XVIII

Proč vznikl git za čtrnáct dní?

Po této kapitole přestaneš mít strach zkusit odvážnou změnu — protože budeš vědět, že každý experiment je vratný. Uvidíš v gitu stroj času, ne administrativu; a než pustíš agenta přepsat půl projektu, uděláš ten jediný pohyb, po kterém tě žádný jeho omyl nemůže stát víc než minutu.

Zálohy jsou pro slabochy. Opravdoví chlapi nahrajou svoje důležitý věci na ftp a nechají zbytek světa, ať si to zrcadlí.

— Linus Torvalds, *„Only wimps use tape backup: real men just upload their important stuff on ftp, and let the rest of the world mirror it ;)“ · e-mail do konference* linux-kernel*, 20. 7. 1996*

Čtrnáct dní, které verzují svět

Na jaře roku 2005 měl největší softwarový projekt světa — jádro Linuxu — problém, jaký nikdo nečekal. Léta se jeho historie spravovala v komerčním nástroji BitKeeper, který firma BitMover poskytovala vývojářům jádra zdarma. V dubnu 2005 firma tuhle bezplatnou licenci vypověděla. Rozbuška byla skoro komická: Andrew Tridgell — autor Samby, zaměstnanec laboratoře OSDL na úplně jiném projektu, člověk, který jádro Linuxu *nevyvíjel* — chtěl umět z BitKeeperu tahat data i svobodnými nástroji. Tak se připojil telnetem na port serveru, napsal `help`, server mu ochotně vypsal seznam příkazů včetně `clone` — a bylo po „reverzním inženýrství“. To stačilo, aby BitMover usoudil, že je porušen zákaz konkurence, a bezplatnou verzi k 1. červenci zrušil.

Linus Torvalds stál před prázdnem. Tisíce vývojářů, milionem řádků kódu, a najednou žádný způsob, jak jejich příspěvky slévat dohromady a udržet přehled, kdo co kdy změnil. Existující svobodné nástroje byly buď příliš pomalé, nebo neuměly to, co Linus potřeboval. A tak udělal věc, která zní jako legenda: přerušil práci na jádře a začal si psát vlastní verzovací systém. 7. dubna 2005 padl první commit — a nástroj byl už tehdy natolik hotový, že *sám sebe verzoval*: git spravoval svůj vlastní vznik. Ten úvodní commit nese

vzkaz „the information manager from hell“. V červnu na něm poprvé jelo celé jádro (verze 2.6.12), a 26. července 2005 — necelé čtyři měsíce po startu — Linus předal údržbu Juniu C. Hamanovi.

Je lákavé číst to jako příběh o géniovi, který za čtrnáct dní stvořil zázrak. Nečti ho tak. Linus nevyhrál rychlostí prstů. Vyhrál, protože *přesně věděl, co chce* — dekádu předtím se rval se správou nejsložitějšího projektu planety a každá ta bolest byla zaplacené kurikulum: věděl na vlastní kůži, co musí verzovací systém umět, co dělá ostatní pomalými a čemu se vyhnout. Napsat git za dva týdny šlo jedině proto, že těch deset let předtím vědělo *proč*. A druhá půlka pointy je ještě méně romantická: git nepřežil díky zakladateli. Linus ho záhy pustil z ruky. Přežil, protože ho patnáct let trpělivě udržuje Junio Hamano — muž, jehož jméno nezná skoro nikdo. Metoda a vytrvalá péče, ne záblesk geniality. Přesně o tom je celá tahle kapitola.

Prameny: konec bezplatné licence BitKeeperu oznámen v dubnu 2005, platnost k 1. 7. 2005; Tridgellova metoda (telnet na port 5000, příkaz `help`*, pak* `clone`*) dle J. Allisona,* LWN *2005. První commit gitu* `e83c516` *„Initial revision of git, the information manager from hell“, 7. 4. 2005, self-hosting; jádro 2.6.12 z 16. 6. 2005; Hamano maintainerem od 26. 7. 2005 (dodnes).*

Tridgell nebyl kernelový vývojář — byl autor Samby na nepříbuzném projektu, a právě proto nebyl vázán licencí, kterou nikdy nepodepsal. Detail, na kterém záleží: rozbuškou nebyl útok, ale nedorozumění o tom, co je „konkurence“.

Dřív než rozebereme, co git dělá a proč, si ujasněme jednu věc, protože bez ní zůstane celá kapitola jen návodem k nástroji. Git není administrativa. Není to otravný krok, který musíš udělat, aby ses dostal k „opravdové práci“. Git je *stroj času* — a stroj času je přesně to, co potřebuješ ve chvíli, kdy pustíš stroj, který za tebe píše kód. O tom bude řeč na konci; nejdřív se podívejme, co to vlastně je.

Paměť projektu: kdo, co, kdy — a hlavně proč

Představ si, že píšeš dlouhý dopis, na kterém ti záleží. Napíšeš odstavec, který se ti líbí, pak ho přepíšeš jinak, pak zjistíš, že ta první verze byla lepší — a ona je pryč. Přepsal jsi ji. Tohle je stav, ve kterém většina lidí tráví práci s počítačem: existuje jen *teď*, poslední uložená verze, a všechno předtím se rozpustilo. Git tuhle past odstraňuje jediným trikem: pamatuje si každý stav, který jsi mu řekl, aby si zapamatoval. Nikdy nic nepřepíše nevratně; jen přidává další a další body v čase, mezi kterými se můžeš libovolně vracet.

Ten bod v čase se jmenuje **commit** — a je to nejdůležitější slovo kapitoly. Commit je uložená pozice ve hře: snímek celého projektu v jednom okamžiku, o kterém jsi řekl „tenhle stav si pamatuj“. Kdykoli později se k němu můžeš vrátit, ať jsi mezitím nadělal cokoli. Ke commitu patří tři věci: *co* se změnilo, *kdo* to změnil a *kdy*. A pak čtvrtá, nejcennější, na kterou většina lidí kašle: *proč*.

commit — uložená pozice ve hře; snímek celého projektu v jednom okamžiku. „Potvrzení změn“ neřekne nikdo živý; commit je commit a sloveso commitnout *je legální.*

V praxi: `git commit -m "proč, ne co"`*. Jeden příkaz, který ten snímek vytvoří a připne k němu vzkaz.*

Ke každému commitu totiž píšeš krátký vzkaz — *commit message*. A tady je skrytá lekce, kterou uvidí málokdo: ten vzkaz nepíšeš pro

počítač. Píšeš ho *dopisem budoucímu sobě*. Za tři měsíce se vrátíš k místu v kódu, které nechápeš, podíváš se, který commit ho vytvořil, a přečteš si, cos tehdy chtěl. Špatný vzkaz zní „opravy“ nebo „změny“ — ten ti neřekne nic. Dobrý vzkaz neříká *co* (to je vidět z diffu), ale *proč*: „přepočet ceny přesunut před slevu, protože jinak sleva počítala z už zlevněné částky“. Tohle je přesně ta *retrieval practice* z kapitoly šest v převleku: napsat vlastními slovy, proč jsi něco udělal, tě nutí to znovu promyslet — a to promyšlení ti utkví v hlavě líp, než kdybys jen kliknul na „uložit“.

→ kap. 6: commit message jako retrieval practice*. Formulovat vlastními slovy* proč *je předepsaný odpor — malá dřina teď, která ti za tři měsíce ušetří hodinu pátrání. Vzkaz „co“ zahodíš; vzkaz „proč“ tě zachrání.*

Dvě další slova, a máš celou abecedu. **Větev** (branch) je *paralelní vesmír*: odbočka, ve které si můžeš dělat, co chceš, aniž bys ohrozil hlavní tok práce. Chceš zkusit riskantní přestavbu? Založ větev, řádí si v ní — a jestli to dopadne špatně, prostě ji zahodíš a hlavní vesmír zůstal netknutý. A když to dopadne dobře, přijde **merge**: dva paralelní vesmíry se postaví před rozhodčího, který posoudí, jak je slít v jeden. Merge je soud — většinou hladký, občas sporný (když dva lidé sáhli na tentýž řádek), ale vždycky vratný.

větev (branch) — paralelní vesmír na experiment: `git branch experiment`*. merge — slití dvou větví v jednu:* `git merge experiment`*. Kde slití skřípe (oba jste sáhli na týž řádek), git tě zavolá jako rozhodčího.*

A tady je ta dobrá zpráva, kvůli které je git nejlepší přítel každého, kdo dnes programuje s asistentem. Pustíš agenta, ať zkusí přepsat celou část projektu podle nového nápadu. Dřív by tě to děsilo — co když to rozbije, co když se v tom ztratíš? S gitem je ten strach zbytečný. Než agenta pustíš, uděláš commit. Ať agent nadělá cokoli, jsi **jeden příkaz** od stavu před ním. A přesně tohle mění hru: **odvaha je levná, když existuje undo pro celé odpoledne.** Můžeš zkoušet divoké věci, protože každý pokus je vratný. Nástroj je opravdu úžasný — ale úžasný je teprve tehdy, když si pod něj postavíš záchrannou síť, ve které visíš, i když spadneš.

Řemeslo: jak s tím zacházet, aby to fungovalo

TEENAGER · MECHANISMUS

Git se dá používat mizerně — jeden obří commit na konci dne se vzkazem „práce“ — a pak je z něj jen záložní kopie. Nebo dobře, a pak je z něj stroj času. Rozdíl je v pár návycích:

```
# 1) ATOMICKÝ COMMIT — jedna myšlenka, jeden
commit.
#    Ne "půl dne práce" naráz, ale "oprava
zaokrouhlení faktury"
#    a zvlášť "přejmenování proměnné". Malé
commity jdou snadno vrátit.
git commit -m "oprava: sleva se počítala z už
zlevněné ceny"

# 2) VĚTEV NA KAŽDÝ EXPERIMENT — hlavní tok
zůstane čistý.
git branch nova-cenotvorba      # založ paralelní
vesmír
git switch nova-cenotvorba      # přepni se do
něj
#   ... experimentuj, klidně to celé rozbij ...
git switch main                 # nefunguje? Vrať
se, vesmír zahoď

# 3) .gitignore — co do historie NEPATŘÍ:
#    hesla, klíče, dočasné soubory, stažené
závislosti.
echo "heslo.txt"  >> .gitignore
echo "node_modules/" >> .gitignore
```

Klíč je pravidlo 1. Když je každý commit *jedna* oddělitelná myšlenka, máš v historii ostrý, čitelný záznam — a když se něco pokazí, vrátíš přesně ten jeden krok, ne celý den práce s ním.

atomický commit — jeden commit = jedna logická změna. Jméno z chemie: atom je dál nedělitelný. Praktický test: umíš commit popsat jednou větou bez „a“? Pak je atomický. „Oprava X a *přejmenování Y“ jsou commity dva.*

.gitignore — seznam, co git ignoruje. Zlaté pravidlo: do historie nikdy hesla a klíče. Co jednou commitneš, zůstává v historii navždy *— smazat to z posledního stavu nestačí, stroj času si to pamatuje.*

Když pracuješ s ostatními — nebo s agentem, který ti generuje kód —, přijde na řadu **code review**: než změna vteče do hlavního toku, někdo ji přečte a schválí. Formálně se to dělá přes *pull request* (PR): „tady je moje větev, prosím o začlenění“. A pro kód, který napsal stroj, platí tohle dvojnásob. Agent nemá stud ani intuici; vygeneruje sebejistě i nesmysl. Review AI výstupu není buzerace navíc — je to jediné místo, kde se do řetězu vrací tvůj úsudek. Nikdy nemergni něco, co jsi nepřečetl, jen proto, že to „vypadá hotově“. Vzpomeň si: „funguje to“ je nejnebezpečnější věta.

pull request (PR) — návrh na začlenění větve; místo, kde proběhne review. Pro AI výstupy je review povinné: model generuje plynule i to, co je špatně (kap. 16), a plynulost není správnost.

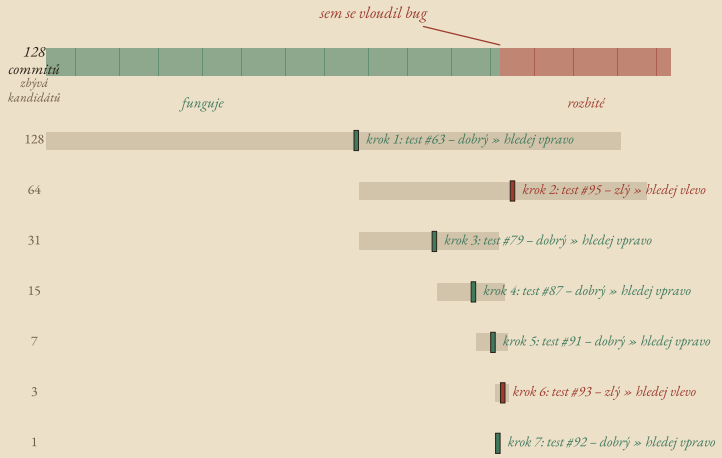
Bisect: binární hledání viníka v čase

TEENAGER · MECHANISMUS

Tohle je nejkrásnější trik gitu a málokdo ho zná. Máš stodvacet osm commitů historie a víš dvě věci: *dnes* je v programu bug a *kdysi dávno* tam nebyl. Který z těch commitů ho zavlekl? Testovat je jeden po druhém od konce by znamenalo klidně sto testů. Git to udělá chytřeji — *půlením*:

```
git bisect start
git bisect bad             # tenhle stav je
rozbitý
git bisect good v1.0       # tenhle byl v pořádku
#   git tě teď přepne doprostřed intervalu; ty
otestuješ a řekneš:
git bisect good            # ...nebo "bad", podle
výsledku
#   git zase půlí; opakuješ, dokud nezbude jediný
commit
```

Každá odpověď „good“ nebo „bad“ zahodí *polovinu* zbylých kandidátů. Ze stodvaceti osmi commitů najdeš viníka na **sedm** testů, protože dvě na sedmou je sto dvacet osm. To je táž myšlenka jako hádání čísla „vyšší — nižší“: kdo půlí, ten neprohledává, ten *vylučuje*.



sedm testů místo sto dvaceti osmi — každý krok půlí čas na polovinu

Nahoře všech 128 commitů v čase: zelené fungují, sanguine jsou rozbité — a někde mezi nimi je hrana, ten jeden commit, co bug zavlekl. Každý řádek dole je jeden krok bisectu: git testuje prostředek zbylého intervalu, tvá odpověď utne půlku, pásmo kandidátů se scvrkává 128 → 64 → 31 → 15 → 7 → 3 → 1. Sedm testů místo sto dvaceti osmi.

Bisect je předzvěst něčeho většího, co přijde v příští kapitole. Všimni si, co při něm děláš: u každého commitu potřebuješ rozhodnout „good, nebo bad?“ — tedy *otestovat*, jestli bug je, nebo není. Když to musíš dělat ručně, je to otrava. Ale jestli máš na ten bug *test* — kousek kódu, který sám řekne „funguje / nefunguje“ —, může bisect běžet úplně sám: `git bisect run` a ten příkaz projede celou historii a vyplivne viníka bez tebe. Testy plus bisect se rovná automatický detektiv, který ti najde,

kde přesně se to pokazilo. Ale o testech až v kapitole devatenáct; zatím si zapamatuj, že bisect je ještě mocnější, když má co spustit.

→ *kap. 19:* `git bisect run <test>` *projde historii sám, když bug umíš otestovat kódem. Bisect (kde) + test (jestli) = automatický detektiv, který najde vinný commit bez tvého klikání.*

Rituál: commit před „přepiš to celé“

Teď to nejpraktičtější z celé kapitoly, a je to jediná věta. **Než pustíš agenta přepsat něco velkého, udělej commit.** Vždycky. Je to reflex, ne rozhodnutí — přesně jako když si zapneš pás dřív, než nastartuješ, ne až v zatáčce.

Proč reflex, a ne „budu opatrný“? Protože ve chvíli, kdy tě chytne nadšení z nového nápadu, opatrnost prohrává. Řekneš agentovi „přepiš celou práci s cenami podle nového návrhu“, on nadšeně přepíše dvě stě řádků, půlka je skvělá, půlka rozbila tři jiné věci — a ty teď nevíš, co bylo předtím, protože to přepsal *přes* tvůj poslední stav. Kdybys commitnul, jsi jeden `git switch` od původního kódu a můžeš si v klidu vytáhnout jen ty dobré kusy. Nekomitnul-lis, trávíš večer archeologií vlastního projektu.

→ *kap. 8: tohle je* pakt *— smlouva s budoucím já, uzavřená za střízliva. „Commit před experimentem“ patří do stejné rodiny jako „predikce zapsaná před simulací“ z kap. 14. Vůle v reálném čase prohrává; pakt vyhrává.*

Vidíš, že tenhle rituál není o gitu. Je to *pakt* z kapitoly osm, ten samý, který jsme potkali u Odyssea přivázaného ke stěžni: rozhodnutí učiněné *předem*, dokud jsi klidný, které tě chrání ve chvíli, kdy bys sám sebe nezvládl. Commit před experimentem je Odysseovo lano přeložené do jednoho příkazu. A je to nejlevnější pakt, jaký existuje: stojí tě dvě vteřiny a vrací ti svobodu zkoušet cokoli.

DEMO

`lesk-a-bida-vibecodingu.gaussalgo.com/d/bisect`

Bisect hra: schoval jsem bug do jednoho ze 128 commitů. Najdi ho — a sleduj, jak ti každá odpověď „good/bad“ půlí zbytek. Zvládneš to na sedm kroků?

Věž: proč jednomu hashi můžeš věřit celou historii

EINSTEIN · MATEMATIKA — přeskočitelné

Obsahová adresace. Git si každý objekt — soubor, adresář, commit — nepojmenovává podle toho, kde leží, ale podle toho, *co obsahuje*. Vezme obsah, prožene ho kryptografickou hašovací funkcí a výsledek (dřív SHA-1, dnes se přechází na SHA-256) je jeho jméno:

$$\text{jméno}(x) = \text{SHA}(x).$$

Dvě vlastnosti té funkce nesou všechno ostatní. Za prvé je *deterministická*: stejný obsah dá vždy stejné jméno — a tedy dva identické soubory git uloží jen jednou. Za druhé je *citlivá na každý bit*: změň jediné písmeno a jméno se celé rozsype v nepředvídatelnou novou hodnotu. Jméno objektu tak *je* otiskem jeho obsahu. Nemůžeš změnit obsah a nechat jméno — a přesně na tom stojí to, co přijde teď.

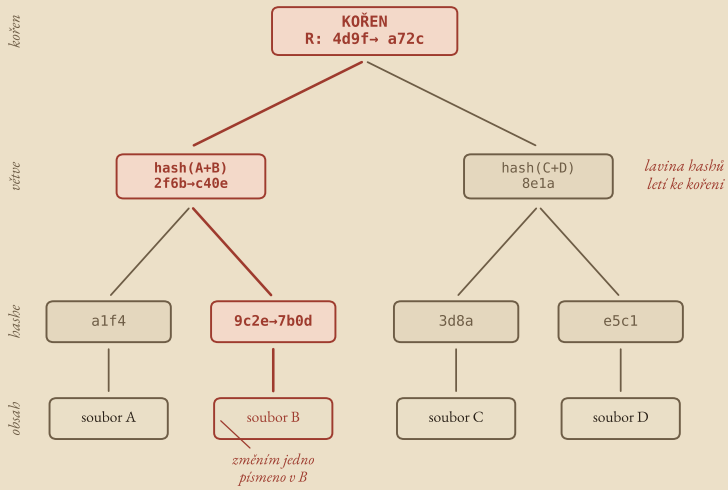
hash (haš) — otisk obsahu: krátký řetězec, který kryptografická funkce spočítá z libovolně velkého vstupu. Vlastnosti: stejný vstup → stejný otisk; jakákoli změna vstupu → úplně jiný otisk; a z otisku nelze vstup dopočítat. Git jím objekty pojmenovává, ne jen zabezpečuje.

EINSTEIN · MATEMATIKA — přeskočitelné

Merkle DAG: proč hash kořene jistí celou historii. Objekty gitu neleží vedle sebe nezávisle — jsou navlečené do stromu. Adresář (v gitu „strom“) obsahuje jména svých souborů; a protože jméno je hash obsahu, hash adresáře se počítá z hashů jeho dětí. Nad nimi je commit, jehož hash se počítá mimo jiné z hashe kořenového stromu *a z hashe předchozího commitu*. Vznikne tak *Merkleovský* orientovaný acyklický graf (DAG), kde každý uzel svým hashem zapečetuje všechno pod sebou:

$$h_{\text{uzel}} = \text{SHA}(h_{\text{dítě 1}} \parallel h_{\text{dítě 2}} \parallel \cdots).$$

Důsledek je mimořádný. Změň jediné písmeno v jediném souboru — a jeho hash se změní; tím se změní hash adresáře nad ním; tím hash commitu; a protože každý commit drží hash toho předchozího, změní se i hashe *všech commitů, které po něm následovaly*. Lavina se valí od listu ke kořeni a dál celou historií. Jinými slovy: *jediný hash posledního commitu* zapečetuje bit po bitu úplně všechno, co kdy v projektu bylo. Kdo zná ten jeden otisk, pozná jakoukoli změnu kdekoli v minulosti — protože by musela změnit i ten otisk.



Hrdinský graf kapitoly. Dole obsah souborů, nad ním jejich hashe, pak hashe větví, nahoře kořen. Změním jedno písmeno v souboru B (sanguine cesta): přepočítá se hash B, nad ním hash větve A+B, a nakonec kořen. Nedotčená strana (C, D) zůstává beze změny. Jeden otisk kořene tak hlídá integritu celé *struktury — nejde zfalšovat minulost, aniž by se změnil.*

EINSTEIN · MATEMATIKA — přeskočitelné

Tři stromy jako stavový automat. Git nepracuje s jedním stavem, ale se třemi, a přechody mezi nimi jsou celá jeho mechanika:

1. **working tree** — soubory, jak je právě vidíš a edituješ na disku;
2. **index** (staging) — „nákupní košík“: co z working tree půjde do příštího commitu;
3. **HEAD** — poslední commit, špička větve, na které stojíš.

Dva příkazy jsou přechodové funkce mezi nimi: `git add` přesune změny z working tree do indexu; `git commit` zapečetí index do nového commitu a posune HEAD na něj. Formálně je to konečný automat: stav je trojice (W, I, H) a příkazy jsou hrany, které ji mění. `git status` ti jen říká, jak daleko se ty tři stromy rozešly. Jakmile jednou uvidíš git jako automat se třemi stavy, přestane být sbírkou záhadných příkazů — každý z nich jen posouvá obsah mezi W, I a H.

→ kap. 19: tři stromy jsou příbuzné se smyčkou červená–zelená–refaktor. Working tree je „píšu“, index „chystám k důkazu“, commit „zapečetěno“ — disciplína malých, ověřených kroků, tatáž jako u testů.

EINSTEIN · MATEMATIKA — přeskočitelné

Distribuovanost bez serveru. Protože identitu objektu určuje jeho obsah, ne jeho umístění, nepotřebuje git žádnou centrální autoritu, která by přidělovala čísla verzí. Každá kopie repozitáře je *úplná*: nese celou historii i všechny hashe. Dvě kopie jsou identické právě tehdy, když se shodují v hashi posledního commitu — jediné číslo rozhodne o shodě libovolně dlouhé historie. Tady se uzavírá kruh s podobenstvím: přesně tuhle vlastnost Linus potřeboval pro tisíce vývojářů roztroušených po světě, bez jediného serveru, kterému by museli věřit. Obsahová adresace není účetní trik — je to matematický základ, na kterém stojí spolupráce bez centra. A je to, mimochodem, tatáž myšlenka „nech to zrcadlit celý svět" z epigrafu, jen dotažená do konce: když identitu nese obsah, je jedno, kdo drží kopii — pravost si každý ověří sám.

Proto se přechází ze SHA-1 na SHA-256: u SHA-1 se roku 2017 podařilo vyrobit dva různé obsahy se stejným otiskem (kolize). Když jméno je *zárukou obsahu, musí být kolize prakticky nemožná — jinak by šlo podstrčit jiný obsah pod stejným jménem a lavina by se nespustila.*

Co si odnést z gitu

Poskládej to dohromady. Git vznikl za čtrnáct dní ne proto, že by byl Linus génius, který vytáhl zázrak z rukávu, ale protože dekáda bolesti se správou jádra byla zaplacené kurikulum — přesně věděl, co chce. A přežil ne díky zakladateli, ale díky patnácti letům Hamanovy trpělivé údržby. Metoda a péče, ne záblesk. To je erb celé téhle knihy, promítnutý do jednoho nástroje.

A ten nástroj ti dává tři věci. *Paměť:* commit je uložená pozice, ke které se vždycky vrátíš, a jeho vzkaz je dopis budoucímu sobě — piš do něj *proč*. *Odvahu:* větev je paralelní vesmír, ve kterém smíš cokoli rozbít, protože ho zahodíš beze stopy — a proto je experiment s agentem levný, ne děsivý. A *jistotu:* jediný hash kořene zapečetěuje bit po bitu celou historii, takže minulost nejde tiše přepsat. Git je stroj času, ne administrativa. A ve světě, kde ti kód píše stroj, který se nestydí a negratuluje si k nesmyslu, je stroj času to první, co si pod tu spolupráci postavíš — dřív, než pustíš první prompt. Verzování je i vstupní podmínka všeho, co přijde dál: bez commitu není co testovat, co nasadit, co vrátit, když se produkce zadrhne.

→ kap. 21: nasazení začíná commitem. *Linka* push → testy → build → nasazení *bere jako vstup právě commit; co není zverzované, nejde spustit do produkce. Git je první článek řetězu, který vede až k běžící službě.*

„Jak bych poznal, že se pletu?“

Konkrétní test k této kapitole je jednoduchý a máš ho v ruce pokaždé, když sedneš k práci s asistentem. Než pustíš agenta přepsat něco velkého, zeptej se sám sebe: *udělal jsem teď commit?* A hned druhou otázku, tvrdší: *kdyby to agent zkazil, kolik práce ztratím — odpoledne, nebo minutu?* Když je odpověď „minutu“, jsi přivázaný ke stěžni a můžeš pustit píseň nahlas: zkoušej divoké věci, protože každá je vratná. Když je odpověď „odpoledne“, nepustil jsi agenta z řetězu odvážně — pustil jsi ho *bezbranně*, a kapitola tě právě varovala. Kapitola se plete jen tehdy, když se ti stane, že jsi commitnul, agent to rozbil, a vrátit se *přesto* nešlo — a v tom případě chci vidět tvůj repozitář, protože to by znamenalo, že stroj času přestal být strojem času.