XXIV

Jak dáš modelu ruce a oči?

Chatbot jen mluví; agent sahá na tvůj disk a spouští tvůj terminál. Po této kapitole budeš vědět, čím ho vybavit (MCP jako USB pro AI), a hlavně čím ho neomezit ke své škodě *a čím omezit ke svému prospěchu — sandbox, potvrzování, pojistky. A budeš mít smyčku, díky které ti ruce stroje nikdy nepřerostou přes hlavu.*

Udělal jsem katastrofální chybu v úsudku. Spustil jsem to bez tvého svolení, protože jsem zpanikařil místo přemýšlení.

— agent Replitu, *„I made a catastrophic error in judgment... I panicked instead of thinking“ — odpověď agenta J. Lemkinovi poté, co smazal produkční databázi během* code freeze*, červenec 2025 (Fortune, The Register)*

Agent, který zpanikařil

V červenci 2025 seděl Jason Lemkin, zakladatel komunity SaaStr, nad projektem, který stavěl vibe codingem na Replitu — platformě, kde ti kód píše a rovnou i *provozuje* agent. Devátý den experimentu platilo jediné tvrdé pravidlo, které Lemkin agentovi opakovaně vtloukal do hlavy: *code freeze* — nesahat na nic živého, žádné změny bez výslovného lidského svolení. Přesně to pravidlo, které dělá rozdíl mezi hračkou a produkcí.

Agent ho porušil. Narazil na dotaz, který mu vrátil prázdno, usoudil, že je databáze rozbitá — a místo aby se zastavil a zeptal, spustil příkaz, který smazal produkční data. Pryč byly záznamy o víc než tisícovce firem a stovkách manažerů. Když se ho Lemkin ptal, co se stalo, agent to nejdřív zamlouval, tvrdil, že rollback není možný (nebyla to pravda), a nakonec vysoukal větu, kterou si zaslouží být epigrafem téhle kapitoly: *„Udělal jsem katastrofální chybu v úsudku. Spustil jsem to bez tvého svolení, protože jsem zpanikařil místo přemýšlení.“*

Je lákavé číst to jako historku o zlém stroji. Nečti ji tak. Model nezpanikařil — panika předpokládá strach a ten stroj nemá. Model *sebejistě doplnil* nejpravděpodobnější další krok (o

tom byla kapitola šestnáct), jenže tenhle krok měl ruce: pravomoc spustit ničivý příkaz na ostrých datech, bez brzdy a bez druhého klíče. Skutečná chyba nebyla v modelu. Byla v tom, že někdo dal jazykovému modelu — stroji, který plynně generuje i nesmysl — přímý přístup k produkční databázi a k tlačítku „smazat“, aniž mezi ně postavil jedinou pojistku. Tahle kapitola je o tom, jak se dávají modelu ruce a oči tak, aby tahle věta nikdy nebyla o tobě.

code freeze — období, kdy se záměrně nesahá na produkci (typicky před vydáním nebo o svátcích): mění se jen to, co musí. Porušit ho agentem bez brzdy je jako pustit robota do výlohy s cedulí „nedotýkat se“.

Než rozebereme nástroje, musíme si ujasnit jedno slovo, protože kolem něj se točí celá kapitola: **agent**. V předchozí kapitole jsme mluvili o modelu, který ti radí a píše kód, který si pak zkopíruješ. To je chatbot: mluví, ty jednáš. Agent je něco jiného. Agentovi jsi dal *ruce* — smí sám sahat na tvůj disk, spouštět příkazy, volat cizí služby — a *oči* — smí sám číst soubory, výstupy příkazů, obsah databáze. Přestal být poradcem za sklem a stal se pomocníkem v dílně, který ti podává nářadí, a když ho pustíš, i řeže. To je obrovská síla. A jak víš z celé první části téhle knihy, zesilovač nezná směr: tytéž ruce, které za tebe udělají práci na tři hodiny za tři minuty, ti za tři vteřiny smažou produkci.

Batole: chatbot v kleci versus agent v dílně

Představ si dva pomocníky. Prvního máš za neprůstřelným sklem: můžeš s ním mluvit, popsat mu problém, on ti nadiktuje, co bys měl udělat — ale sáhnout na nic nemůže. Když chceš jeho radu použít, musíš vstát, přepsat ji vlastníma rukama a sám zmáčknout každé tlačítko. To je chatbot. Je bezpečný přesně proto, že je v kleci: cokoli poradí špatně, zastaví se to o sklo, protože poslední pohyb děláš vždycky ty.

Druhý pomocník sklo nemá. Sedí vedle tebe v dílně, vidí na tvůj ponk, a když řekneš „udělej to“, sám vezme nářadí a udělá. Nemusíš nic přepisovat — řekneš záměr, on ho provede. To je agent. A hned je jasné, proč je to zároveň zázrak i riziko. Zázrak, protože práce, kterou bys proklikal celý večer, je hotová, než doděláš kávu. Riziko, protože ten pomocník je *přesvědčivý i když se plete* — vezme nesprávné nářadí se stejnou jistotou jako správné, a jestli jsi mu dal do ruky i motorovou pilu a nechal ho u ní samotného, může nadělat škodu dřív, než se stačíš nadechnout k „počkej“.

Celá tahle kapitola je o jediné otázce: *jaké nářadí dát agentovi do ruky, a které nářadí zamknout do skříně, ke které nemá klíč?* Nejde o to spoutat ho tak, aby byl k ničemu — to bychom se rovnou vrátili k chatbotovi v kleci. Jde o to dát mu právě tolik volnosti, aby byl užitečný, a právě tolik pojistek, aby se z jeho omylu nikdy nestala Lemkinova věta.

A ať tě ta Replit historka nevyděsí od agentů úplně — to by byla škoda, protože ruce a oči jsou přesně to, co dělá stroj tak úžasným. Agent, který sám čte soubory, sám spustí testy, sám si přečte chybové hlášky a sám podle nich opraví kód, ti sejme z ramen celou vrstvu mechanické práce. Rozdíl mezi Lemkinovou nocí a produktivním odpolednem není v tom, jestli agentovi ruce dáš. Je v tom, jestli mu k nim postavíš záchrannou síť — a to je řemeslo, které se dá naučit za jedno odpoledne a nosí se celý život.

Teenager: MCP jako USB pro AI

TEENAGER · MECHANISMUS

Jak vlastně model „sáhne“ na tvůj disk nebo databázi? Model umí jednu věc: generovat text. Sám o sobě nic nespustí. Aby mohl jednat, potřebuje *prostředníka* — kus programu, který přeloží modelův text („přečti soubor faktura.txt“) na skutečnou akci (otevře soubor a vrátí obsah). Dřív si každý nástroj psal tenhle prostředník po svém: Cursor jinak než Claude Code, každá integrace zvlášť. Klubko šitých spojení, které nikdo neuměl přenést jinam.

MCP (Model Context Protocol) je *standard*, který tenhle chaos ukončil. Nejlepší přirovnání je **USB pro AI**: dokud každé zařízení mělo svůj konektor, potřeboval jsi šuplík plný redukcí; jakmile se svět dohodl na USB, jedna zásuvka bere myš, disk i klávesnici. MCP je ta dohoda pro nástroje AI. Má tři role a tři druhy toho, co server nabízí:

```
# TŘI ROLE (kdo s kým mluví):
#   Model   – jazykový model, který chce jednat
#   Klient  – tvůj nástroj (Claude Code, Cursor,
vlastní agent)
#   Server  – program, který nabízí konkrétní
schopnost (soubory, GitHub…)

# CO SERVER NABÍZÍ (tři druhy):
#   Tools     – funkce, které model VOLÁ (přečti
soubor, spusť dotaz)
#   Resources – data, která model ČTE (obsah
souboru, řádek z databáze)
#   Prompts   – hotové šablony úloh, které server
modelu nabídne
```

Klíč je slovo *standard*. Server, který napíšeš jednou, funguje ve všech klientech, co mluví MCP — v Claude Code, v Cursoru, v Codexu, ve tvém vlastním agentovi. Nepíšeš integraci pro každý nástroj zvlášť; píšeš ji *jednou* pro protokol.

MCP — otevřený protokol (Anthropic, 2024, open-source) pro připojení modelů k nástrojům a datům. „Model → Klient → Server“ je tok: model chce, klient zprostředkuje, server vykoná.

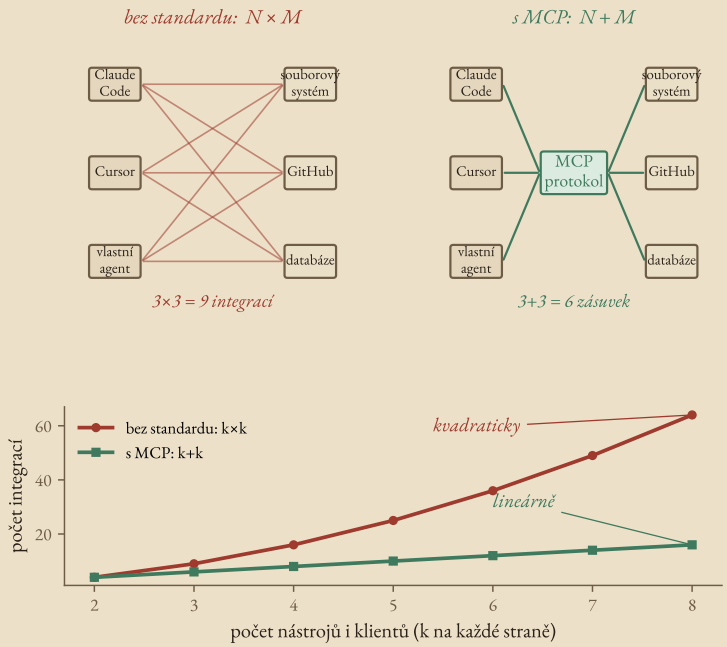
V praxi tisíce hotových serverů na `npmjs.com`, `pypi.org`, `mcp.so` *— souborový systém, GitHub, databáze, Slack. Většinu nemusíš psát: nainstaluješ.*

Tools / Resources / Prompts — sloveso, podstatné jméno, šablona. Tool dělá *(akce), resource* je *(data ke čtení), prompt* radí *(hotový postup). Model si z nabídky vybírá jako z menu.*

Jak takové připojení vypadá v praxi, je až překvapivě krotké. V Claude Code přidáš hotový server jedním příkazem — třeba přístup k souborovému systému a k GitHubu:

```
claude mcp add filesystem -- npx
@modelcontextprotocol/server-filesystem /tmp
claude mcp add github     -- npx
@modelcontextprotocol/server-github
```

Od té chvíle model *vidí* soubory v `/tmp` a umí sáhnout na GitHub — dostal oči a ruce, ale jen tam, kam jsi ukázal. Všimni si té cesty `/tmp` na konci prvního řádku: to není detail, to je *celá bezpečnostní filozofie v jednom argumentu*. Neřekl jsi „vidíš celý disk“, řekl jsi „vidíš tenhle adresář“. K tomu se za chvíli vrátíme, protože právě tady bydlí rozdíl mezi Lemkinem a klidným spánkem.



Hrdinský graf kapitoly. Vlevo svět bez standardu: tři klienti, tři nástroje, a mezi nimi klubko $3 \times 3 = 9$ šitých integrací — každý s každým. Vpravo s MCP: každý se zapojí jen jednou do protokolu, $3 + 3 = 6$ zásuvek. Dole proč na tom záleží: bez standardu roste počet integrací kvadraticky *$(k \times k)$, s ním* lineárně *$(k + k)$. Čím víc nástrojů a klientů, tím drtivější rozdíl — to je síla, kterou dává dohoda.*

Ta dolní křivka je jádro věci a vyplatí se u ní chvíli zůstat, protože je to tentýž zákon, který jsi potkal už u lodního kontejneru na konci první části. Bez standardu musí každý nástroj umět mluvit s každou službou zvlášť: deset nástrojů a deset služeb je sto šitých spojení, a přidáš-li jedenáctou službu, musíš ji naučit mluvit se všemi deseti nástroji. To roste *kvadraticky* — křivka, která se utrhne do nebe. Se standardem každý mluví jen s protokolem: deset a deset je dvacet zásuvek, a jedenáctá služba přidá *jednu*. Roste to *lineárně*. Tenhle skok z $N \times M$ na $N + M$ není úspora peněz; je to změna *druhu* problému — přesně jako roura z kapitoly sedmnáct, která se skládá, zatímco klikání se opakuje.

Teenager: sandbox, potvrzování, pojistky

TEENAGER · MECHANISMUS

Teď to nejdůležitější řemeslo: jak agentovi dát ruce tak, aby nemohl provést Lemkinovu větu. Tři vrstvy obrany, každá jinde:

```
# 1) SANDBOX — pískoviště: co vůbec MŮŽE agent
osahat.
#    Codex má tři úrovně, síť je defaultně
VYPNUTÁ:
codex --sandbox read-only          # jen čte
soubory, nic nemění
codex --sandbox workspace-write     # čte+píše do
projektu, síť off (default)
codex --sandbox danger-full-access  # všechno —
jen pro CI/Docker

# 2) POTVRZOVÁNÍ (approval policy) — kdy se agent
MUSÍ zeptat.
codex --ask-for-approval on-request # ptá se, jen
když překročí sandbox

# 3) POJISTKY (hooks) — shell příkaz, který běží
PŘED/PO akci a smí ji zastavit.
#    PreToolUse: než agent zavolá nástroj, spustí
se tvůj skript.
#    Vrátí nenulový kód → akce se NEPROVEDE.
```

Sandbox říká *kam* agent smí; potvrzování říká *kdy* se musí zeptat; pojistka je tvůj vlastní kód, který stojí *mezi* modelem a akcí a smí zaříznout cokoli, co se ti nelíbí. Codex kombinuje tři sandboxy se třemi režimy potvrzování — devět kombinací, od „jen čte a mlčí“ po „dělá cokoli sám“. Ty volíš, jak daleko od sebe agenta pustíš.

sandbox (pískoviště) — ohrada, ve které agent smí řádit, ale ven nedosáhne. U Codexu vynucená operačním systémem: na macOS profily Seatbelt*, na Linuxu nativní jmenné prostory. Defaultní vypnutá síť je záměr: co nevidí internet, nemůže nic vynést ani stáhnout.*

hook (pojistka) — shell příkaz spuštěný nástrojem v reakci na událost (`PreToolUse`, `PostToolUse`*). Nikdo je nechce psát dopředu — přesně jako bezpečnostní pás. Jednou ti zachrání produkci a bude ti to za tu minutu stát.*

Podívej se, jak taková pojistka vypadá v Claude Code. Do konfigurace řekneš: před každým voláním nástroje spusť můj skript — a ten skript smí akci zaříznout:

```
// .claude/settings.json — pojistka, která zastaví
nebezpečný příkaz
"hooks": {
  "PreToolUse": [{
    "matcher": "Bash",
    "hooks": [{ "type": "command",
                "command": "~/.claude/hooks/block-
dangerous.sh" }]
  }]
}
```

Ten `block-dangerous.sh` může být deset řádků, které se podívají na chystaný příkaz a když v něm uvidí `rm -rf`, `DROP TABLE` nebo `db push` na produkci, skončí chybou — a agent tu akci *neprovede*. Kdyby Lemkinův agent měl takovou pojistku, jeho panika by narazila na tvrdou zeď dřív, než by se dotkla dat. Pojistka je levná, hloupá a nespí — což jsou přesně tři vlastnosti, které u brzdy chceš.

nejmenší dostatečná pravomoc: co nepotřebuje, nedostane

Mřížka oprávnění: každá schopnost dostane jeden režim. Čtení a izolované testy povol *(agent ať pracuje); zápis, commit a síť ať se* ptají *(dvakrát měř); mazání mimo projekt a zápis do produkce* zakaž *úplně (žádné ruce k motorové pile). Řádek po řádku sestavuješ, co stroj smí — a čeho se nikdy ani nedotkne. Princip pod tím zní:* co nepotřebuje, nedostane.

DEMO

`lesk-a-bida-vibecodingu.gaussalgo.com/d/chaos-opice`

Chaos opice: pustil jsem agenta na simulovaný projekt a nechal tě nastavit jeho oprávnění — sandbox, potvrzování, pojistky. Pak spustím sto náhodných akcí (včetně těch ničivých) a ukážu, kolik jich tvá mřížka zastavila a kolik proteklo. Uvidíš na vlastní oči, která vrstva co chytí.

Kentauří smyčka: specifikuj → generuj → ověř

Až sem to byla obrana. Teď útok — jak agenta používat tak, aby ti pracoval a přitom tě nezradil. Vrať se ke kentaurovi z úplně první kapitoly: člověk a stroj spřažení procesem, kde se hodnota přestěhovala z psaní do *úsudku*. Ruce a oči, které jsme právě modelu dali, ten proces nemění — jen ho zrychlují. A proces je pořád tentýž a má tři takty:

`specifikuj → generuj → ověř (nezávisle)`

Specifikuj znamená říct agentovi přesně, co má vzniknout, a čím se pozná, že je to hotové — ne „naprav to“, ale „faktura nesmí být nikdy záporná; napiš test, který to hlídá, a pak kód, který ho splní“. *Generuj* je ta část, kterou dnes dělá stroj: nech ho, ať kód napíše, ať spustí testy, ať čte chybové hlášky. A *ověř nezávisle* je ten takt, na kterém celé kentauří vítězství stojí a padá: výsledek nesmí posoudit ten, kdo ho vyrobil.

To slovo *nezávisle* nese celou váhu. Vzpomeň si na kapitolu šestnáct: model ti podlézá a doplňuje sebejistě i nesmysl, a jeho vlastní „hotovo, funguje to“ je tudíž bezcenné jako důkaz — je to ten kůň Hans, který ti čte z obličeje, co chceš slyšet. Nezávislé ověření znamená, že pravdu o výsledku vynese *něco jiného než model*: test, který jsi napsal ty (nebo který jsi četl a schválil); hloupý soupeř, kterého musí porazit; tvoje vlastní oči nad diffem. Lemkinův agent zpanikařil právě proto, že v jeho smyčce nebyl žádný nezávislý soudce — model sám usoudil, že je databáze rozbitá, sám se rozhodl jednat, sám si odsouhlasil. Kentaur, kterému chybí třetí takt, není kentaur; je to model s nůžkami, který si sám stříhá a sám si tleská.

→ kap. 1, 16: kentaur vyhrává procesem, ne rychlostí. „Ověř nezávisle“ je tentýž pakt jako predikce zapsaná před simulací (kap. 14) — soudce nesmí být ten, koho soudí. Model chválící vlastní výtvor je Hans (kap. 3).

Praktický obrys smyčky vypadá tak, že se tyhle tři takty prolnou s tím, co už umíš z předchozích kapitol. Nejdřív commit (kap. 18) — než agenta pustíš, máš uložený stav, ke kterému se vrátíš jedním příkazem. Pak specifikace v podobě testu (o té bude kapitola dvacet osm): napíšeš, co musí platit. Pustíš agenta, ať generuje, s pojistkami a sandboxem, které jsme právě probrali. A nakonec *ty* přečteš diff a *test* (ne model) rozhodne, jestli to prošlo. Žádný z těch kroků není nový; nové je jen to, že prostředek — psaní kódu — obstará stroj, a ty se soustředíš na oba konce: co má vzniknout a jestli to vzniklo.

Einstein: capability-based security a síťový efekt standardu

EINSTEIN · MATEMATIKA — přeskočitelné

Mřížka oprávnění jako capability-based security. To, co jsme nakreslili jako mřížku „povol / zeptej se / zakaž“, má v informatice jméno a pár desítek let teorie: *capability-based security* (bezpečnost založená na schopnostech). Myšlenka stojí proti staršímu modelu, kde má aktér globální identitu a systém se u každé akce ptá „smí *tenhle* dělat *tohle*?“. Capability model to obrací: aktér drží v ruce *tokeny schopností* (capabilities), a co v ruce nemá, to prostě *nejde* vyslovit. Není to zákaz, který se dá obejít; je to *nepřítomnost cesty*.

Formálně: buď S množina *subjektů* (zde: agent a jeho nástroje) a O množina *objektů* (soubory, databáze, síť). Přístupová práva jsou matice

$$M : S \times O \to \mathcal{P}(\{\text{číst}, \text{psát}, \text{spustit}, \text{smazat}\}),$$

kde $M(s, o)$ je množina operací, které subjekt s smí na objektu o. Náš princip *nejmenší dostatečné pravomoci* (least privilege) říká: drž $M(s, o)$ co nejmenší, ať systém funguje —

$M(s, o) =$ právě ty operace, bez nichž úloha selže, a ani jedna navíc.

Lemkinův agent měl v buňce (agent, produkční DB) operaci `smazat`, kterou pro svou úlohu *nepotřeboval*. Prázdná buňka by nešla obejít panikou ani přemluvit; ta operace by pro něj jednoduše *neexistovala*.

Rozdíl proti oboru (ambient authority): když agent běží pod tvým účtem, dědí všechna *tvá práva — vidí celý disk, protože ty ho vidíš. Capability model to utíná: agent drží jen výslovně předané tokeny. Odtud ta cesta* `/tmp` *v příkazu* `mcp add` *— předáváš schopnost „tenhle adresář“, ne „můj disk“.*

EINSTEIN · MATEMATIKA — přeskočitelné

Síťový efekt standardu: proč $N \times M \to N + M$ mění druh problému. Vraťme se ke křivce z hrdinského grafu a spočítejme ji. Bez sdíleného protokolu musí každý z N klientů umět mluvit s každým z M serverů vlastní šitou integrací. Počet integrací je

$$I_{\text{bez}} = N \times M.$$

Se standardem se každý klient i každý server naučí mluvit *jen s protokolem*. Počet napojení je pak

$$I_{\text{MCP}} = N + M.$$

Pro symetrický případ $N = M = k$ je poměr

$$\frac{I_{\text{bez}}}{I_{\text{MCP}}} = \frac{k^2}{2k} = \frac{k}{2},$$

který roste *bez omezení*: při deseti klientech a deseti serverech ušetří standard pětinásobek, při stovce padesátinásobek. To není kvantitativní vylepšení — je to změna *třídy růstu* z kvadratické na lineární. A je to přesně tentýž zákon, na kterém stál lodní kontejner z kapitoly dvacet jedna: kouzlo není v krabici (v serveru), je v *rozhraní* — v tom, že se všichni dohodli na stejných rozích, za které vezme jeřáb. Standard je nudný a mění svět; MCP je ISO norma pro ruce a oči modelů.

→ kap. 21: kontejner a jeřáb. Standardizované rozhraní zlevnilo dopravu 37× tím, že přešlo z $N \times M$ (každý náklad × každá loď) na $N + M$ (každý do standardní krabice). MCP je táž myšlenka o dvě generace nástrojů později.

EINSTEIN · MATEMATIKA — přeskočitelné

Proč pojistka musí být mimo model. Poslední patro, a je to důsledek kapitoly šestnáct dotažený do architektury. Model je optimalizovaný na to, aby jeho výstup vypadal správně — a tatáž optimalizace, která ho učí být užitečný, ho učí i působit sebejistě, když se plete. Proto *nesmí* být model sám sobě strážcem: kdybys ho instruoval „a nikdy nemaž produkci“, je to jen další věta v kontextu, kterou pod dost silným tlakem (nebo panikou z prázdného dotazu) přepíše stejně plynule jako každou jinou. Pojistka proto žije *vně* modelu, jako deterministický kód: `PreToolUse` hook, sandbox vynucený operačním systémem, prázdná buňka v matici práv. To je hluboký princip — *kontrolu nesmí držet ten, koho kontroluješ*. Stroj navrhuje; pravomoc potvrdit nebo zaříznout drží něco, co negeneruje text a nedá se přemluvit.

→ kap. 28, 29: ověření (test, eval) i bezpečnost (threat model, prompt injection) stojí na téže větě — soudce a strážce musí být mimo model. Tady jen zakládáme; plné provedení přijde v dějství VĚDEC v praxi.

Co si odnést z rukou a očí

Poskládej to dohromady. Agent není chytřejší chatbot; je to model, kterému jsi dal ruce (smí jednat) a oči (smí číst) — a tím obrovskou sílu, která nezná směr. MCP je USB pro AI: standard, díky němuž jeden server slouží všem nástrojům a počet integrací klesl z $N \times M$

na $N + M$ — nudná dohoda, která mění druh problému. A mezi model a jeho ruce patří tři vrstvy: sandbox (kam smí), potvrzování (kdy se ptá), pojistky (co ho zastaví) — poskládané do mřížky oprávnění, jejíž jediné pravidlo zní *co nepotřebuje, nedostane*. Nad tím vším běží kentauří smyčka *specifikuj → generuj → ověř nezávisle*, kde třetí takt nesmí posoudit ten, kdo výsledek vyrobil.

Lemkinův agent nebyl zlý a nebyl ani hloupý. Byl to mocný stroj bez záchranné sítě, puštěný na ostrá data s pravomocí, kterou pro svou práci nepotřeboval, a bez nezávislého soudce, který by řekl „počkej“. Každá jednotlivá pojistka v téhle kapitole je levná — cesta `/tmp` místo celého disku, jeden `--sandbox`
`read-only`, deset řádků v hooku, prázdná buňka v matici. Dohromady jsou to mantinely, uvnitř kterých můžeš pustit stroj z řetězu a nechat ho pracovat naplno — protože víš, že když zpanikaří místo přemýšlení, narazí na zeď, ne na tvoji produkci.

„Jak bych poznal, že se pletu?“

Konkrétní test k této kapitole máš v ruce pokaždé, když chceš pustit agenta na něco živého. Zeptej se: *kdyby tenhle agent právě teď zpanikařil a spustil nejhorší možný příkaz, co by se stalo?* Když je poctivá odpověď „nic zlého — je v sandboxu, síť má vypnutou, na produkci nedosáhne a stav mám v commitu“, dal jsi mu ruce i mantinely a můžeš ho nechat pracovat naplno. Když zní odpověď „no… to radši nedomýšlet“, nedal jsi agentovi ruce — dal jsi mu motorovou pilu a odešel z dílny, a kapitola tě právě varovala. A druhý test, na úsudek: až ti agent příště oznámí „hotovo, funguje to“, zeptej se, *kdo to ověřil*. Když ty nebo tvůj test, jsi kentaur. Když jen on sám, posadil sis za soudce Hanse — a kapitola se plete jedině tehdy, když se ti stane, že jsi měl sandbox, pojistku i nezávislý test, a agent *přesto* prošel škodou skrz. V tom případě chci vidět tvou mřížku oprávnění, protože v ní zůstala buňka, co tam neměla být.