

„Novinka“, kterou děláme už 50 let: Co přichází po generování kódu?

Jak jsem nechal AI agenta přepsat sám sebe, proč generování kódu není novinka a co budou dělat programátoři budoucnosti.

Nechal jsem AI agenta vyvíjet sám sebe. A pak se to celé zamotalo.

Vyvíjíme Promptbook (github.com/webgptorg/promptbook). Je to mimo jiné AI coding agent. A protože vyvíjíme coding agenta, napadlo nás něco přirozeného: co kdyby Promptbook vyvíjel sám sebe?

Začalo to nevinně. Přidej CLI flag. Oprav bug. Přidej podporu dalšího modelu. Postupně jsme mu svěřovali celé funkce a moduly. A potom jsme udělali další krok: přestali jsme mu úkoly zadávat.

Promptbook už dokázal rozpoznávat práci, vytvářet úkoly, vykonávat je a pokračovat dál. Ne hodinu. Dny nebo týdny. Prakticky nepřetržitě. A fungovalo to. Jenže zatímco člověk vytváří technický dluh osm hodin denně, autonomní agent na něm může pracovat čtyřicet hodin denně.

Nové funkce. Nové abstrakce. Další závislosti. Další testy. Opravy testů. Opravy oprav testů. Pomalejší build. Složitější projekt. Merge konflikt. Další konflikt. Až jsem poprvé v životě řešil merge tří různých commitů najednou.

Tři větve změn. Jeden stále zamotanější projekt. ... A DOST.

Nechtěl jsem další týden rozplétat technický dluh. Chtěl jsem zahodit implementaci a začít znovu.

Starý Promptbook vytvořil nový Promptbook

Nechal jsem ze stávající aplikace vytvořit specifikaci. Ne seznam tříd ani kopii architektury, ale lidsky srozumitelný popis toho, co má software dělat – a co už dělat nemusí. Výsledkem byl relativně stručný Markdown dokument.

V Promptbooku máme také Book, vlastní jazyk pro definování agentů, jejich cílů, pravidel a specifikací: jeden z pokusů o programování na vyšší úrovni abstrakce. Pro tento experiment ale není konkrétní syntaxe rozhodující.

Spustil jsem starý, zašpagetizovaný Promptbook nad novým projektem. Neměl přístup ke svému původnímu zdrojovému kódu. Dostal specifikaci a zadání: vytvoř podle ní aplikaci. A on ji vytvořil. Aniž by věděl, že píše vlastního nástupce.

Podle mé zkušenosti vznikl software s prakticky stejnou požadovanou funkcionalitou, přibližně 20× rychlejší a 100× menší. Nejde o nezávislý benchmark ani důkaz úplné ekvivalence, ale jako experiment to bylo přesvědčivé.

Nejdřív jsem si říkal: tohle je úplně nový způsob vývoje. A pak mi došlo, že princip známe už desítky let.

Vynalezl jsem něco nového. Nebo snad ne?

Programátor napíše C. Kompilátor vytvoří nižší reprezentaci a nakonec strojové instrukce. Programátor obvykle neřeší konkrétní registry nebo instrukce. Změní C, spustí kompilaci a výstup klidně celý zahodí. Může použít novou verzi GCC a dostat jiný strojový kód. Nikomu to nepřipadá revoluční. Zdroj pravdy je o úroveň výš.

C → assembly → machine code. A v mém experimentu: specifikace → TypeScript → JavaScript → machine code.

Pro šťouraly: ano, překlad TypeScriptu do JavaScriptu se běžně označuje jako transpilace. Skutečný řetězec navíc obsahuje další mezikroky. Pointa je v posunu hranice mezi vyšší reprezentací a artefaktem.

To, že starý Promptbook vytvořil svou novou verzi, připomíná bootstrapping kompilátoru. GCC může kompilovat nové GCC, aniž by potřebovalo vědět, že kompiluje samo sebe. Není to technická ekvivalence – AI interpretuje mnohem volnějším zadáním – ale analogie je silná.

Ještě před zdrojovým kódem: lidé a fyzické stroje

Když se dnes řekne programování, vybaví se nám text v editoru. Ale tento obraz není samozřejmý. Slovo computer kdysi označovalo člověka, který prováděl výpočty. Algoritmus mohl být napsaný na papíře, zatímco jeho vykonavatelem byl člověk. Ještě před symbolickými jazyky tak existoval rozdíl mezi popisem postupu a jeho vykonáním, jen nebyl zabalený do dnešní softwarové infrastruktury.

U některých raných elektronických strojů se výpočetní úloha nastavovala fyzickým propojováním kabelů a přepínačů. Program byl do značné míry konfigurací stroje. Není přesné říct, že se vždy „předrátovávaly elektronky“: konkrétní technologie se lišily. Podstatné je, že program ještě nebyl samozřejmým přenosným textovým artefaktem nezávislým na konkrétním stroji.

Pak přišel strojový kód a symbolické zápisy instrukcí. Assembly dal lidem názvy instrukcí a symbolické adresy; assembler je převedl do reprezentace, které rozuměl procesor.

Najednou existoval textový program a mechanismus jeho převodu na vykonatelné instrukce. První velké oddělení: program už nebyl jen fyzickým nastavením stroje.

Vyšší jazyky: když přestaneme popisovat instrukce

Fortran, COBOL, C a další jazyky posunuly hranici ještě výš. Místo toho, aby člověk rozhodoval o každém registru, začal zapisovat výrazy, funkce, datové struktury a algoritmy. Překladač měl za úkol rozhodnout, jak takový popis realizovat na konkrétním procesoru.

Představme si jednoduché sčítání dvou čísel v C. Výraz `a + b` není přímým příkazem použít určitý registr a konkrétní strojovou instrukci. Je to popis operace v jazyce, který má definovanou sémantiku. Překladač může zvolit různé implementace podle architektury, optimalizací a okolního

programu. A přesto programátor nadále pracuje s jedním zdrojovým zápisem.

Tady se odehrála zásadní výměna: část nízkoúrovňové kontroly jsme dobrovolně odevzdali, abychom mohli myslet ve větších celcích. V běžném programování nepovažujeme tuto ztrátu kontroly za selhání. Považujeme ji za výhodu abstrakce. Nikdo nechce do každé funkce v C ručně přepisovat stovky instrukcí assembly jen proto, aby měl „všechno pod kontrolou“.

Stejný posun najdeme mimo programovací jazyky. SQL vyjadřuje, jaká data požadujeme, a databázový optimalizátor volí plán dotazu. CSS vyjadřuje pravidla vzhledu, nikoli posloupnost pixelových operací grafické karty. Vyšší reprezentace nechává nižší vrstvě prostor pro realizaci. Abstrakce tedy není jen pohodlnější syntaxe; je to delegace rozhodování.

Druhá linie dějin: jak spolu vrstvy komunikují

Vedle historie jazyků běží druhá, stejně zajímavá historie: historie vztahu mezi vyšší a nižší vrstvou. Nejjednodušším modelem je překlad. Napíšu zdroj, přeložím ho a vykonávám výsledek. Ale už dávno existují také interpretery, které vyšší reprezentaci vykonávají za běhu, a virtuální stroje, které kombinují různé mezistupně.

Interpretace nevznikla až s Pythonem a JIT nevznikl až s Javou. Tyto techniky mají dlouhou a překrývající se historii. Moderní JavaScriptový engine může nejprve vytvářet bytecode, některé části interpretovat a často používané cesty později optimalizovat do strojového kódu. Podobně různé implementace Javy, C# nebo Pythonu volí různé kombinace překladů a vykonávání.

Důležité není přiřadit každému jazyku jednu nálepku. Důležité je pochopit, že s vyšší abstrakcí přibývaly možnosti, kdy a jak ji převést do nižší. Kompilace předem, interpretace za běhu, just-in-time materializace podle aktuální potřeby: všechny tyto vztahy mohou existovat vedle sebe.

Nová vrstva: lidský záměr jako program

Dnes možná přidáváme další úroveň. Člověk už nemusí začínat TypeScriptem. Může začínat popisem požadovaného chování, omezení a cíle. Model pak vytváří konkrétní implementaci. Přirozený jazyk samozřejmě nemá pevnou formální sémantiku C a jedna specifikace může připouštět mnoho správných implementací. Proto je nutné výsledek ověřovat. Ale jako pracovní jazyk, v němž člověk programuje na vyšší úrovni, už fungovat může.

V této perspektivě může být TypeScript pro autora specifikace tím, čím je assembly pro autora programu v C. Někdy ho čte, ladí a opravuje, ale jeho hlavním vyjádřením záměru je vyšší vrstva. Když programátor v C přeloží projekt, neříká obvykle „vygeneroval jsem strojový kód“. Říká „napsal jsem program v C“. Možná jednou nebudeme říkat „AI mi vygenerovala aplikaci“, ale „programoval jsem ve specifikaci“.

Tím se ovšem mění i to, co od specifikace očekáváme. Pokud ji zahltneme detaily interní implementace, nevyužíváme vyšší abstrakci. Pouze jsme nižší kód přepsali do slov. Specifikace má vymezit důležité invarianty produktu, veřejné kontrakty a požadované výsledky. Nemá nahrazovat každou funkci a každý interní endpoint.

3 důvody, proč AI není klasický kompilátor. Nebo možná je?

1. „Kompilátor je deterministický. AI přece hází kostkou.“

Když spustím GCC nad stejným zdrojovým kódem, očekávám stejný výsledek. Když dám AI stejnou specifikaci, může napsat dva odlišné programy. Na první pohled zásadní rozdíl. Jenže výstup GCC není určen pouze textem programu: závisí na verzi překladače, optimalizacích, cílové architektuře i sestavovacím prostředí. Kompilátory navíc obsahují chyby a jejich chování se v čase mění. To samozřejmě nesmazává zásadní rozdíl: jazyk C má definovanou sémantiku, zatímco přirozený jazyk připouští mnoho výkladů.

Ale ani náhodnost generativních modelů není nějaká metafyzická vlastnost AI. Sampling je mechanismus výběru; v některých prostředích jej lze omezit, nastavit seed, fixovat model a běhovou konfiguraci. Seed sám o sobě nezaručuje identické výsledky napříč hardwarem či službami, a reprodukovatelný výstup ještě neznamená správný výstup. Přesto je rozdíl méně černobílý než jednoduché „deterministické versus náhodné“. Je to rozdíl v míře předvídatelnosti, v garancích a v tom, co všechno musíme při reprodukci zafixovat. A tyto hranice se mohou měnit.

2. „Kompilace je levná. AI generování je celý vývojový sprint.“

Dnešní překladač běžně přeloží malý program během okamžiku. Agent může nad specifikací pracovat hodiny, spouštět testy, opravovat chyby a spotřebovat významné prostředky. To je skutečný rozdíl. Ale možná jej přeceňujeme jen proto, že žijeme v době rudimentárních „kompilátorů“ pro přirozený jazyk.

Klasické překladače prošly desetiletími optimalizací, inkrementálních buildů, cacheování a paralelizace. První překladače navíc vznikaly už v 50. letech, kdy byly výpočetní zdroje nesrovnatelně vzácnější. Ani dnes není sestavení velkého projektu vždy levné. Nelze z toho vyvodit, že AI generování někdy dosáhne ceny kompilace C: řeší otevřenější problém a musí hledat, ověřovat a rozhodovat. Ale současná pomalost není důkazem, že jde o principiálně nesouvisející proces. Může jít zčásti o vlastnost současného stupně technologie.

3. „Vygenerovaný kód se neupravuje. Jenže s AI jej upravujeme pořád.“

U C se typicky vracíme ke zdrojovému souboru, nikoli k výslednému assembly. U AI často otevřeme vygenerovaný TypeScript a pokračujeme přímo v něm. Ani tato hranice ale není absolutní. V C existuje inline assembly, intrinsics, pragmata a další způsoby, jak z vyšší vrstvy ovlivnit nižší reprezentaci. A opačně: i v AI lze zafixovat veřejné rozhraní, referenční implementaci nebo test a nadále generovat ostatní části.

Rozdíl však nespočívá pouze v tom, zda se nižší vrstvy někdy dotkneme. Podstatné je, **jakou autoritu jí přiznáme**. Musíme všechno popsat v horní specifikaci? Můžeme něco v nižší vrstvě zachovat, aniž bychom celou tuto vrstvu povýšili na nedotknutelný zdroj pravdy? Právě tato otázka nás dovede ke čtvrtému vztahu mezi abstrakcemi.

Specifikace na dvě stě stran může být špatná specifikace

Vidím trend vytvářet stále delší specifikace. Padesát stran. Sto. Dvě stě. Někdy je celé generuje AI, jako by počet stránek dokazoval kvalitu. Podle mě je to často obráceně.

Pokud specifikace popisuje každou interní funkci, endpoint a implementační rozhodnutí, možná jsme nevytvořili vyšší abstrakci. Jen jsme přepsali zdrojový kód do angličtiny.

Každá úroveň abstrakce má popisovat to, co jí přísluší. V C nechci ručně určovat registry procesoru. V TypeScriptu nechci ručně spravovat paměť. V produktové specifikaci nechci diktovat interní endpointy, pokud nejsou samy důležitým kontraktem. Jeden příklad

interfaceu může být užitečný, ale systematické přenášení implementace do specifikace popírá smysl abstrakce.

Často za tím stojí strach: chci mít všechno pod kontrolou. Jenže vyšší jazyky jsme vytvořili právě proto, abychom všechno pod kontrolou mít nemuseli. Dobrá specifikace přesně vymezí to důležité a zbytek deleguje.

A co když už v nižší vrstvě existuje něco cenného, co nechci ztratit?

3 známé vztahy mezi vyšší a nižší abstrakcí

Na dějinách programování mě zajímají dvě paralelní linie. První je růst abstrakce: od fyzického nastavení stroje přes instrukce a vyšší jazyky až k popisu lidského záměru. Druhá je proměna vztahu mezi těmito vrstvami. Není totiž jediný způsob, jak se z vyšší reprezentace stane nižší. A právě ve světě AI začínají být rozdíly mezi těmito způsoby překvapivě praktické.

Než přidáme něco nového, stojí za to projít tři známé modely. Nejsou to výhradní kategorie jazyků: jeden runtime může používat všechny tři. U každého se nejdřív podívejme na tradiční programování a teprve potom na jeho AI analogii.

1. Kompilace: přeložit vyšší zápis do nižší reprezentace

Klasický svět. Napíšu C a překladač vytvoří strojový program, případně cestou assembly. Nebo napíšu TypeScript a nástroj jej převede do JavaScriptu (pro šfouraly: obvykle tomu říkáme transpilace). Autor pracuje s vyšší reprezentací, zatímco výsledný program běží na reprezentaci nižší. Změním zdroj, sestavím nový výsledek; starý výstup mohu zahodit.

AI svět. Napíšu specifikaci aplikace v přirozeném jazyce. Agent vytvoří TypeScript, testy a další soubory. Po ověření spouštím výslednou aplikaci bez účasti původního generátoru. To je čistá, nejpřímočařejší podoba spec-driven developmentu: specifikace plní roli zdroje, kód roli generovaného výstupu. Rozdíl proti GCC je ve slabších sémantických zárukách; proto je nutné generovaný software testovat a ověřovat.

Specifikace → AI build → TypeScript → JavaScript → vykonávání

2. Interpretace: vykonávat vyšší zápis přímo

Klasický svět. Interpret čte program a vykonává jeho operace během běhu. Pěkným didaktickým příkladem je jednoduchý interpret jazyka BASIC nebo shell, který postupně vyhodnocuje příkazy. U moderního Pythonu se běžně nejprve vytváří bytecode a ten vykonává virtuální stroj; ani „interpretovaný jazyk“ proto neznamena absenci mezireprezentace. Jde o způsob vykonávání, nikoli neměnnou vlastnost syntaxe.

AI svět. Mám agenta pro zákaznické e-maily a zadám mu: „Pročti nové zprávy, na běžné dotazy odpověz, složité případy předej člověku.“ Nevznikne nutně žádná trvalá e-mailová aplikace. Agent instrukci průběžně vyhodnocuje podle každé příchozí zprávy, volá API a nástroje, případně si na okamžik vytvoří shellový příkaz nebo malý JavaScript, který po dokončení zahodí. Uživatel spravuje instrukci, nikoli její trvale generovaný zdroják. To je interpretace přirozenojazyčného programu v praktickém smyslu.

Instrukce + příchozí e-mail → agent → volání nástrojů → odpověď

3. Just-in-time: materializovat nižší vrstvu až ve chvíli potřeby

Klasický svět. Java je známý příklad, nikoli vynálezce JIT. Java Virtual Machine může začít vykonáváním bytecodu a při opakovaném používání určité části programu ji přeložit do optimalizovaného strojového kódu. Profilování odhalí „horkou“ cestu a runtime investuje do kompilace, protože další běhy budou levnější. Historie interpretace a JIT sahá před Javu; podstatný je ekonomický princip: něco dražšího připravím teprve tehdy, když se mi to vyplatí.

AI svět. Agent zpracovává data a zjistí, že pořad dokola čistí stejný typ CSV. Namísto opakovaného uvažování nad každým řádkem si vytvoří skript a uloží jej pro další běhy. Nebo si vytvoří pomocnou knihovnu či algoritmus. Uživatel požádá o porovnání obchodních scénářů a agent na požádání sestaví **generative UI**: interaktivní graf a ovládací prvky pro právě tento úkol. Tyto aplikace a utility nevznikají předem; materializují se podle aktuální potřeby. Mohou žít ve skryté pracovní složce runtime, aniž by uživatel jejich zdrojový kód vůbec viděl.

Když agent potřebuje spát: JIT jako budík

Zajímavým příkladem jsou dlouhodobě běžící agenti. Představme si agenta, který má reagovat na příchozí e-mail nebo změnu webové stránky. Naivní řešení je probudit každou minutu celý jazykový model a nechat jej znovu kontrolovat, zda se něco nestalo. Funguje to, ale spotřebovává tokeny a výpočetní prostředky i tehdy, když není co dělat.

Agent místo toho může **vygenerovat malý nízkourovňový hlídač**: cron úlohu, webhook handler, skript sledující schránku nebo heuristiku pro změnu stránky. Sám se uspí a nechá levný skript sledovat konkrétní podmínku. Jakmile nastane událost, skript probudí dražšího agenta. Je to podobné hardwarovým přerušením a nízkopříkonovým mechanismům, které probudí hlavní procesor až při relevantní události.

V Promptbooku je takovým vyšším zadáním například dlouhodobý goal agenta. Některé podmínky probuzení lze odvodit a materializovat jako jednoduchý skript, jiné vyžadují složitější posuzování samotným modelem. To, zda se vyplatí něco převést do klasického kódu, je optimalizační rozhodnutí. Stejně jako u JIT v Javě investujeme jednou do nižší reprezentace, abychom ušetřili náklady při opakovaném běhu. Analogii nepovažují za totožnou implementaci: JVM optimalizuje již

definovaný program, zatímco agent může teprve navrhovat hlídací algoritmus.

Goal agenta → jednorázově vytvořený hlídač → levné čekání
↓ událost
probuzení agenta

A právě tady vzniká další otázka. Co když uživatel ten vygenerovaný skript, komponentu nebo rozhraní chce vidět a některé jeho vlastnosti chce zachovat? Tím se dostáváme za hranici skrytého JIT artefaktu.

Ale počkat. Nemá AI ještě 4. vztah?

Anchoring: nižší vrstva získá omezenou autoritu

A teď to, co podle mě tyto tři modely doplňuje. Agent vygeneroval UI. Je krásné a vyladěné. Nebo vytvořil API, na kterém už závisejí zákazníci. Nebo algoritmus, který konečně správně řeší obtížný okrajový případ. Při příštím generování nechci zahodit úplně všechno, ale nechci ani prohlásit celý dosavadní zdrojový kód za nedotknutelný.

Vyberu proto konkrétní invarianty a ukotvím je. Někdy je invariantem přesný soubor. Jindy veřejné rozhraní, vizuální identita, chování nebo výsledek regresního testu. Vygenerovaný artefakt se nestane nejvyšším zdrojem pravdy, ale získá omezenou autoritu pro další transformace. To je anchoring: selektivní zachování hodnoty v nižší vrstvě.

Kompilace se ptá: „Co vytvořit z vyššího zadání?“ Interpretace: „Jak zadání vykonat teď?“ JIT: „Co se vyplatí vytvořit až v okamžiku potřeby?“ Anchoring přidává otázku: **„Co z již vytvořeného musí přežít příští změnu?“** Právě proto ho nevnímám jen jako další způsob generování, ale jako čtvrtý vztah mezi abstrakčními vrstvami. Není to tvrzení, že před AI neexistovaly invarianty, referenční implementace nebo částečné regenerace. Nové je především to, jak zásadní roli mohou tyto hranice získat v generativním vývoji.

Specifikace + vybrané anchory + nový úkol
↓
AI transformace
↓
Nový kód, zachované invarianty

Všechny čtyři vztahy mohou existovat v jednom projektu současně. Agent může interpretovat dlouhodobý cíl, JIT vytvořit utilitu, ze specifikace zkompileovat novou aplikaci a přitom respektovat ukotvené API. Nejde o čtyři konkurenční typy AI, ale o čtyři způsoby, jak vyšší a nižší reprezentace spolupracují.

Anchoring: když vygenerovaný artefakt není jen odpad

Představte si, že AI vytvoří webovou aplikaci. Má nádhernou komponentu: přesný design, animace a vyladěné chování. Chcete přidat jinou funkci. Můžete všechno znovu vygenerovat a doufat, že komponenta dopadne stejně. Nebo ponechat celý projekt a jen přidávat kód. První cesta ztrácí detaily. Druhá konzervuje technický dluh.

Potřebujeme třetí možnost: ukotvit jen to, na čem záleží.

Anchoring je princip, při kterém určitým částem vygenerovaného nebo dříve existujícího artefaktu přiznáme hodnotu, kterou chceme zachovat při dalších transformacích. Nejsou primárním zdrojem pravdy. Ale nejsou ani zcela zahoditelné.

Můžeme mluvit o code anchoringu, UI anchoringu, design anchoringu, behavior anchoringu nebo feature anchoringu. Kotvou může být komponenta, interakce, vizuální detail, API kontrakt nebo charakteristické chování.

Ukotvení nemusí znamenat zachování identických bajtů. Design může přežít novou implementaci. Chování může být ukotveno testem. Kód konkrétním modulem. Není to jen cache, Git commit ani obyčejný build artefakt: je to rozhodnutí, jakou autoritu má mít část výsledku při budoucí změně.

Specifikace říká, co má vzniknout. Anchory říkají, co z toho, co už vzniklo, stojí za zachování.

Ani zdroj pravdy, ani zahoditelný artefakt

V tradičním buildu bývá hranice relativně jasná: zdrojový kód uchováváme, build výstup lze znovu vytvořit. V AI-native vývoji vzniká spektrum. Některé části jsou nejvyšší specifikací. Jiné lze regenerovat. A některé jsou ukotvené: mají menší autoritu než specifikace, ale stále obsahují hodnotu.

Anchoring proto není jen jiné jméno pro iterativní generování. Iterovat lze i nad zcela zahoditelným výstupem. Anchoring popisuje selektivní kontinuitu. A platí i pro starší projekty, které žádnou původní specifikaci neměly.

Co přesně může být kotva?

Začněme vizuálním návrhem. Agent vytvoří dashboard. Specifikace říká, že má být přehledný, ale konkrétní typografie, proporce a animace se podaří až při třetím pokusu. Můžu ukotvit celý návrh, konkrétní komponentu nebo jen vybrané vizuální invarianty. Neříkám tím, že každý CSS řádek je nedotknutelný. Říkám, že při příštím generování nechci přijít o určitý výsledek.

Druhý příklad je veřejné API. Specifikace produktu může říkat „zákazníci mohou získat faktury přes API“. Ale skutečný systém už používá konkrétní endpointy, názvy polí a chování chybových odpovědí. Jestli na nich závisejí externí klienti, mají hodnotu i tehdy, když by nová implementace mohla být elegantnější. Kotvou není nutně starý controller; kotvou je kompatibilní kontrakt.

Třetí příklad je algoritmus. Model vygeneruje optimalizaci, která po dlouhém ladění zvládá specifickou okrajovou situaci. Kdybych ji popsal do vyšší specifikace včetně každé implementační podrobnosti, zničil bych smysl abstrakce. Můžu ale zachovat referenční implementaci, sadu regresních testů nebo vlastnost, kterou má nová implementace prokazatelně splnit.

Čtvrtý příklad je obsah. Agent připraví tón komunikace značky, ilustraci, video nebo konkrétní onboardingovou sekvenci. Výsledek je částečně generovaný, ale po schválení se stává cenným. Nechci jej při každém novém běhu přepsat jen proto, že jej lze znovu vytvořit. Tady mluvíme o design anchoringu, content anchoringu nebo behavior anchoringu, nikoli pouze o code anchoringu.

Pátý příklad je starší projekt, který nikdy žádnou vyšší specifikaci neměl. Jeho zdrojový kód může být jediným dochovaným záznamem určitých rozhodnutí. Teprve postupně z něj extrahuji vyšší záměr a rozlišuji, které existující vlastnosti jsou závazné a které pouze náhodně přežily. Anchoring tedy nemusí začínat vygenerováním artefaktu. Může být také způsobem, jak z již existujícího systému selektivně převzít hodnotu.

Klíčové slovo je **invariant**. Někdy jím bude přesná podoba souboru. Jindy rozhraní, chování nebo výsledek, který může přežít kompletní přepis implementace. Kotva by proto měla vyjadřovat rozsah, důvod, způsob ověření a případně pravidla uvolnění. Jinak je

„ukotvení“ jen hezcí jméno pro strach cokoliv změnit.

A co na to Git?

Dnešní Git výborně verzujeme soubory, ale jejich význam pro regeneraci typicky nezaznamenává. V praxi často rozlišujeme commitnutý zdroj a ignorovaný build output, jenže tato dvojice není úplná: i vygenerovaný lockfile, schválený design nebo referenční komponenta mohou mít důvod zůstat v repozitáři. Samotný `.gitignore` navíc nerozhoduje, co je zdroj pravdy.

Možná budeme potřebovat třetí **sémantickou** kategorii: anchor. Soubor nebo vlastnost, kterou lze technicky přegenerovat, ale její vybrané invarianty mají při další transformaci zůstat zachovány. Nemusí jít o novou funkci jádra Gitu; klidně to mohou být metadata, pravidla agentů a validační testy. Podstatné je, aby verzování jednou umělo zachytit nejen co uchováváme, ale i proč to nechceme ztratit.

Končí věk frameworků?

Knihovna a framework jsou zvláštní případy abstrakce. Často jsou napsané ve stejném jazyce jako aplikace, ale nabízejí vyšší pojmy a hotová rozhodnutí: místo ruční práce s HTTP requesty používám webový framework, místo přímé manipulace s DOM komponentovou knihovnu. Nejde tedy jen o překlad mezi dvěma syntaxemi. Abstrakce může existovat **uvnitř téhož jazyka** jako sada konvencí, kontraktů a opakovaně použitelných mechanismů.

Moje hypotéza je, že při programování přirozeným jazykem se hranice mezi jazykem, frameworkem a konkrétní aplikací začne více rozmazávat. Agent může část pravidel převzít jako „framework“, část přímo interpretovat, část materializovat do kódu a část ukotvit. To však není dokázaný směr dějin ani tvrzení, že frameworky zmizí. Je to pracovní předpověď vycházející z toho, jak se v praxi prolínají instrukce, nástroje a generované artefakty.

Co se vlastně stalo s Promptbookem?

Autonomní agent dlouhodobě přidával funkce. Každý úkol vycházel ze stávajícího projektu. Protože jsme prakticky všechno zachovávali, nepozorovaně jsme ukotvovali skoro celý zdrojový kód. Ne proto, že každá komponenta byla cenná. Prostě jsme ji nechávali na místě.

Z vygenerovaného výsledku se stal balík historických rozhodnutí. Když jsem nechal vytvořit specifikaci a Promptbook přepsat, udělal jsem dvě věci: vytáhl důležitý záměr do vyšší abstrakce a

zrušil většinu nechtěných ukotvení.

Nevyměnil jsem princip autonomního vývoje. Nezavrhl jsem Agent-Project-Task Framework. Jen jsem přestal předstírat, že každý řádek dosavadní implementace stojí za zachování.

Agent, Project, Task: kde tyto vrstvy žijí

V Promptbooku používáme Agent-Project-Task Framework (APT), podrobněji popsany ve whitepaperu na <https://www.ptbk.io/cs/whitepaper>.

Agent nese roli, cíl, pravidla, znalosti a nástroje. V naší analogii částečně připomíná konfiguraci překladače, ale také samostatně rozhoduje o práci. Project je kontextová nádoba: obsahuje specifikace, soubory, historii a případné ukotvené artefakty. Task je konkrétní krok změny, který může zadat člověk nebo může vzniknout z cíle agenta či externí události.

Tyto tři části tvoří živý kontext. APT byl stejný před přepsáním Promptbooku i po něm. Změnil se obsah projektu a míra ukotvení, nikoli princip frameworku.

Dark factory a hranice autonomie

Naše dlouhodobá vize je něco jako dark factory: systém, v němž agenti rozvíjejí a provozují svěřenou agendu bez neustálého lidského zadávání. Projekt má cíl, agenti rozpoznávají práci, vytvářejí úkoly, ověřují výsledky a pokračují.

Současně si myslím, že absolutní autonomie je v praxi nedosažitelný ideál. Ne proto, že modely nikdy nebudou dost chytré, ale protože hodnota není jen technická vlastnost výsledku. Někdo musí rozhodnout, co zachovat, který kompromis přijmout, kterou vlastnost už nikdo nepotřebuje a kdy je lepší implementaci zahodit.

AI s těmito rozhodnutími může pomáhat. Ale odpovědnost za záměr a hodnoty projektu nikam magicky nezmizí.

Programátor budoucnosti nebude hlavně psát další řádky

Myslím si, že mnoho lidí, jejichž práce dnes spočívá hlavně v produkování dalších funkcí, bude muset změnit způsob práce. Neznamena to ale, že programování končí. Pro mě je programování ve specifikacích pořád programování. Jen na jiné úrovni abstrakce.

Klíčová dovednost nebude pouze umět vytvořit další funkci. Bude to umět poznat, co má skutečnou hodnotu, co má být zdrojem pravdy, co ukotvit a co bez lítosti zahodit. Jaký cíl má agent sledovat. Jaké hranice má respektovat. Kdy další generování pomáhá a kdy jen hromadí balast.

Možná takovým lidem budeme dál říkat programátoři. Možná jim někdo vymyslí jiné označení. Mně připadá, že jde stále o programování: jen jsme se posunuli o další úroveň abstrakce.

A právě proto se nebojím, že by se práce lidí jednoduše vypařila. Některé dnešní role nepochybně zaniknou nebo se promění. Ale schopnost rozhodovat o hodnotě, prioritách a správných hranicích abstrakce bude ještě důležitější.

Když jsem přepsal Promptbook, nebylo největším úspěchem to, že AI dokázala napsat nový kód. To už uměla dávno.

Nejdůležitější bylo, že jsem konečně dokázal říct, který starý kód už nemá cenu zachovávat.

Budoucnost programování možná nebude o tom, kolik kódu dokážeme vytvořit. Bude o tom, kolik ho dokážeme s klidným svědomím pustit.

O autorovi. Pavol Hejný je softwarový vývojář a zakladatel projektů v oblasti AI. Vývoji softwaru se věnuje přibližně 15 let a poslední čtyři roky se intenzivně zabývá využitím AI při programování. Pracuje na Promptbooku a dalších projektech na pomezí webových technologií, agentů a automatizace. Tento článek je osobní technologická úvaha: historické analogie jsou vysvětlením, anchoring a výhled do budoucna jsou hypotézy otevřené diskusi.