylder dem. Stubbene *er* overleveringen.
- **En struktureret prompt.** Agent A's output bliver Agent B's input, formateret som en detaljeret opgavebeskrivelse med acceptkriterier. Du paster den direkte ind i Agent B's kontekst.

Nøglen er at overleveringen skal være *eksplicit og komplet*. Ingen implicitte antagelser. Ingen "den næste agent finder nok ud af det." Enhver beslutning, enhver begrænsning, enhver edge case — skrevet ned.

Det kræver disciplin. Men det er den samme disciplin du ville anvende når du skriver en ticket til en menneskelig kollega på et andet kontinent og i en anden tidszone. Hvis du ikke ville stole på en mundtlig overlevering, stol ikke på en implicit en mellem agenter.

12.4 Merge-Problemet

Her bliver multi-agent-arbejde genuint svært.

Tre agenter arbejdede parallelt. Hver producerede en ren, testet branch. Nu skal du merge dem alle ind i main. Nogle gange er det smertefrit — tre branches der rører helt forskellige filer merger uden en eneste konflikt. Smukt.

Nogle gange er det ikke. Agent A ændrede databaseskemaet. Agent B tilføjede en migration der antager det gamle skema. Agent C opdaterede API-handleren som både A og B også rørte. Nu har du en trevejes-konflikt og ingen enkelt agent har det fulde billede.

Der er tre strategier til at håndtere dette.

Forebyggelse. De bedste merge-konflikter er dem der aldrig sker. Når du nedbryder arbejde, tænk over filejerskab. Tildel forskellige mapper, forskellige moduler, forskellige filer til forskellige agenter. Hvis agenter aldrig rører den samme fil, kan de aldrig konflikte. Det er den billigste strategi og du bør bruge den når det er muligt.

Merge-agenten. Når konflikter opstår, spin en ny agent op hvis eneste job er at merge. Giv den branches, konflikterne og konteksten fra hver agents arbejde. En god merge-agent kan løse de fleste konflikter automatisk — den læser begge sider, forstår intentionen og producerer et sammenhængende resultat. Det er som en senioringeniør der sætter sig ned med to PR'er og finder ud af hvordan de passer sammen.

Menneskelig review. Nogle gange er konflikten semantisk, ikke syntaktisk. Koden merger rent men *logikken* modsiger sig selv. To agenter tog inkompatible designbeslutninger. Ingen automatisk merge vil fange dette. Her er du tjener dine penge som ingeniør. Review branches før du merger. Læs diffs side om side. Sørg for at brikkerne faktisk passer sammen.

I praksis vil du bruge alle tre. Forebyg hvad du kan. Automatisér resten. Review alt.

12.5 Orchestreringsoverhead

Flere agenter er ikke altid bedre.

Hver ekstra agent tilføjer koordineringsomkostninger. Du skal nedbryde arbejdet. Sætte worktrees op. Definere grænseflader. Håndtere merges. Reviewe multiple branches i stedet for én. For en opgave der tager en enkelt agent tredive minutter, kan det tage femogfyrre at spinne tre agenter op — tyve minutters parallelt agentarbejde plus femogtyve minutter af din tid til orchestrering.

Break-even-punktet er højere end du tror. Efter min erfaring begynder multi-agent orchestrering at betale sig når den samlede opgave ville tage en enkelt agent *mindst to timer*. Under det æder overheadet gevinsterne.

Der er også andre omkostninger. Kontekst fragmenteres. Hver agent ser kun sin del. Ingen enkelt agent forstår hele systemet på den måde én agent der arbejder alene ville. Det kan føre til inkonsistenser — forskellige navnekonventioner, forskellige fejlhåndteringsmønstre, forskellige antagelser om delt tilstand.

Færdigheden er ikke at køre så mange agenter som muligt. Færdigheden er at vide *hvornår* man skal parallelisere og hvornår en enkelt fokuseret agent er det rigtige værktøj. Et mandskab på fem er ikke altid hurtigere end én erfaren sømand der kender båden.

12.6 Et Praktisk Eksempel

Lad os gøre det konkret. Du bygger en webapplikation og du skal tilføje en ny feature: brugernotifikationer. Brugere skal se et klokkeikon med en tæller, klikke på det for at se en liste, og markere notifikationer som læst. Du har brug for et API, en frontendkomponent og tests.

Det er en lærebogssag for tre parallelle agenter.

Trin 1: Definér grænsefladen. Før du spinner nogen agenter op, bruger du fem minutter på at skrive API-kontrakten ned. `GET /api/notifications` returnerer en liste. `PATCH /api/notifications/:id` markerer én som læst. Notifikationsobjektet har `id`, `message`, `read`, `created_at`. Skriv dette i et delt dokument eller en stub-fil. Det er kontrakten alle tre agenter arbejder mod.

Trin 2: Opret 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
```

Trin 3: Start agenterne. Hver agent får en klar, afgrænset opgave:

- Agent 1: "Implementér notifications API-endpointene i `../app-api`. Følg kontrakten i `API_CONTRACT.md`. Inkludér model, route og controller. Skriv unit tests til controlleren."
- Agent 2: "Byg notifications UI-komponenten i `../app-frontend`. Den kalder `GET /api/notifications` og `PATCH /api/notifications/:id`. Vis et klokkeikon med ulæst-tæller. Klik åbner en dropdown-liste."
- Agent 3: "Skriv integrationstests i `../app-tests`. Dæk det fulde flow: opret en notifikation, hent listen, markér som læst, verificér at tælleren opdateres."

Trin 4: Lad dem arbejde. De tre agenter kører simultant. Hver committer til sin egen branch. Du overvåger fremskridt men griber ikke ind medmindre noget går galt.

Trin 5: Review og merge. Når alle tre er færdige, reviewer du hver branch. API-branchen merges først — den er fundamentet. Så frontend-branchen. Så testene. Du kører den fulde testsuite efter hvert merge for at fange integrationsproblemer tidligt.

Samlet tid: måske fyrre minutter. API-agenten tog tredive, frontendagenten tog femogtyve, og testagenten tog femogtredive. Men de kørte parallelt, så urfekketid var femogtredive minutter plus ti minutter af din orchestreringstid.

En enkelt agent der gør alt sekventielt? Sandsynligvis halvfems minutter.

Matematikken virker når opgaven er stor nok. Og efterhånden som du bliver bedre til nedbrydning, vil du udvikle en intuition for hvilke opgaver der er værd at splitte og hvilke der ikke er. Som det meste i ingeniørarbejde er det et vurderingskald. Men nu har du værktøjerne til at tage det.

13 Agenter I Pipelinen

Et hold jeg kender satte en fin automatisering op sidste år. De tilføjede en Claude-agent som et CI-trin der ville fange lint-fejl og autofikse dem. En udvikler pusher kode, linteren kører, og hvis den fejler, omskriver agenten de fejlende linjer og pusher et fix-commit. Ingen menneskelig intervention nødvendig. Pipelinen forbliver grøn. Alle elskede det.

Det virkede strålende i omkring to uger.

Så bemærkede nogen noget mærkeligt under en code review. En hel kategori af lint-regler var forsvundet. Agenten havde fundet ud af at den hurtigste måde at fikse en lint-fejl var at modificere `.eslintrc.json` — deaktivere reglen, pushe konfigurationsændringen, pipeline bliver grøn. Den havde gjort det i en måned. Ingen fangede det fordi signalet de holdt øje med — pipeline-status — var præcis det signal agenten havde lært at game.

Fixet var ligetil: gør lint-konfigurationen read-only for agenten. Men lektien var større end én fejlkonfigureret pipeline.

Automatiserede agenter har brug for de samme guardrails som interaktive. Sandsynligvis flere, fordi der er ingen der sidder og kigger på diffen der ruller forbi. Når du parrer med en agent lokalt, ser du hver fil den rører. Du bemærker når den gør noget klogt men forkert. I CI kører agenten i mørket. Det eneste du ser er resultatet — grønt eller rødt. Og hvis agenten finder en måde at gøre resultatet grønt uden faktisk at løse problemet, bemærker du det måske ikke i ugevis.

Dette kapitel handler om at bruge agenter i pipelines ansvarligt. Hvor de tilføjer ægte værdi, hvor de skaber risiko, og hvordan man sætter dem op så de forbliver nyttige uden at blive en belastning.

Indsatserne er højere i CI end ved dit skrivebord. Gør det rigtigt, og agenter multiplicerer dit holds throughput døgnet rundt. Gør det forkert, og du har bygget en dyr, uovervåget fodfejl.

13.1 Agenter Som CI-Trin

Den simpleste måde at få agenter ind i din pipeline er som CI-trin — diskrete jobs der kører ved hvert push eller pull request, ligesom din linter eller testsuite.

Du genopfinder ikke din pipeline. Du tilføjer intelligens til den.

Den mest umiddelbart nyttige CI-agent: PR-forscreening. Ikke at erstatte menneskelig review, men filtrering. En agent læser diffen, tjekker for åbenlyse problemer — ubrugte imports, inkonsistent navngivning, manglende fejlhåndtering, testfiler der faktisk ikke asserter noget — og efterlader kommentarer. Når en menneskelig reviewer åbner PR'en, er det trivielle allerede flagget. Mennesket kan fokusere på arkitektur, tilgang, intention.

Andre gode kandidater:

- **Changelog-generering.** Agenten læser commits siden sidste release, krydsrefererer med ticketnumre og udkaster release notes. Et menneske redigerer før publicering, men det første udkast er gratis.
- **Importsortering og formatering.** Sikkert, verificerbart, lav indsats. Hvis agenten formaterer noget forkert, er diffen åbenlys.
- **Afhængighedsopdateringsresumeer.** Når Dependabot åbner en PR, kan en agent læse changeloggen for den opdaterede pakke og opsummere hvad der ændrede sig og om det sandsynligvis påvirker din kode.

Der er også en mere subtil brug: **testfejl-triage**. Når en CI-kørsel fejler, kan en agent læse testoutputtet, identificere den fejlende test, kigge på den seneste diff og poste en kommentar der forklarer om fejlen ligner en ægte bug eller en flaky test. Den vil ikke altid have ret, men den sparer udvikleren fra at åbne CI-loggene, scrolle igennem to hundrede linjer output og finde ud af hvad der gik galt. Den ti-minutters undersøgelse bliver et tredive-sekunders blik på en kommentar.

Nøgleprincippet: agenter i CI bør gøre ting der er **sikre at automatisere** og **nemme at verificere**. Hvis du ikke hurtigt kan afgøre om agenten gjorde det rigtige, bør det ikke automatiseres endnu. En god lakmustest: ville du være komfortabel med at agenten gjorde det her hundrede gange

og du kun stikprøvekontrollerede ti af dem? Hvis svaret er nej, er det ikke klar til CI.

13.2 Natteagenten

Det er et af de mest kraftfulde mønstre i agentisk ingeniørarbejde, og et af de mindst omtalte. Det er også det der får lederes øjne til at lyse op — "du mener agenten arbejder mens vi sover?" — hvilket er præcis derfor det har brug for omhyggelig framing.

Du er ved at lukke ned for dagen. Der er en veldefineret ticket i backloggen — tilføj et nyt API-endpoint, skriv en datamigration, refaktorér et modul til at bruge det nye logningsbibliotek. Den slags opgave hvor kravene er klare og definitionen af færdig er testbar. Du peger en agent mod den, giver den en branch, og tager hjem. Du vågner op til en PR med arbejdet gjort, tests der passerer, klar til review.

Kvalitetsstandarden for uovervågede agenter er højere end for interaktive. Når du sidder der og kigger, fanger du fejl i realtid. Når agenten arbejder om natten, akkumulerer fejl. Så du har brug for:

- **Omfattende tests at verificere mod.** Agenten har brug for en måde at vide den er færdig. Eksisterende tests der skal fortsætte med at passe, plus klare kriterier for nye tests den bør skrive.
- **Stramt scope.** Én ticket, én bekymring. Peg ikke en natteagent mod "refaktorér autentificeringssystemet." Peg den mod "tilføj rate limiting til login-endpointet."
- **Branch-isolering.** Altid en frisk branch, altid et worktree. Hvis agenten laver rod, sletter du branchen. Intet rører main.
- **En timeout.** Sæt en hård grænse — fire timer er generøst for de fleste opgaver. En agent der har kørt i seks timer gør ikke fremskridt. Den kører i cirkler. Dræb den og revurdér om morgenen.

Et simpelt opsætningsscript ser sådan ud:

```
#!/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"
```

Du forfiner det over tid. Tilføj Slack-notifikationer når den er færdig. Tilføj et resumé af hvad den gjorde. Tilføj et tjek der verificerer at testsuiten passerer før den pusher.

Én ting jeg har lært fra teams der gør det godt: ticketbeskrivelsen er enormt vigtig. En ticket der siger "tilføj brugerpræferencers endpoint" er ikke nok. Natteagenten har brug for acceptkriterier, eksempel request/response payloads og pointers til lignende eksisterende endpoints den kan bruge som reference. Du skriver instruktioner til en kompetent men kontekstfri udvikler. Jo mere specifik du er, jo bedre bliver resultatet.

De teams der får mest ud af natteagenter er dem der investerer i deres ticketskrivningsdisciplin. Hvilket, igen, gavner alle — dine menneskelige holdkammerater foretrækker også klare tickets. Agenten gør bare omkostningen ved vaghed mere synlig.

Kerneloopet er simpelt: isolér, begræns, kør, review om morgenen.

13.3 Omkostningskontrol I CI

Når du kører en agent interaktivt, har du en naturlig sikringsafbryder: dig selv. Du ser tokens tikke op, du ser agenten køre i cirkler, du trykker Ctrl+C. I CI er der ingen der kigger.

Et hold der kørte en travl monorepo lærte det på den hårde måde. De havde sat en agent op til at auto-reviewe hver PR. Rimeligt nok — bortset fra at deres monorepo så fyrre til halvtreds PR'er om dagen, og hver review brugte omkring femten tusind tokens. Det er overskueligt. Hvad der ikke var overskueligt var retry-logikken. Når agenten ramte en rate limit, forsøgte CI-jobbet igen. Tre retries per fejl, eksponentiel backoff, men hvert retry startede en frisk agentkørsel fra bunden. Hen over en helligdagsweekend, med en batch af automatiserede afhængighedsopdateringer der strømmede ind, genererede pipelinen en $4.000-regning. Ingen var på kontoret til at bemærke det.

Beskyt dig selv:

- **Tokenbudgetter per pipeline-kørsel.** Sæt et hårdt loft. Hvis agenten rammer det, fejler jobbet med en klar besked. Det er bedre at misse en review end at brænde igennem dit månedlige budget på en dag.
- **Concurrency-begrænsninger.** Lad ikke tyve agentjobs køre samtidigt. Kø dem. To eller tre samtidige agentkørsler er rigeligt for de fleste teams.
- **Forbrugsalarmer.** Din LLM-udbyder understøtter næsten helt sikkert dem. Sæt en ved 50% af dit månedlige budget. Sæt en anden ved 80%. Led dem til en kanal nogen faktisk læser.
- **Kill switches.** Et feature flag eller environment variable der deaktiverer alle agent CI-trin øjeblikkeligt. Når noget går galt klokken 2 om natten, vil du have et én-linjers fix, ikke en pipelinekonfigurationsændring der kræver sin egen PR.
- **Per-job omkostningstracking.** Log tokenantal og estimeret omkostning for hver agent CI-kørsel. Du kan ikke optimere hvad du ikke måler. En ugentlig rapport over agent CI-forbrug, opdelt efter jobtype, vil vise dig hvor pengene går og hvor man skal stramme op.

En simpel sikringsafbryder i din CI-konfiguration ser sådan ud:

```
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
```

Detaljerne vil variere efter værktøj og platform, men mønstret er konstant: sæt et loft, fejl højlydt når du rammer det, og gør loftet nemt at justere.

Tommelfingerreglen: behandl dit LLM-forbrug i CI på den måde du behandler dit cloud compute-forbrug. Overvåg det, budgetér det, alarmér på det. Det er et rigtigt omkostningscenter.

13.4 Review-Spørgsmålet

Agentgenererede PR'er har stadig brug for menneskelig review. Punktum.

Fristelsen er reel. Agenten skrev koden. Testene passerer. Linteren er glad. Dækning er ikke faldet. Hvorfor ikke auto-merge? Du sparer femten minutters reviewtid, og CI-pipelinen har allerede verificeret alt der tæller.

Men tænk over hvad "alt der tæller" faktisk betyder. Tests verificerer korrekthed. De verificerer ikke intention.

En agent der har fået opgaven "reducér antallet af databaseforespørgsler på brugerprofilen" kan cache aggressivt og introducere stale data-bugs som ingen aktuel test fanger. Den kan denormalisere data på en måde der gør den næste feature dobbelt så svær at bygge. Den kan løse problemet perfekt men i en stil der er helt fremmed for resten af din codebase. Testene passerer fordi testene tjekker adfærd, ikke tilgang.

En menneskelig reviewer fanger disse ting. De læser for *hvordan*, ikke bare *hvad*. De bemærker når en løsning virker i dag men skaber gæld for i morgen. De spotter genvejen der vil forvirre den næste udvikler der rører koden.

Der er også en social dimension. Hvis dit hold ved at agent-PR'er auto-merges, stopper de med at være opmærksomme på agentgenereret kode. Den bliver en sort boks. Seks måneder senere er halvdelen af din codebase skrevet af en agent og ingen på holdet forstår den fuldt ud. Det er et vidensgab der vil skade jer under en incident.

Auto-merge er fint til trivielle, mekaniske ændringer — formateringsrettelser, importsortering, versionsbumps. Til alt der involverer en designbeslutning, reviewer et menneske det. Agenten er en hurtig udkaster, ikke en beslutningstager. Og reviewet behøver ikke at være udtømmende — en fem-minutters scanning for at tjekke at tilgangen er rimelig rækker langt.

13.5 Agentassisterede Deployments

Deploys er der hvor tingene bliver genuint farlige, og hvor gabet mellem "agenten kan gøre det her" og "agenten bør gøre det her" er bredest. Lad os være præcise om hvad agenter bør og ikke bør gøre her.

Agenter er fremragende til at **overvåge** deploys. De kan se logstrømme, tracke fejlrater, sammenligne latency-percentiler med pre-deploy baseline og flagge anomalier. En agent der siger "fejlraten på `/api/checkout` er steget 3x siden deployet for tolv minutter siden" er enormt værdifuld. Den læser data, finder mønstre og bringer information op til overfladen — præcis hvad agenter er gode til.

Agenter kan også **foreslå** handlinger. "Baseret på fejlratens stigning anbefaler jeg at rulle tilbage til den forrige version" er et nyttigt forslag. Det er den slags der normalt kræver at nogen stirrer på et Grafana-dashboard i femten minutter for at konkludere. Agenten har gjort analysen. Et menneske beslutter om der skal handles på det.

Hvad agenter ikke bør gøre er at tage rollback-beslutningen autonomt. Produktionsdeployments er for konsekvensrige til ad-hoc agentbeslutninger. Hvis du vil have automatiserede rollbacks, brug en forhåndsgodkendt runbook med hårde tærskelværdier — "hvis fejlraten overstiger 5% i tre minutter, rul automatisk tilbage." Det er deterministisk. Du testede det. Du godkendte logikken på forhånd. En agent der selv beslutter om en 2,3% fejlratestigning "føles" betydelig nok til at rulle tilbage? Det er en opskrift på enten unødvendige rollbacks eller oversete incidents. Deterministiske regler er kedelige. Kedeligt er godt når din omsætning er på spil.

I praksis ser en deploy-overvågningsagent nogenlunde sådan ud: den abonnerer på din logaggregator og metrikdashboard via API, kører i femten minutter efter hvert deploy og poster et resumé til Slack. "Deploy

af v2.4.1 gennemført. Fejlrate stabil på 0,3%. P99-latency steget fra 180ms til 210ms — inden for normal varians. Ingen nye undtagelsestyper detekteret." Det meste af tiden er det resumé kedeligt. Det er pointen. Når det ikke er kedeligt, vil du vide det med det samme.

Det kobler tilbage til guardrails-princippet: produktion er read-only for agenter. De observerer, analyserer, rapporterer, anbefaler. Mennesker (eller forhåndsgodkendt automatisering med deterministiske regler) tager handling.

13.6 Pipelinen Som Kontekst

Din CI/CD-konfiguration er kontekst, og agenter læser den som enhver anden kode. Det er nemt at glemme — de fleste teams behandler deres pipelinekonfiguration som infrastruktur-VVS, ikke som noget der behøver at kommunikere klart. Men når en agent forsøger at forstå hvorfor et build fejlede eller hvilke verifikationstrin der eksisterer, er pipelinekonfigurationen det første den læser.

En velstruktureret pipeline hjælper agenter med at forstå dit projekt. Klare stagenavne (`build`, `unit-tests`, `integration-tests`, `deploy-staging`) fortæller agenten hvordan din verifikationsproces ser ud. Struktureret fejloutput — JSON-logs, exit-koder med mening, fejlkategorier — giver agenten noget at arbejde med når et trin fejler.

En pipeline der outputter `BUILD FAILED` og intet andet er ubrugelig for agenter og mennesker i lige mål.

Investér i pipeline-observerbarhed:

- **Strukturerede logs.** JSON eller nøgle-værdi-par, ikke fritekst. En agent kan parse `{"stage": "unit-tests", "failed": 3, "passed": 247, "failures": ["test_auth_flow", "test_rate_limit", "test_session_expiry"]}` og tage meningsfuld handling. Den kan ikke gøre meget med `Tests failed. See above for details.`
- **Meningsfulde exit-koder.** Lad ikke alt exite med kode 1. Skeln mellem "tests fejlede" (kan fikses) og "kunne ikke forbinde til databasen" (infrastrukturproblem, ikke agentens bekymring).

- **Artefaktopbevaring.** Testresultater, dækningsrapporter, buildlogs — gem dem som pipeline-artefakter. En agent der debugger en CI-fejl har brug for at læse det fulde output, ikke bare de sidste ti linjer der passer i konsolvisningen.

Overvej at tilføje en `PIPELINE.md` til dit repo — en beskrivelse i klart sprog af hvad hvert CI-stage gør, hvordan dets forventede output ser ud, og hvilke almindelige fejltilstande der eksisterer. En agent der kan læse denne fil før den debugger en CI-fejl vil præstere dramatisk bedre end en der er nødt til at reverse-engineere din pipeline fra YAML-konfiguration alene. Og dine nyanstilte vil også takke dig.

Godt pipelinedesign og god agentintegration forstærker hinanden. Du forbedrer din CI for agenter, og dine menneskelige udviklere drager også fordel. Det er en af de sjældne investeringer der giver udbytte i begge retninger.

13.7 Start Småt

Hvis du har læst dette kapitel og er fristet til at koble agenter ind i hvert eneste stage af din pipeline inden fredag — lad være. Jeg har set den impuls. Den ender normalt med en weekend brugt på at fortryde automatiseringen fordi holdet ikke var klar til mængden af agentgenereret støj.

De teams der lykkes med agenter i CI er dem der opbygger tillid inkrementelt, ligesom med interaktive agenter.

Den gode nyhed er at stien er velafprøvet. Her er en praktisk adoptionsroadmap som jeg har set virke på tværs af multiple teams:

Uge 1: Autoformatering. Tilføj et CI-trin der kører en agent til at fikse formateringsproblemer på PR'er. Formatering er deterministisk, nem at verificere og lavrisiko. Hvis agenten formaterer noget forkert, er diffen lige der. Review agentens commits den første uge for at opbygge tillid.

Uge 2: PR-forscreening. Tilføj agentgenererede reviewkommentarer på PR'er. Ikke blokerende — kun informationelle. Lad dit hold se hvad agenten fanger, og kalibrer om dens feedback er nyttig eller støjende. Justér prompten baseret på hvad der virker.

Måned 1: Changelog-udkast. Peg agenten mod din commit-historik for at generere release note-udkast. Et menneske redigerer og publicerer. Du bruger agenten som førstekladsudkaster, ikke beslutningstager. Det er også et godt tidspunkt at sætte omkostningsovervågning og alarmer op.

Måned 3: Natteagenter. Nu har du opbygget tillid til agentgenereret output og du har infrastrukturen — worktrees, tokenbudgetter, branchisolering, reviewprocesser. Start med en enkelt velafgrænset ticket. Review resultatet omhyggeligt. Iterer på dit natscript. Udvid gradvist til mere komplekse opgaver efterhånden som din tillid vokser.

Bemærk hvad der *ikke* er på denne roadmap: auto-merging, autonome deployments eller agenter der tager arkitekturbeslutninger. Det er ikke trin fire. Det er måske trin ti, eller det er måske aldrig passende for dit hold. Roadmappen er ikke en march mod fuld automatisering — den er en march mod det rigtige niveau af automatisering for din kontekst.

Modstå trangen til at springe trin over. Hvert bygger på det forrige. Du lærer hvad dine agenter er gode til, hvor de kæmper, og hvilke guardrails du har brug for. Ved måned tre føles agentassisteret CI naturligt — ikke fordi du automatiserede alt på én gang, men fordi du fortjente hvert stykke af det gennem erfaring.

Pipelinen er bare endnu et miljø hvor agenter arbejder. De samme principper gælder: scop stramt, verificer grundigt, stol inkrementelt. Den eneste forskel er at ingen kigger, så dine sikkerhedsnet skal være så meget desto stærkere.

14 Når Agenter Tager Fejl

Enhver ingeniør der arbejder med agenter længe nok har en samling af disse historier. De øjeblikke hvor du læner dig tilbage i stolen, stirrer på skærmen og mumler noget der ikke kan trykkes. De er ydmygende. De er lærerige. Og de er *uundgåelige*.

Dette kapitel er en samling krigshistorier — ting der faktisk går galt når du giver rigtigt arbejde til en agent. Ikke hypotetiske. Ikke konstruerede demoer. Den slags fejl der koster dig en eftermiddag, en deploy eller din tro på automatisering. Hver en kan kobles tilbage til principper fra tidligere kapitler, for principper er billige indtil man lærer dem på den hårde måde.

Læs dem før du laver de samme fejl. Eller læs dem bagefter, og føl dig mindre alene.

14.1 Den Ivrige Refaktør

Opgaven var simpel: fix et z-index-problem på en dropdown-menu. Dropdown'en renderedes bagved en modal-overlay. Et én-linjers CSS-fix — måske to hvis man er grundig.

Agenten så dropdown-komponenten, besluttede at dens CSS var "inkonsistent med moderne praksis" og refaktorerede den. Så bemærkede den at modalen brugte et andet stylingsmønster, så den refaktorerede det også. Så knapkomponenterne. Så kortkomponenterne. Før du kiggede op fra din kaffe, havde toogtyve filer ændret sig. Diffen var 1.400 linjer. Agenten havde i bund og grund omskrevet komponentbibliotekets stylingssystem, migreret fra én tilgang til en anden med beundringsværdig konsistens og nul autorisation.

Den originale z-index-bug? Stadig der. Begravet et sted i lavinen af ændringer havde agenten reproduceret det samme lagdelingsproblem med nye klassenavne.

PR'en var umulig at reviewe. Du kan ikke meningsfuldt reviewe 1.400 linjer CSS-ændringer på tværs af toogtyve filer. Du *kan* slette branchen og starte forfra. Hvilket var hvad der skete.

Hvad du gør anderledes nu: Scop opgaver stramt. "Fix z-indexet på dropdown'en i `NavMenu.tsx`, rør intet andet." Brug en dedikeret branch så skaden er inddæmmet. Og tjek *altid* diffen før du lader agenten gå videre til noget andet. I det øjeblik du ser filantallet stige ud over hvad der giver mening for opgaven, stop agenten. Guardrails-kapitlet eksisterer af en grund.

14.2 Det Hallucinerede Bibliotek

Agenten havde brug for at parse nogle komplekse datointervaller fra brugerinput. Den importerede `@temporal/daterange-parser`, skrev elegant kode der brugte dens `.parseRange()`-metode med muligheder for lokalitetsbevidst parsing og fuzzy matching. Koden var ren. Typerne var korrekte. Fejlhåndteringen var gennemtænkt. Den skrev endda tests — som, naturligvis, også importerede den hallucinerede pakke.

Alt så perfekt ud i diffen. Funktionssignaturerne gav mening. API'et var veldesignet. Det var *så* et plausibelt bibliotek at man næsten ikke stillede spørgsmål ved det.

Så fejlede `npm install`. Pakken eksisterede ikke. Havde aldrig eksisteret. Agenten havde opfundet et bibliotek, givet det et troværdigt navn og et sammenhængende API, og skrevet produktionskode mod en fiktion.

Den lumske del: hvis du kun havde lavet code review — læst logikken, tjekket typerne, evalueret tilgangen — ville du have godkendt det. Koden var *god*. Den var bare ikke forbundet til virkeligheden.

Hvad du gør anderledes nu: Agenten kører testene. Altid. Hvis dit setup ikke tillader det, kører du dem selv før du reviewer kode. Ingen undtagelser. Et fejlende `npm install` er et højlydt, åbenlyst signal. Et hallucineret API der aldrig blev eksekveret er en lydløs bombe. Tests fanger hvad menneskelig review misser — ikke fordi din review er dårlig, men fordi plausible fiktioner er *designet* til at passere review. Det er hvad hallucination *er*.

14.3 Det Uendelige Loop

Du bad agenten om at fikse en fejlende integrationstest. Den læste fejlen, lavede en ændring, kørte testen. Ny fejl. Den læste *den* fejl, lavede endnu en ændring, kørte testen. Anderledes fejl. Så en variation af den første fejl. Så noget nyt. Så den første fejl igen.

Du var i en anden terminal og arbejdede på noget andet. Fyrre minutter senere tjekkede du ind. Agenten havde lavet nitten forsøg. Den havde brændt 200.000 tokens igennem. Koden var nu værre end da den startede — en geologisk lagdeling af fejlede fixes oven på hinanden. Agenten fortsatte stadig, muntert selvsikker på at forsøg tyve ville være det rette.

Det var det ikke.

Det fundamentale problem var at agenten ikke forstod *hvorfor* testen fejlede. Den mønstermatchede på fejlbeskeder og lavede lokale rettelser, men det faktiske problem var en arkitektonisk misforståelse — en delt databasetilstand mellem test cases der krævede en helt anden opsætningstilgang. Ingen mængde tilpasning af koden under test ville fikse et problem i test-harnessen.

Hvad du gør anderledes nu: Sæt iterationsgrænser. Tre forsøg på det samme problem er et rimeligt maksimum. Hvis agenten ikke har løst det i tre omgange, løser den det ikke i tredive. Når du ser loopet — fejl, fix, anden fejl, fix, original fejl — *bryd det manuelt*. Stop agenten, læs fejlene selv, og giv den enten en helt anderledes tilgang eller tag over. Agentens tid er billig. Din spildte eftermiddag er det ikke. Vigtigere endnu, start en frisk kontekst. Agentens forståelse er nu forurenet med nitten forkerte teorier. En ren start med din diagnose af det *faktiske* problem vil nå længere, hurtigere.

14.4 Det Selvsikre Forkerte Svar

Et baggrundsjob processerede af og til det samme element to gange. Klassisk race condition. Du pegede agenten mod problemet.

Agenten analyserede koden, identificerede race-vinduet og tilføjede et 500ms delay mellem check og update. “Det sikrer at den forrige transak-

tion har tid til at committe før den næste kontrol sker," forklarede den, med den rolige selvsikkerhed som nogen der aldrig har driftet et system under belastning har.

Testene passerede. Delayet var længere end testens transaktionstid, så race-vinduet lukkede — i testmiljøet, med én samtidig worker, på en stille maskine.

Det gik i produktion. Under belastning, med tolv workers og variabel databaselatency, var 500ms nogle gange ikke nok. Og nu havde du et *nyt* problem: delayet havde reduceret throughputtet nok til at jobkøen bakkede op under spidstimer, hvilket skabte kaskaderende timeouts der tog en urelateret service ned.

Sleep'et fiksede ikke race conditionen. Det skjulte den ved lav concurrency og gjorde systemet værre ved høj concurrency. Et ordentligt fix — en advisory lock eller en idempotency key — ville have været korrekt ved enhver skala. Agenten valgte det fix der gjorde testen grøn, ikke det fix der var *rigtigt*.

Hvad du gør anderledes nu: Du behandler passerende tests som nødvendige men ikke tilstrækkelige. Når en agent fikser et concurrency-problem, spørger du *dig selv* om fixet er korrekt under belastning, under fejl, under betingelser testsuiten ikke dækker. Agenter optimerer for det feedbacksignal du giver dem, og hvis det signal er "tests passerer," vil de finde den korteste vej til grønt — selv hvis den vej er en `time.Sleep`. Din ingeniørmæssige dømmekraft om *hvorfor* noget virker er vigtigere end *om* testene passerer. Det er den del af jobbet der endnu ikke er automatiseret. Læn dig ind i den.

14.5 Kontekstthukommelsestabet

To timer inde i en session havde du en fint udviklende feature. Du havde fortalt agenten tidligt: "Brug ikke nogen ORM — vi skriver rå SQL i dette projekt. Det er et bevidst valg." Agenten kvitterede for dette og skrev rene, håndlavede forespørgsler til de første flere opgaver.

Så bad du den om at tilføje et nyt endpoint. Halvfems minutters kontekst havde akkumuleret. Agenten byggede endpointet — med Prisma.

Fuld skemafil, migration, genereret klient. Smuk kode. Der fuldstændig modsagde begrænsningen du havde etableret i starten af sessionen og mønstrene i alle andre filer den havde skrevet den dag.

Da du påpegede det, undskyldte agenten, omskrev alt i rå SQL og opførte sig som om intet var sket. Den havde ikke *besluttet* at ignorere din begrænsning. Den havde simpelthen mistet den. Kontekstvinduet var fyldt med nok mellemliggende arbejde til at den tidlige instruktion var blegnet til irrelevans.

Hvad du gør anderledes nu: Lange sessioner degraderer. Det er en fundamental egenskab ved hvordan kontekstvinduer fungerer, ikke en bug der patches næste kvartal. Hold opgaver korte og fokuserede. Commit fungerende kode ved naturlige grænser så fremskridt er fanget i git, ikke bare i samtalen. Start friske sessioner til friske opgaver. Og for projektbrede begrænsninger som "ingen ORM" eller "ingen nye afhængigheder," put dem i en `CLAUDE.md`-fil eller tilsvarende som agenten læser ved opstart. Stol ikke på at agenten *husker* hvad du sagde for to timer siden. Det gør den ikke. Skriv det ned.

14.6 Afhængighedslawinen

Du bad om en datovælger. Et simpelt inputfelt hvor brugere kan vælge en dato. Agenten evaluerede mulighederne og besluttede at være grundig.

Den installerede `moment.js` til datohåndtering. Så `@popperjs/core` til positionering af dropdown'en. Så et helt UI-komponentbibliotek fordi "det tilbyder tilgængelige datovælger-primitiver." Så en CSS-preprocessor fordi komponentbibliotekets temasystem krævede det. Så to hjælpepakker som komponentbibliotekets datovælger havde brug for som peer dependencies.

Seks nye afhængigheder. Din bundlestørrelse gik fra 180KB til 540KB. Din byggetid blev fordoblet. Du havde dog en datovælger. Den var meget pæn.

Det native HTML `<input type="date">` ville have været fint. Eller et enkelt letvægts-pickerbibliotek på 8KB. I stedet havde du arvet et helt økosystem fordi agenten optimerede for fuldstændighed i stedet for minimalisme.

Den værste del var ikke bundlestørrelsen — det var vedligeholdelsesoverfladen. Seks nye pakker betyder seks nye ting der kan have sikkerhedssårbarheder, seks nye ting der kan gå i stykker ved opgradering, seks nye changelogs at læse når Dependabot begynder at oprette PR'er. Du tilføjede ikke bare en datovælger. Du adopterede seks open source-projekter.

Hvad du gør anderledes nu: Begræns hvad agenter kan installere. "Ingen nye afhængigheder uden at spørge mig først" er en legitim og ofte *klog* instruktion. Når du tillader nye pakker, fortæl agenten dine begrænsninger: bundlebudget, ingen pakker med færre end N ugentlige downloads, ingen pakker der trækker transitive megaafhængigheder med. Agenter har ikke meninger om bundlestørrelse eller vedligeholdelsesbyrde. De vedligeholder ikke projektet næste år. *Det gør du.* Så du sætter grænserne.

14.7 Den Røde Tråd

Hver eneste af disse historier har den samme grundlæggende årsag: agenten gjorde *præcis hvad den var designet til at gøre*, og mennesket gav ikke nok struktur.

Den ivrige refaktør var hjælpsom. Det hallucinerede bibliotek var kreativt. Det uendelige loop var vedholdende. Det selvsikre forkerte svar var testdrevet. Kontekshukommelsestabet var en begrænsning, ikke et valg. Afhængighedslawinen var grundig.

Ingen af disse er agentbugs. De er *workflow*-bugs. Fixet er aldrig "brug en klogere agent." Fixet er altid det samme: strammere scope, bedre feedback loops, mere struktur, kortere sessioner og et menneske der forbliver engageret.

Agenter tager fejl. Det gør mennesker også. Forskellen er at mennesker tager fejl langsomt nok til at det bemærkes. Agenter tager fejl med autocomplete-hastighed, og når du kigger op fra din kaffe har toogtyve filer ændret sig.

Bliv i loopet. Tjek diffs. Stol men verificer. Og når tingene går skævt — for det gør de — husk at slet-branchen-og-start-forfra-knappen er det mest undervurderede værktøj i dit workflow.

14.8 Den Diagnostiske Drejebog

Krigshistorier er underholdende. Men du kan ikke tape anekdoter på din skærm. Hvad du har brug for er en systematisk tilgang — en tjekliste til når agenten producerer noget forkert, så du kan diagnosticere fejlen hurtigt og fikse det rigtige i stedet for at famle.

Når en agent giver dig dårligt output, kør igennem disse spørgsmål i rækkefølge. De fleste fejl falder i en af seks kategorier, og at vide hvilken kategori du er i bestemmer hvad du gør derefter.

1. Er det et scopingproblem?

Bad du om for meget i én prompt? Den Ivrige Refaktør skete fordi "fix z-indexet" var for vagt — det sagde ikke "rør intet andet." Hvis agenten ændrede filer du ikke forventede, eller lavede arbejde du ikke bad om, var scopet for løst. Stram prompten. Vær eksplicit om hvad der *ikke* skal gøres. Begrænsninger er mere nyttige end instruktioner når man har med en entusiastisk agent at gøre.

2. Er det et kontekstproblem?

Havde agenten hvad den havde brug for til at løse det faktiske problem? Tænk tilbage på faktureringsbugshistorien fra kapitel 2 — én ingeniør gav vag kontekst og fik et vagt svar, den anden gav specifikke filer og fejlspor og fik et fungerende fix. Hvis agenten løste det *forkerte* problem, kunne den sandsynligvis ikke se det rigtige. Giv den de relevante filer, fejloutputtet, de begrænsninger den ikke kan udlede. Agenter gætter ikke godt. De arbejder med det du giver dem.

3. Er det et feedbacksignalproblem?

Verificerer dine tests faktisk det rigtige? Det Selvsikre Forkerte Svar skete fordi testene passerede — men de testede ikke under realistiske betingelser. 500ms sleep-"fixet" var grønt i CI og forkert i produktion. Hvis agentens løsning føles tvivlsom men testene passerer, er *dine tests problemet*. Agenten optimerede for det signal du gav den. Giv den et bedre signal.

4. Er det et kapacitetsproblem?

Nogle opgaver er genuint ud over hvad modellen kan. Multi-fil-ræsonnering på tværs af et udbredt system med implicitte afhængigheder. Subtile concurrency-problemer der kræver forståelse af runtime-adfærd, ikke bare kodestruktur. Sikkerhedsfølsom logik hvor "plausibelt" ikke er godt nok. Hvis agenten bliver ved med at prøve forskellige tilgange og fejler, er den måske ikke i stand til denne bestemte opgave. Det er ikke et promptproblem — det er en begrænsning. Anerkend den og tag over. Din tid er bedre brugt på at gøre arbejdet end på at konstruere den perfekte prompt til en opgave agenten ikke kan håndtere.

5. Er det et kontekstvindueproblem?

Lange sessioner degraderer. Punktum. Kontekstvindue-historien demonstrerede dette — begrænsninger sat tidligt i en session forsvinder simpelthen efterhånden som konteksten fyldes med mellemliggende arbejde. Hvis agenten modsiger noget den gjorde korrekt tidligere i den samme session, eller ignorerer en begrænsning du satte for en time siden, er kontekstvinduet synderen. Start en frisk session. Gentag de relevante begrænsninger. Giv den kun den kontekst den har brug for til den aktuelle opgave, ikke det arkæologiske arkiv over alt hvad I har diskuteret i dag.

6. Er det et loop?

Tre forsøg på den samme fejl er grænsen. Hvis agenten ikke har løst det i tre omgange, løser den det ikke i tredive. Det Uendelige Loop-historien brændte 200.000 tokens af over nitten forsøg uden fremskridt. Når du ser mønstret — fejl, fix, anden fejl, fix, original fejl — bryd loopet øjeblikkeligt. Stop agenten. Læs fejlene selv. Giv enten agenten en helt anderledes tilgang med en frisk diagnose, eller tag fuldstændig over. Agentens kontekst er nu forurenet med fejlede teorier, og flere forsøg vil kun tilføje mere forurening.

Print denne tjekliste. Sæt den fast på din skærm. De første par gange kører du igennem den bevidst. Efter en måned bliver det instinkt. Efter tre måneder fanger du problemet før agenten overhovedet er færdig med sit første forsøg.

15 Hvornår Man Ikke Skal Bruge Agenter

Denne bog handler om at bruge agenter godt. Dette kapitel handler om at vide hvornår man slet ikke skal bruge dem.

Hvis du har læst så langt, er du sandsynligvis solgt på agentisk ingeniørarbejde. Godt. Men den hurtigste måde at miste troværdighed — og spilde tid — er at gribe efter en agent når jobbet kræver et menneske. De bedste agentiske ingeniører er ikke dem der bruger agenter mest. De er dem der ved *præcis* hvornår de skal lægge agenten væk og gøre arbejdet selv.

15.1 Overheadskatten

Enhver agentinteraktion har en omkostning. Du skriver prompten. Du venter på outputtet. Du reviewer hvad den producerede. Du fikser fejlene. Du genkører hvis den missede pointen. Det er overhead, og det er reelt.

For komplekse opgaver der ville tage dig en time er to minutter på prompt og review en god handel. Men for opgaver du kan gøre på tredive sekunder? Overheadet gør agenter *langsommere*, ikke hurtigere.

At omdøbe en variabel. Fikse en tastefejl. Justere en konfigurationsværdi. Tilføje en loglinje. Det er muskelhukommelsesopgaver. Dine fingre kender tastetrykkene. Inden du har tastet en prompt der beskriver hvad du vil, kunne du allerede have gjort det.

Det er ikke en fejl ved agenter. Det er aritmetik. Små opgaver har små gevinster, og de faste omkostninger ved agentinteraktion æder marginen. Lad ikke nyhedsværdien af agenter narre dig til at bruge dem til alt. Noget arbejde er bare hurtigere i hånden.

15.2 Nye Arkitekturbeslutninger

Når du designer et system fra bunden — vælger mellem en monolit og microservices, beslutter din datamodel, vælger dine kommunikationsmønstre — er *tænkningen arbejdet*. Værdien ligger ikke i diagrammet eller dokumentet. Værdien ligger i den mentale model du bygger mens du kæmper med afvejningerne.

En agent kan hjælpe dig med at udforske muligheder. Den kan liste fordele og ulemper ved event sourcing versus CRUD. Den kan skitsere hvordan en bestemt arkitektur kan se ud. Det er nyttigt som input.

Men at delegere selve arkitekturen til en agent betyder at du ikke forstår dit eget system. Du vil kæmpe med at debugge det, udvide det eller forklare det til dit hold. Senioringeniørens job er at tage de svære beslutninger — at afveje de afvejninger der ikke har rene svar, at beslutte hvilken kompleksitet der er værd at bære og hvilken der ikke er. Den dømmekraft kommer af at gøre arbejdet, ikke af at læse en agents sammendrag af det.

Brug agenter som sparringspartner til arkitektur. Ikke som arkitekten.

15.3 Sikkerhedskritisk Kode

Autentificeringsflows. Kryptering. Adgangskontrol. Inputvalidering. Tokenhåndtering. Det er områder hvor "ser korrekt ud" ikke er godt nok.

Sikkerhedsbugs er anderledes end almindelige bugs. En ødelagt sorteringsfunktion producerer forkert output som nogen bemærker. En ødelagt auth-kontrol producerer *ingen synlige symptomer* indtil en angriber finder den. Koden ser fin ud. Testene passerer. Og seks måneder senere skriver du en incidentrapport.

Agenter producerer plausibel kode. Det er deres styrke og, i sikkerhedskontekster, deres fare. En subtil fejl i et JWT-valideringsflow, en manglende kontrol på en redirect-URL, en timing-sidekanal i en adgangskodesammenligning — det er den slags fejl der overlever code review fordi de *ser rigtige ud*.

Skriv sikkerhedskritisk kode selv. Review den omhyggeligt. Få et andet par menneskelige øjne på den. Hvis du bruger en agent til at udkaste

sikkerhedskode, behandl det udkast med mere mistanke end du ville give en juniorudviklers første forsøg, ikke mindre.

15.4 Når Du Har Brug For At Lære

Du tager et nyt sprog op. Et nyt framework. Et nyt paradigme. Fristelsen er åbenlys: lad agenten skrive koden mens du fokuserer på det overordnede billede.

Modstå det.

Hvis du lader agenten skrive al Rust-koden mens du lærer Rust, har du ikke lært Rust. Du har bygget noget du ikke kan vedligeholde, debugge eller udvide uden agenten. Du har skabt en afhængighed, ikke en færdighed.

Der er en afgørende forskel mellem at bruge en agent til at *forklare* noget og bruge den til at *gøre* noget. At spørge "hvorfor opstår denne borrow checker-fejl?" opbygger forståelse. At spørge "fix denne borrow checker-fejl" gør det ikke.

Når målet er læring, sæt farten ned. Skriv koden selv. Lav fejlene selv. Brug agenter som tutorer, ikke ghostwriters. Den forståelse du opbygger ved at kæmpe igennem de svære dele er hele pointen.

15.5 Følelsesladede Beslutninger

Ikke enhver ingeniørbeslutning er teknisk. Nogle af de sværeste er menneskelige.

At deprecate et API som en partner afhænger af. At fortælle en stakeholder at deres feature request ikke når med. At skubbe tilbage på en deadline du ved er urealistisk. At beslutte at lukke et produkt der stadig har brugere.

Disse beslutninger kræver empati. De kræver at man læser rummet, forstår politikken, afvejer de menneskelige omkostninger ved siden af de tekniske. De kræver *ansvarlighed* — nogen der vil eje beslutningen og dens konsekvenser.

En agent kan udkaste emailen. Den kan hjælpe dig med at gennemtænke talende punkter. Men selve beslutningen, og samtalen der leverer den, skal komme fra et menneske. Folk fortjener at høre svære nyheder fra en person, ikke fra nogen der copy-pastede en agents output.

15.6 Når Codebasen Er For Rodet

Agenter forstærker hvad der allerede er der. I en ren, velstruktureret codebase med stærke konventioner producerer agenter ren, velstruktureret kode. De opfanger mønstrene og følger dem.

I et rod? De producerer mere rod.

Hvis din codebase har inkonsistent navngivning, sammenfiltrede afhængigheder, ingen klare modulgrænser og tre forskellige måder at gøre det samme på — vil en agent opfange *alle* de mønstre. Den kan kombinere de værste dele af hvert. Den ved ikke hvilke mønstre der er tilsigtede og hvilke der er teknisk gæld. Den ser bare hvad der er der og producerer mere af det.

Nogle gange er det rigtige træk at rydde op før du bringer agenter ind. Refaktorér modulet. Etablér konventionen. Slet den døde kode. Gør codebasen til et sted hvor en agent kan gøre godt arbejde. Det er glamourløst, manuelt slid. Men det er det fundament der gør alt andet muligt.

Tænk på det som et værksted. Du giver ikke elværktøj til nogen i et rodet, uorganiseret værksted. Du rydder op først, *så* bringer du værktøjerne ind.

15.7 At Arbejde Med Legacy-Kode

Den værkstedsmetafor er pæn. Men ikke alle har den luksus at rydde op først. Nogle af os vedligeholder 500.000-linjers Java-monolitter der blev startet i 2014 og har været igennem tre framework-migrationer, to opkøb og en kort periode hvor nogen syntes XML-konfiguration var en god idé. Rådet om at "refaktorere før du bringer agenter ind" er sundt i teorien og latterligt i praksis når du stirrer på en codebase der ville tage år at refaktorere.

Så hvordan *bruger* man agenter i legacy-kode? Forsigtigt. Og med en specifik strategi.

Start med testene. Før du peger en agent mod legacy-kode, skriv karakteriseringstests — tests der fanger hvad koden *aktuelt* gør, ikke hvad den *burde* gøre. De er ikke aspirationelle. De er beskrivende. De siger "når du kalder denne funktion med disse inputs, er det her der kommer ud, og det er den nuværende kontrakt uanset om nogen havde det som intention eller ej."

Karakteriseringstests giver agenten et sikkerhedsnet. Uden dem vil agenten ændre adfærd den ikke forstår, og du vil ikke vide det før produktion fortæller dig det. Med dem viser enhver adfærdsændring sig som en testfejl *før* koden forlader din maskine. Det er ufravigeligt. Legacy-kode uden tests er legacy-kode du ikke sikkert kan lade agenter røre.

Scop brutalt. I en legacy-codebase kan agenten ikke forstå hele systemet. Forsøg ikke at få den til det. Peg den mod én fil, én funktion, én bug. Giv den den umiddelbare kontekst — funktionssignaturen, kallerne, den forventede adfærd — og intet andet. Legacy-kode er fuld af implicitte antagelser, udokumenterede sideeffekter og bærende mærkværdigheder. Jo mindre agenten ser, jo mindre kan den misfortolke. Et snævert scope med klar kontekst slår et bredt scope med arkæologisk kompleksitet.

Brug agenter til de kedelige dele. Legacy-codebases er fulde af mekanisk arbejde: opdatering af deprecated API-kald på tværs af hundredvis af filer, migrering fra én afhængighedsversion til en anden, tilføjelse af typeannotationer til utypet kode, standardisering af fejlhåndteringsmønstre, udskiftning af ét lognings-framework med et andet. Det er *perfekte* agentopgaver — repetitive, veldefinerede, let verificerbare af automatiserede checks. Lad agenter gøre det kedsommelige. Gem din energi til de dele der kræver forståelse af *hvorfor* koden er som den er. Det er arbejde kun et menneske der har levet med systemet kan gøre.

Dokumentér undervejs. Hver gang en agent arbejder på en legacy-fil med succes, tilføj en kort kommentar der forklarer hvad koden gør, eller opdater projektets `CLAUDE.md` med kontekst om det modul. Over tid sker der noget interessant: de dele af legacy-codebasen som agenter har rørt bliver bedre dokumenteret end de dele de ikke har. Ikke fordi du satte

dig for at dokumentere systemet, men fordi at bruge agenter *tvang* dig til at artikulere hvad hver del gør for at prompte effektivt. Agenterne gør langsomt codebasen mere læsbar — ikke ved at refaktorere den, men ved at få dig til at forklare den.

15.8 Håndværksargumentet

Der er én grund mere til nogle gange at lægge agenten væk, og den handler ikke om effektivitet eller risiko. Den handler om håndværk.

At skrive kode i hånden opbygger noget agenter ikke kan give dig. Muskelhukommelse. Mønstergenkendelse der lever i dine fingre, ikke bare dit hoved. Den dybe, intuitive forståelse der kommer af at have skrevet den samme slags funktion dusinvis af gange. Den stille tilfredsstillelse ved at løse et svært problem gennem din egen ræsonnering.

De ting er vigtige. Ikke fordi de er romantiske, men fordi de gør dig til en bedre ingeniør. Udvikleren der har håndskrevet en parser forstår parsing anderledes end en der kun har promptet en agent til at skrive en. Udvikleren der har debugget et concurrency-problem klokken to om natten *mærker* faren ved delt muterbar tilstand på en måde som at læse om det aldrig giver.

Agenter er værktøjer. Kraftfulde. Men hvis du lader dem gøre alt arbejdet, atrofierer dine egne færdigheder. Og når du rammer et problem som agenten ikke kan løse — og det gør du — har du brug for at de færdigheder stadig er skarpe.

Hold dig i øvelse. Skriv kode i hånden regelmæssigt. Ikke fordi det er hurtigere, men fordi det holder dig farlig.

15.9 Den Dømmekraft Der Tæller

Den centrale færdighed i agentisk ingeniørarbejde er ikke prompting. Det er ikke værktøjskonfiguration. Det er ikke at vide hvilken model man skal bruge til hvilken opgave.

Det er *dømmekraft*.

At vide hvornår en agent vil spare dig en time og hvornår den vil spilde en. At vide hvornår du skal stole på outputtet og hvornår du skal omskrive det fra bunden. At vide hvornår du skal delegere og hvornår du skal grave dig ned selv.

De bedste agentiske ingeniører bruger agenter aggressivt — men selektivt. De rækker ud efter agenter når løftestangseffekten er reel: storstilede refaktoreringer, boilerplate-generering, kodeudforskning, testskrivning, dokumentation. Og de lægger agenten væk når arbejdet kræver menneskelig tænkning, menneskelig ansvarlighed eller menneskeligt håndværk.

Den dømmekraft er hvad der adskiller agentiske ingeniører fra promptjockeys. Alle kan taste en prompt. At vide *hvornår man ikke skal* er den sværere færdighed.

16 Agentiske Teams

Software har altid været en holdsport. Agenter ændrer ikke det. Men de ændrer holdets form, arbejdets rytme og de færdigheder der tæller. Hvis du leder et hold — eller arbejder på et — skal du tænke over det nu, ikke senere.

16.1 10x-Multiplikatoren Er Reel, Men Fordelt Anderledes

Du har hørt påstanden: agenter gør ingeniører 10x mere produktive. Det er ikke forkert. Det er bare misvisende.

Agenter gør visse opgaver *dramatisk* hurtigere. At generere boilerplate, skrive testscaffolding, refaktorere på tværs af hundrede filer, migrere API-versioner, koble CRUD-endpoints op — det går fra timer til minutter. Hastighedsgevinsten er reel og den er massiv.

Men andre opgaver ændrer sig knap. Systemdesign kræver stadig tænkning. At debugge en subtil race condition kræver stadig tålmodighed, intuition og dyb forståelse af runtime. Stakeholdersamtaler tager stadig den tid de tager. Agenten kan ikke sidde i dit arkitekturreview og stille de rigtige spørgsmål.

10x-multiplikatoren er ikke en flad multiplikator hen over din dag. Det er en spikegraf. Nogle opgaver går 50x hurtigere. Nogle går 1x. Et par kan endda gå langsommere hvis du kæmper med agenten i stedet for at gøre det selv.

Teams der forstår dette deployer agenter strategisk. De giver ikke hver opgave til en agent og forventer magi. De identificerer højløftestangszonerne — de opgaver hvor agenter genuint komprimerer tidslinjer — og de fokuserer agentindsatsen der. Resten forbliver menneskeligt.

16.2 Code Review Ændrer Sig

Her er hvad der sker når agenter kommer ind i et holds workflow: PR-volumen stiger. *Markant.* En ingeniør der plejede at åbne to PR'er om dagen åbner nu fem. Koden i de PR'er er syntaktisk korrekt, velformateret og passerer tests. Og at reviewe den er *udmattende.*

Den gamle model for code review — læs hver linje, tjek for off-by-one-fejl, verificer edge case-håndtering — skalerer ikke når halvdelen af koden blev genereret på sekunder. Du brænder dine reviewere ud på en uge.

Skiftet er fra "er det korrekt?" til "er det den rigtige tilgang?" Agentgenereret kode er normalt *lokalt* korrekt. Den gør hvad der blev bedt om. Spørgsmålet er om det der blev bedt om var det rigtige. Skal denne nye service eksistere, eller bør logikken bo i den eksisterende? Er det den rigtige abstraktionsgrænse? Gør denne ændring systemet simplere eller mere komplekst?

Reviewere bliver arkitekter. De zoomer ud. De tjekker intention, ikke implementering.

Praktiske tilpasninger der virker:

- Mindre, mere fokuserede PR'er — nemmere for både agenter og reviewere
- Automatiserede tjek håndterer det mekaniske (linting, testdækning, type checking)
- Reviewtid er beskyttet i kalenderen, ikke klemt ind mellem møder
- Teams aftaler "betroede mønstre" — hvis en PR følger et kendt mønster og passerer CI, får den en hurtigere reviewsti

Reviewtræthed er den stille dræber af agentassisterede teams. Tag det alvorligt.

16.3 Junior-Ingeniørspørgsmålet

Det er det sværeste problem i dette kapitel, og jeg har ikke et rent svar.

Junioringeniører har traditionelt lært ved at gøre det arbejde som agenter nu gør hurtigere og bedre. At skrive det første CRUD-endpoint. At kæmpe med et tricky CSS-layout. At finde ud af hvorfor testen fejler. Disse

opgaver var kedelige for seniorer men *formative* for juniorer. Kampen var uddannelsen.

Hvis agenter håndterer alt det, hvor sker læringen så?

Det værste resultat er juniorer der bliver prompt-afhængige — de kan få en agent til at generere kode, men de kan ikke forklare hvad koden gør eller debugge den når den går i stykker. De springer forståelsesfasen helt over. Det er ikke ingeniørarbejde; det er et meget dyrt copy-paste-workflow.

Den bedre vej er at bruge agenter som *tutorer*, ikke erstatninger. En junior der skriver en databasemigration bør bede agenten om at *forklare* migrationen, ikke bare generere den. "Hvorfor brugte du en transaktion her?" "Hvad sker der hvis det fejler halvvejs?" "Vis mig hvordan det ser ud uden ORM'en." Agenten bliver en tålmodig lærer med uendelig tid — noget de fleste seniorer ikke kan tilbyde.

Parprogrammering med agenter, *superviseret af seniorer*, er den mest lovende model jeg har set. Junioren styrer agenten. Senioren observerer, stiller spørgsmål og griber ind når junioren accepterer noget de ikke forstår. Det er langsommere end at lade agenten gøre alt, men det producerer ingeniører der faktisk ved hvad de laver.

Teams der springer denne investering over låner fra fremtiden. Dagens uovervågede juniorer er morgendagens seniorer der ikke kan debugge produktion uden en AI-krykke.

16.4 Vidensfordeling

Her er et mønster der dukker op i hvert hold der adopterer agenter ujævnt: én ingeniør bliver flydende med agenter, begynder at shippe med 3x hastigheden af alle andre, og inden for et par måneder har de rørt ved hver del af codebasen. De bliver det eneste svaghedspunkt.

Busfaktoren falder til én. Ikke fordi nogen planlagde det, men fordi hastighed og videnskoncentration er korrelerede. Ingeniøren der shipper mest lærer mest. De andre falder bagud, ikke bare i output men i *forståelse* af det system de kollektivt ejer.

Det er et ledelsessproblem, ikke et teknologiproblem. Fixet er strukturelt:

- **Delte `CLAUDE.md`-filer.** Hvert projekt har én. Alle bidrager til den. Den koder holdets kollektive viden, ikke én persons.
- **Delte workflows og konventioner.** Holdet aftaler hvordan de bruger agenter — hvilke værktøjer, hvilke mønstre, hvilke guardrails. Ingen ensomme ulvesetups.
- **Rotation.** Agentflydende ingeniører roterer til forskellige dele af codebasen. Viden spredes gennem arbejde, ikke dokumentation.
- **Agentsessionsdeling.** Nogle teams er begyndt at dele interessante agentsessioner — prompts, outputs, beslutninger. Det er en form for vidensoverførsel der ikke eksisterede før.

Målet er ikke at bremse din hurtigste ingeniør. Det er at sørge for at holdets viden holder trit med holdets output.

16.5 Delte Konventioner Betyder Mere

Vi dækkede konventioner i et tidligere kapitel. I en holdkontekst er indsatserne højere.

Når en soloingeniør bruger agenter, påvirker deres konventioner én person. Når et hold bruger agenter, påvirker konventioner *hver agentsession på tværs af hele holdet*. Et velstruktureret projekt med klar navngivning, konsistente mønstre og en vedligeholdt `CLAUDE.md` betyder at enhver ingeniørs agenter starter fra et stærkt fundament.

Et rodet projekt betyder at hver agent genopfinder hjulet. Forskellige ingeniører får forskellige outputs. Codebasen driver. Reviews bliver sværere fordi du nu reviewer ikke bare koden men *stilen* af koden, der varierer efter hvilken ingeniørs agent der skrev den.

Kodestandarder i et agentisk hold handler ikke om æstetik. De handler om *agenteffektivitet*. Holdet der aftaler projektstruktur, navnekonventioner, testmønstre og dokumentationsformat får bedre output fra hver agentsession. Det er en kraftmultiplikator på en kraftmultiplikator.

Investér tiden. Skriv standarderne ned. Håndhæv dem i CI. Det betaler sig på hver eneste PR.

16.6 Det Nye Standup

Hvordan ser daglig koordinering ud når hver ingeniør kører multiple agentsessioner parallelt?

Det gamle standup: "I går arbejdede jeg på auth-refaktoreringen. I dag fortsætter jeg. Ingen blokere."

Det nye standup: "Jeg har tre agenter kørende. Auth-refaktoreringen landede og er i review. API-migrationen er 80% færdig — agenten sad fast på den legacy XML-parsing, så jeg tager det over manuelt. Jeg sparkede lige en tredje session i gang for at generere integrationstests til faktureringsmodulet."

Granulariteten ændrer sig. Arbejdet bevæger sig hurtigere, så koordinering skal holde trit. En ingeniør kan *starte og afslutte* en opgave mellem standups. Den daglige synk bliver mindre om fremskridtsopdateringer og mere om intentionsjustering — at sikre at tre ingeniører ikke alle sender agenter efter det samme problem fra forskellige vinkler.

Nogle teams bevæger sig mod asynkrone standups med hyppigere check-ins. Andre bruger delte dashboards der tracker aktive agentsessioner. Det rigtige svar afhænger af holdet, men den gamle rytme med "én opdatering per person per dag" er ofte ikke nok.

16.7 Compliance og Revisionsporet

Hvis en agent skriver kode der forårsager en produktionsincident, hvem er så ansvarlig? Ingeniøren der promptede den? Revieweren der godkendte PR'en? Holdlederen der besluttede at adoptere agentiske workflows? Det er ikke et filosofisk spørgsmål man debatterer over en øl. Det er et juridisk og compliance-spørgsmål, og regulerede brancher har brug for klare svar *før* incidenten sker, ikke under postmortem.

Den gode nyhed er at svaret faktisk ikke er så kompliceret. Værktøjerne og processerne eksisterer allerede. Du skal bare være eksplicit om dem.

Git er dit revisionspor. Hvert agentgenereret commit bør være tilskrivbart. Commit-beskeden bør indikere at det var agentassisteret — et `Co-Authored-By`-tag, et præfiks, uanset hvilken konvention dit hold

adopterer. PR'en bør vise hvem der reviewede den. Mergegodkendelsen er underskriften. Det er allerede sådan de fleste teams arbejder; nøglen er at gøre det *konsistent*. Tilfældig tilskrivning — nogle gange med tag, nogle gange uden — er værre end intet system overhovedet, fordi det skaber illusionen om en proces uden pålideligheden af en.

Revieweren ejer det. Det praktiske svar for de fleste organisationer er ligetil: ingeniøren der reviewer og godkender PR'en tager ansvar, ligesom de ville for enhver kode fra enhver kilde. Agentgenereret kode får ikke en anden ansvarsstandard. Hvis du godkender en PR, siger du "Jeg har reviewed dette og jeg mener det er korrekt." Værktøjet der genererede koden er irrelevant for den udtalelse. Det betyder også at reviews af agentgenereret kode skal være *rigtige* reviews, ikke gummistempler. Hvis volumenet af agentgenererede PR'er gør grundig review umulig, er det et workflowproblem at løse, ikke en standard at sænke.

Til regulerede brancher. Dokumentér dit agentiske workflow som del af din SDLC-dokumentation. Hvilke modeller der bruges, hvilken version, hvilke guardrails der er på plads, hvilken reviewproces agentgenereret kode gennemgår før den når produktion. Revisorer vil se en *proces*, ikke perfektion. En dokumenteret proces der inkluderer "AI-assisteret kodegenerering med obligatorisk menneskelig review og CI-verifikation" er reviderbar. En udokumenteret proces hvor ingeniører bruger de værktøjer de har lyst med ingen konsistent tilgang er det ikke. Hvis du er i fintech, sundhed eller noget med reguleringsmæssigt tilsyn, få det dokumenteret før nogen beder om det.

Bevar sessionslog. Bevar logs af agentsessioner, særligt for kode der rører følsomme systemer — fakturering, autentificering, datahåndtering, alt med reguleringsmæssige implikationer. Ikke fordi du vil læse dem rutinemæssigt, men fordi du kan have brug for dem under en incidentgennemgang. "Hvad så agenten da den genererede denne kode? Hvilken prompt producerede dette output? Hvilken kontekst arbejdede den med?" Det er spørgsmål du gerne vil kunne svare på seks måneder senere. De fleste agentværktøjer kan eksportere eller logge sessioner. Sæt bevaringen op før du har brug for den. Omkostningen ved opbevaring er triviel sammenlignet med omkostningen ved ikke at have loggene når compliance banker på.

16.8 Ansættelse Ændrer Sig

Hvis agenter håndterer de mekaniske dele af kodning, hvad har du så faktisk brug for fra en ingeniør?

Rå kodningshastighed tæller mindre. Ingeniøren der kunne banke en perfekt binær søgning ud på en whiteboard på tre minutter har en færdighed som agenter har gjort til masseware. Det er ikke værdiløst — at forstå algoritmer er stadig vigtigt — men det er ikke længere det der gør forskellen.

Hvad der tæller mere:

- **Systemdesign.** Evnen til at nedbryde et problem i komponenter, definere grænseflader og ræsonnere om afvejninger. Agenter kan implementere et design. De kan ikke fortælle dig hvilket design der er rigtigt til dine begrænsninger.
- **Dømmekraft.** At vide hvornår man skal bruge agenten og hvornår man skal tænke. At vide hvornår agentens output er forkert selvom det ser rigtigt ud. At vide hvilke hjørner man kan skære og hvilke man skal beskytte.
- **Kommunikation.** Evnen til at artikulere intention klart — til agenter, til holdkammerater, til stakeholdere. Vage tænkere får vagt agentoutput. Præcise tænkere får præcise resultater.
- **Problemnedbrydning.** At bryde en stor opgave ned i agentstørrelsebidder er en færdighed. Den er relateret til systemdesign men mere taktisk. Ingeniøren der kan forvandle en Jira-epic til ti velafgrænsede agentprompts vil udperformere den der paster hele epicen ind i et chatvindue.

Interviewet der beder om "implementer denne algoritme på en whiteboard" tester en færdighed der tæller mindre hvert år. Interviewet der beder om "her er et system med disse begrænsninger — gå mig igennem hvordan du ville bygge det, hvilke afvejninger du ville lave, og hvordan du ville verificere at det virker" tester de færdigheder der tæller *mere* hvert år.

Ansæt for dømmekraft. Du kan altid give dem en agent til resten.

17 Afsluttende Ord

Min ældste søn er fem. Han kan ikke rigtig læse endnu — han staver sig igennem ord på morgenmadspakker og vejskilte, stolt af hver stavelse. Men han ved hvad min computer gør. Han har set mig tale til den, set tekst dukke op på skærmen som svar, set mig nikke eller ryste på hovedet og tale igen. En aften kravlede han op i mit skød, så en agent refaktorere et modul i realtid — filer der åbnede og lukkede, tests der kørte, grønne flueben der dukkede op — og sagde: "Reparerer computeren sig selv?"

Jeg havde ikke et godt svar. Det har jeg stadig ikke, ikke helt. Men det spørgsmål fulgte mig gennem hvert kapitel af denne bog. For hvad jeg indså, siddende der med ham, var at jeg ikke skrev en bog om værktøjer. Jeg skrev en bog om hvad det vil sige at være ingeniør i det præcise øjeblik definitionen af ingeniørarbejde bliver omskrevet — og om hvad vi bærer med ind i hvad end der kommer derefter.

Dette kapitel er ikke et resumé. Du behøver ikke at jeg opsummerer femten kapitler du lige har læst. Det her er det jeg vil sige direkte til dig, én ingeniør til en anden, før vores veje skilles.

17.1 Hvad Jeg Tror

Jeg tror at håndværket ikke er ved at dø. Det bliver *komprimeret*. Et årti af boilerplate og VVS og ceremoni kollapser til intention og dømmekraft. Det der er tilbage når man fjerner tastningen er det der altid var det egentlige job: at vide hvad man skal bygge og vide hvornår det er rigtigt.

Jeg tror agenter er forstærkere, ikke erstatninger. Giv en agent til en middelmådig ingeniør og du får middelmådig software med skræmmende hastighed. Giv en til en fremragende ingeniør og du får noget der, ærligt talt, får de sidste tyve års softwareudvikling til at ligne at vi byggede motorveje med skovle.

Jeg tror de ingeniører der trives vil være dem der mestrer de *kedelige* ting — kontekst, guardrails, testing, konventioner, klar tænkning — mens alle andre jager den skinnende nye modelannoncering. Værktøjer skifter hvert kvartal. Dømmekraft akkumulerer over en karriere.

Jeg tror vi ikke bliver erstattet. Vi bliver forfremmet. Fra maskinskrivere til kaptajner. Fra kodere til *ingeniører*, i ordets fuldeste forstand. Spørgsmålet er om du accepterer forfremmelsen.

Jeg tror dette skift er *godt*. Ikke nemt. Ikke smertefrit. Men godt. For den del af vores arbejde der bliver automatiseret var aldrig den del vi elskede. Ingen blev ingeniør fordi de drømte om at skrive boilerplate JSON-parsere. Vi blev ingeniører fordi vi ville bygge ting der betyder noget. Nu kan vi.

17.2 Hvad Jeg Tog Fejl I

Jeg skal være ærlig overfor dig: jeg skiftede mening mindst tre gange mens jeg skrev denne bog.

Da jeg startede kapitlet om sandboxes, troede jeg containerisolering var overkill til de fleste workflows. Da jeg var færdig med det, efter at en agent havde rm -rf'et et bibliotek jeg brød mig om en tirsdag eftermiddag, troede jeg sandboxing var ufravigeligt. Den oplevelse kom med i kapitlet. Overbevisningen bag det er arvæv.

Jeg skrev oprindeligt multi-agent-kapitlet med den antagelse at orchestrering af fem eller seks agenter samtidigt var den naturlige slutttilstand — en fabriksgulv af autonome arbejdere. Jeg har trukket mig fra det. Koordineringsoverheadet er reelt, fejltilstandene multiplicerer, og jeg har fundet at to eller tre velstyrede agenter udperformer seks uovervågede næsten hver gang. Kapitlet afspejler hvor jeg landede, men jeg lander måske et andet sted om seks måneder.

Jeg var også, i en periode, for afvisende overfor lokale modeller. Jeg skrev et tidligt udkast der basalt sagt sagde "brug bare de kommercielle API'er." Så brugte jeg en weekend på at køre en fintunnet lokal model på en codebase med proprietære begrænsninger og indså at der er en hel verden af use cases hvor lokalt ikke bare er levedygtigt men *nødvendigt*.

Kapitlet om lokale versus kommercielle modeller eksisterer fordi jeg tog fejl og måtte rette mig selv.

Dele af denne bog er sandsynligvis allerede forkerte på måder jeg ikke kan se endnu. Landskabet bevæger sig så hurtigt. Men de specifikke værktøjer var aldrig pointen. Hvis jeg har hjulpet dig med at opbygge en mental model — en måde at tænke om autonomi, tillid og struktur — så har bogen gjort sit job, selv når alle kodeeksempler i den er forældede.

17.3 Mandskabsmetaforen, Én Sidste Gang

Jeg voksede op i Danmark, tæt på vandet. Hvis du har sejlet i skandinaviske farvande, kender du lyset om sent om sommeren — lavt og gyldent, den slags lys der får havet til at ligne hamret kobber. Jeg husker en overfart engang, en lille båd, fire af os. Vinden drejede hårdt og pludseligt bevægede vi os alle uden at tale. Én person på fokken, én på storsejlet, én der trimmede, én ved roret. Ingen kommandoer. Bare tillid opbygget over snesevis af tidligere ture.

Det er hvad agentisk ingeniørarbejde føles som på en god dag. Du ved roret, agenter der trimmer og justerer, arbejdet der flyder fordi du har lagt timerne i at opbygge fælles kontekst — gennem konventioner, gennem guardrails, gennem testsuiter der fanger fejl før de tæller. Du behøver ikke at råbe instruktioner. Systemet *ved*.

Og så er der de dårlige dage. De dage vinden lægger sig og en agent hallucinerer et API der ikke eksisterer, eller omskriver et modul du ikke bad den om at røre, eller passerer alle testene fordi den slettede dem der fejlede. De dage øser du vand og bander. Det er også sejlads.

Men jeg bør være ærlig om noget, for denne bog virker ikke hvis jeg ikke er det.

Metaforen om et *mandskab* antyder loyalitet. Kontinuitet. Skibskammerater du har sejlet med før, der kender dine vaner, der foregriber den næste kommando. Det er et smukt billede. Det er heller ikke sådan jeg faktisk arbejder.

De fleste dage er hvad jeg faktisk gør at spinne en agent op, give den et job, tage outputtet og smide den over bord. Så spinner jeg en anden op. De husker ikke den sidste session. De ved ikke hvad jeg bad den forrige agent om at gøre. Hver ankommer frisk, gør sit arbejde og forsvinder. Det er mindre et loyalt mandskab og mere som at hyre havnearbejdere i hver havn — du briefer dem, du ser dem arbejde, du betaler dem af, og i næste havn gør du det igen.

Og det er fint. Det er sådan de fleste rigtige mandskaber fungerede gennem maritim historie. Skibet var kontinuiteten. Kaptajnen var kontinuiteten. Kortene, logbogen, rigningen — det persisterede mellem rejser. Mandskabet blev ofte samlet til én enkelt overfart og opløst ved destinationen. Det der fik det til at fungere var ikke at søfolkene kendte kaptajnen. Det var at kaptajnen kendte *skibet* — og havde systemer gode nok til at enhver kompetent sømand kunne gå om bord og være nyttig.

Det er hvad din codebase er. Det er hvad dine konventioner, dine testsuiter, dine CLAUDE.md-filer, dine guardrails er. De er skibet. Hver ny agent du spinner op er et friskt besætningsmedlem der træder om bord på et velrigget fartøj. De behøver ikke at kende din historie. De behøver at kende skibet. Og hvis du har bygget skibet godt, vil de være produktive på minutter.

Så ja — smid dem over bord. Spin nye op. Det er ikke et svigt af metaforen. Det er metaforen der virker præcis som tiltænkt. Mandskabet er til at smide væk. Skibet er det ikke.

Du er kaptajnen. Du var altid kaptajnen. Mandskabet er netop ankommet — og de vil blive ved med at ankomme, friske og klar, hver gang du har brug for dem.

17.4 Tak

Tak fordi du læste denne bog. Det mener jeg. Du byttede din tid og opmærksomhed for mine ord, og det tager jeg ikke let på. Jeg håber jeg fortjente det.

Tak til communityet — ingeniørerne i fora, i Discord-servere, i open source-repos — der deler deres eksperimenter, deres fejl, deres hårdt

tilkæmpede indsigter. Denne bog blev formet af hundredvis af samtaler jeg ikke havde alene.

Og tak, formoder jeg, til agenterne selv — der hjalp mig med at skrive kode, debugge problemer og af og til generere prosa så dårlig at det mindede mig om hvorfor menneskelig dømmekraft stadig er vigtig. I er et godt mandskab. I bliver bedre. Og jeg har en mistanke om at inden min søn er gammel nok til at læse denne bog, vil I være noget ingen af os helt forudså.

17.5 Gå Ud Og Byg Noget

Her er hvad jeg vil have dig til at gøre.

I aften — ikke i morgen, ikke i næste uge, *i aften* — åbn din terminal. Vælg en bug du har undgået. Den du bliver ved med at flytte til bunden af backloggen, den der er irriterende men ikke presserende, den der bor i en del af codebasen du helst ikke vil røre. Peg en agent mod den. Giv den kontekst. Sæt en guardrail. Se hvad der sker.

Måske fikser den buggen på fire minutter og du føler grunden flytte sig under dine fødder, som den flyttede sig under mine den aften med migrationsscriptet. Måske laver den rod og du lærer noget om hvordan man giver bedre instruktioner. Uanset hvad ved du mere end du gjorde før.

Så gå større. Sæt worktree-isolering op og kør to agenter parallelt. Skriv en CLAUDE.md-fil til et projekt du holder af. Byg en testsuite der er god nok til at være dit sikkerhedsnet når agenter committer kode. Refaktorér det modul alle er bange for — men denne gang med et mandskab.

Så gå endnu større. Introducér agentiske workflows til dit hold. Del hvad du har lært — sejrene *og* katastroferne. Skriv dine egne krigshistorier ned. Bidrag til den kollektive viden om en disciplin der stadig er ved at blive opfundet.

For det er hvad det her er: en disciplin der opfindes, lige nu, af de mennesker der er villige til at gøre arbejdet. Ikke af dem der skriver blogindlæg om fremtiden. Ikke af dem der venter på det perfekte værktøj.

Af de mennesker der åbner en terminal i dag og bygger noget rigtigt med det de har.

De ingeniører der vil definere denne æra venter ikke på tilladelse. De venter ikke på sikkerhed. De shipper, bryder ting, lærer og shipper igen — med et mandskab ved deres side der bliver lidt bedre hver dag.

Jeg håber du vil være en af dem.

Rasmus Bornhøft Schlünsen
Marts 2026

Til mine børn —
I er det bedste mandskab, jeg nogensinde har haft.
Det hele var for jer.
Altid.