Od bitů k LLM:

Přirozený jazyk jako programovací jazyk

Feynmanovsky vhled do 80 let historie programování: od binárního kódu po LLM, od punch cards po GitHub Copilot. Proč počítače mluví dvojkově, jak každá generace jazyka posunula hranici abstrakce — a co to znamená, když programovacím jazykem je dnes čeština.

> *"There's a new kind of coding I call 'vibe coding', where you fully give in to the vibes, embrace exponentials, and forget that the code even exists."*

— Andrej Karpathy, 2. února 2025

9 kapitol	Historie jazyků · Binární základ · Metodologie · LLM · Budoucnost
Klíčový argument	"Programovací jazyk zítřka? Možná myšlenka. Ale gramotnost kódu: nezastarává."
Styl	Feynman × Tufte — každý princip s vizualizací, každý claim s evidencí
Verze	Březen 2026

Obsah

Vrstva	Éra	Příklad	Kdo píše?
Fyzická / Strojový kód	1940–1950	`10110100 11001010...`	Hrstka specialistů
Assembler	1950–1960	`MOV AX, BX`	~10 000 světově
Strukturované (C, Fortran)	1957–1980	`for(i=0;i<n;i++)`	Stovky tisíc
Vysokoúrovňové (Python)	1991–2022	`for x in seznam:`	~27 milionů
LLM / Přirozený jazyk	2022–dnes	"Udělej mi tabulku..."	**Kdokoliv**

0	Úvod — Proč mluvit s počítačem?	3
1	Fyzický základ: Proč počítače mluví dvojkově	4
2	Strojový kód a assembler — mluva křemíku	5
3	Strukturované jazyky — C, Fortran, COBOL	6
4	Vysokoúrovňové jazyky — Python, Java, JS	7
5	Metodologie, která přetrvá: TDD, Git, Stack Overflow	8
6	Doménové jazyky — SQL, HTML, Regex, DSL	9
7	LLM: Přirozený jazyk jako programovací jazyk	10
8	Konec profese, nebo zrození sportu?	11
9	Závěr — Co si odnést do budoucnosti	12

0 — Úvod: Proč mluvit s počítačem?

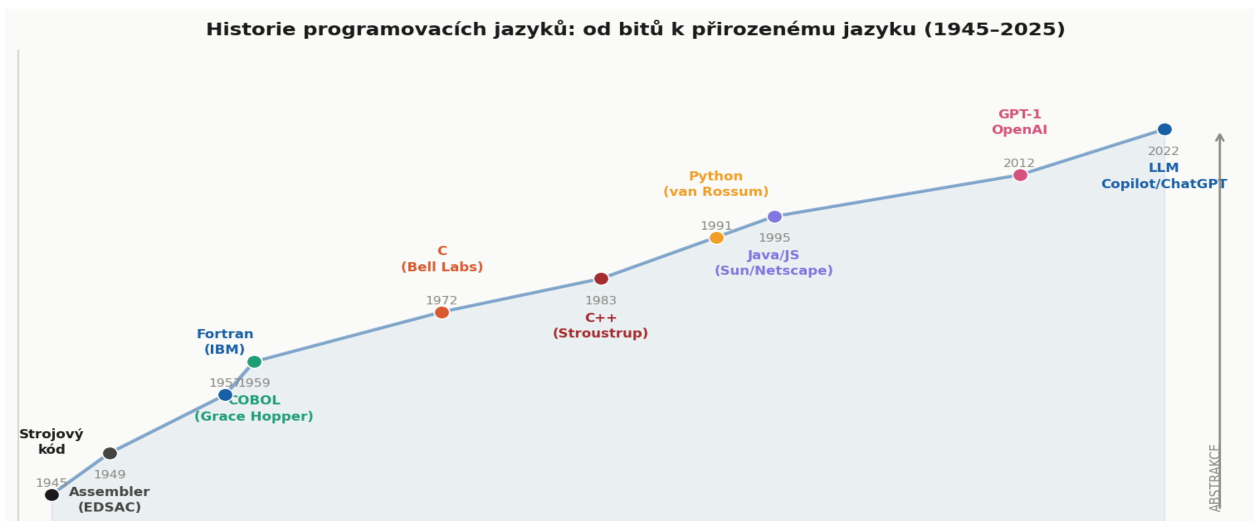
Klíčová otázka: Proč trvalo 80 let, než jsme se mohli s počítačem bavit česky — a co to změní?

V roce 1945 programátor nebyl ten, kdo *přemýšlí* — byl ten, kdo *překládá*. Každý krok algoritmu musel být přeložen do binárního kódu, do série nul a jedniček, které procesor doslova rozuměl jako elektrickým impulzům. Abstrakce neexistovala. Mezi myšlenkou a strojem stála zeď z čísel.

Tato publikace sleduje 80letou cestu, na které se tato zeď postupně bořila. Assembler přinesl slova místo čísel. Fortran přinesl vzorce. Python přinesl čitelnost. A ChatGPT, Claude, Copilot — přinesli češtinu. Každá generace jazyka posunula *kde* musí stát hranice porozumění: blíže k člověku, dál od křemíku.

Feynmanova otázka k úvodu

Pokud nedokážete vysvětlit, proč počítač nerozumí analogovému signálu, proč používá dvě cifry místo desíti — nerozumíte tomu, co se děje uvnitř každého příkazu v Pythonu. Začneme od fyziky.



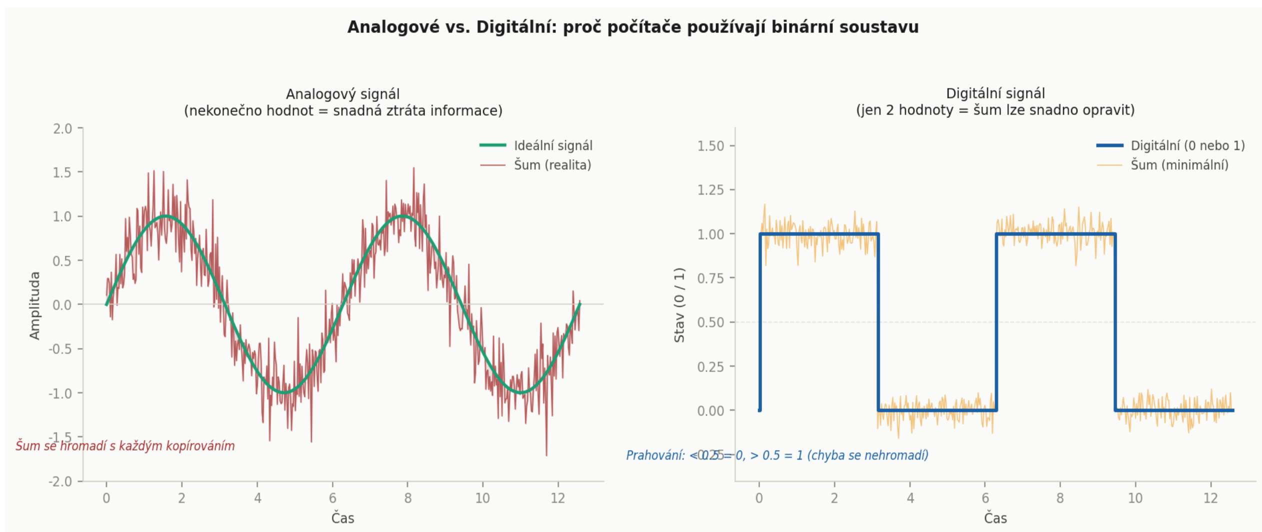
Obr. 0.1 — Přehled 80 let: od strojového kódu (1945) po LLM prompty (2022). Osa Y symbolizuje míru abstrakce — vzdálenost od hardwaru směrem k přirozenému jazyku.

1 — Fyzický základ: Proč počítače mluví dvojkově

Klíčový princip: Binární soustava není libovolná volba — je přímým důsledkem fyziky tranzistoru.

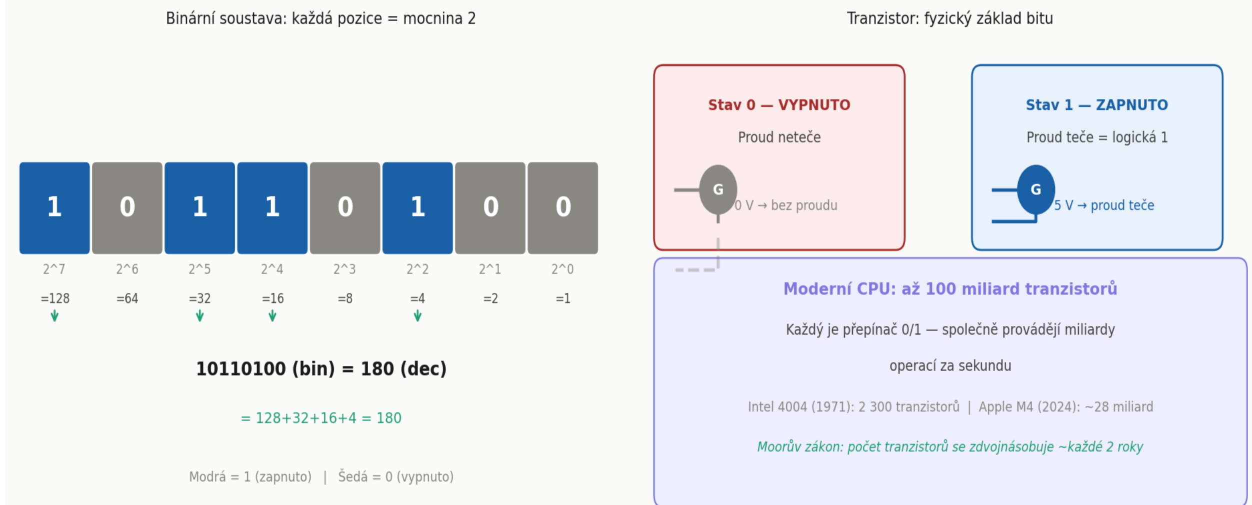
Intuitivně bychom možná čekali, že počítač pracuje s desítkovou soustavou — vždyť máme deset prstů. Nebo snad analogově, jako živé systémy, kde mozek operuje s kontinuálními potenciály, ne s bity. Ale fyzika materiálů rozhodla jinak: tranzistor je přepínač. Buď vede proud, nebo nevede. Žádný "trochu".

Tento fyzický fakt má elegantní důsledek: **šum se nekumuluje**. Když digitální signál projde tisíci tranzistory a trochu se "zkřiví", prahování (< 0,5 V = 0, > 0,5 V = 1) ho perfektně obnoví. Analogový signál by se s každou kopií degradoval.



Obr. 1.1 — Analogový vs. digitální signál: u analogového se šum hromadí s každým přenosem (vlevo). Digitální signál lze prahováním vždy obnovit — proto jsou počítačové operace 100% reprodukovatelné.

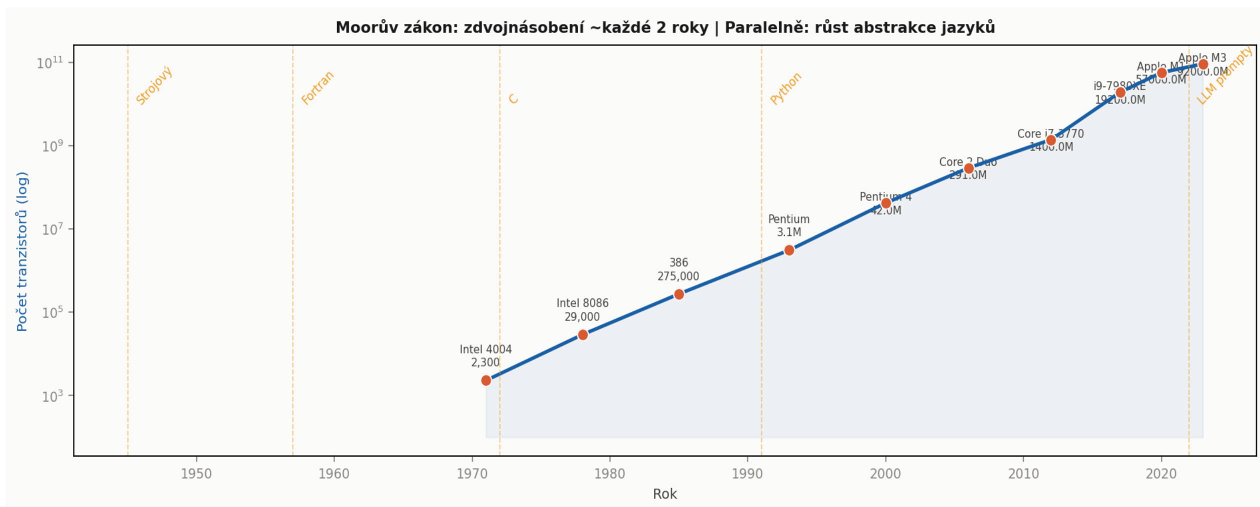
Základní jednotka informace je **bit** (binary digit) — hodnota 0 nebo 1. Osm bitů tvoří **bajt**, který dokáže zakódovat 256 různých hodnot ($2^8 = 256$). Písmeno "A" = bajt 65 = 01000001 v binárním zápisu. Barva pixelu = 3 bajty (červená, zelená, modrá = 16,7M barev).



Obr. 1.2 — Tranzistor jako fyzický základ bitu (nahoře) a váhový systém binárních pozic: každá pozice = mocnina 2. Moderní CPU: ~28–92 miliard tranzistorů.

Analogové a živé systémy: proč mozek nepočítá binárně?

Neurony pracují s kontinuálními potenciály — "analogový" výpočet. Biologické systémy tolerují chyby a nepřesnosti; vyvíjely se pro přežití, ne pro přesné výpočty. Počítač byl navržen pro reprodukovatelnost: 2+2 musí být vždy přesně 4. Proto tranzistor, proto binárně. AI (LLM) paradoxně aproximuje — vrací se k "analogovosti" přírody na digitálním substrátu.



Obr. 1.3 — Moorův zákon: počet tranzistorů se zdvojnásobuje přibližně každé 2 roky (1971–2023). Paralelně narůstala abstrakce programovacích jazyků.

Hexadecimální soustava: proč programátoři píší 0xFF

Binární soustava je fyzicky přirozená, ale pro člověka nepraktická — číslo 255 vypadá jako 11111111. Programátoři proto používají **šestnáctkovou (hexadecimální) soustavu**, kde každá cifra odpovídá přesně 4 bitům. Místo 11111111 napíšete jednoduše FF.

```
Binárni → Hex → Decimál 0000 0 0 0001 1 1 1001 9 9 1010 A 10 ← písmena A–F pro hodnoty 10–15 1111 F 15 1111 1111 FF 255 ← 1 bajt = přesně 2 hex cifry 0100 1111 4F 79 ← písmeno "O" v ASCII Proč hex a ne desítky? 16 = 2^4 → 1 hex cifra = přesně 4 bity Desítky (10) není mocninou 2 → konverze binárni↔dek je složitá
```

Kde hexadecimální čísla vidíte každý den

• Barvy v CSS: #FF5733 = červená (R=255, G=87, B=51 v decimálu) • Paměťové adresy: 0x7FFF5FBFF8B0 — adresa proměnné v RAM • MAC adresy síťových karet: 00:1A:2B:3C:4D:5E • SHA256 hash souboru: a3f1bc... (64 hex znaků = 256 bitů) • Chybové kódy Windows: STOP 0x0000007B (Blue Screen of Death)

Proč MB není 1 000 kB: binární vs. dekadické předpony

Jeden z nejběžnějších zdrojů zmatku v IT: výrobci pevných disků uvádějí 1 TB = 1 000 000 000 000 bajtů, ale operační systém zobrazí "931 GB". Kdo lže? *Nikdo* — jen používají různé definice.

```
DEKADICKÉ předpony (SI – mezinárodní systém, výrobci disků): 1 kB = 1 000 B (kilo = 10^3) 1 MB = 1 000 000 B (mega = 10^6) 1 GB = 1 000 000 000 B (giga = 10^9) 1 TB = 10^12 B BINÁRNÍ předpony (IEC 80000-13 – OS, RAM, flash paměti): 1 KiB = 1 024 B (kibibajt = 2^10 = 1 024) 1 MiB = 1 048 576 B (mebibajt = 2^20) 1 GiB = 1 073 741 824 B (gibibajt = 2^30) Příklad: disk "1 TB" = 10^12 B = 10^12 / 2^30 GiB ≈ 931 GiB → Windows zobrazí "931 GB" (myslí tím GiB), výrobce píše "1 TB"
```

Proč 1 024 a ne 1 000?

Počítačová paměť se adresuje v mocninách 2: 2^10 = 1 024 je nejbližší mocnina 2 k číslu 1 000. RAM musí mít velikost 2^n (128 MB, 256 MB, 512 MB, 1 GB...) — jinak by adresování bylo složité. Proto "kilobajt" v kontextu paměti přirozeně znamenal 1 024 B. Standard IEC zavedl v roce 1998 termíny KiB/MiB/GiB pro jednoznačnost — ale praxe je dodnes chaotická: Windows říká "GB" ale myslí GiB, macOS od verze 10.6 správně říká "GB" a myslí GB.

2 — Strojový kód a assembler: mluva křemíku

Klíčový princip: Strojový kód = instrukce přímo pro CPU. Assembler přidal první abstrakci: čitelná slova místo čísel.

První programátoři neprogramovali — *kódovali*. Každá instrukce měla svůj binární vzor; přičíst dvě čísla znamenalo zapsat specifickou sekvenci bitů na konkrétní paměťovou adresu. ENIAC (1945) se programoval fyzickým přepojováním kabelů. Bez chyby. Vždy.

```
; Assembler x86 – secti dve cisla (a + b) MOV AX, [a] ; nacti promennou a do registru AX MOV BX, [b] ; nacti promennou b do registru BX ADD AX, BX ; secti: AX = AX + BX MOV [result], AX ; uloz vysledek zpet do pameti ; V Pythonu: result = a + b (jeden radek!)
```

Assembler (1949, EDSAC) přinesl revoluci: místo binárního kódu 10110000 01100001 programátor napsal MOV AL, 61h. Překlad z mnemonik na bity zajistil **assembler** — první překladač v historii. Zásadní omezení: každý procesor měl jiný assembler. Program napsaný pro Intel nefungoval na Motorole.

Grace Hopper a první kompilátor (1952)

Rear Admiral Grace Hopper jako první argumentovala, že počítač může sám překládat "anglicky podobné" příkazy do strojového kódu. Výsledek: kompilátor A-0 a later COBOL — první jazyk blízký přirozenému obchodnímu jazyku. "Proč programátoři musí mluvit jazykem stroje? Stroj by se měl naučit jazyk programátora." — G. Hopper, 1952

3 — Strukturované jazyky: C, Fortran, COBOL

Klíčový přechod: Od instrukcí pro CPU k příkazům blízkým lidské logice. Abstrakce násobek 10 oproti assembleru.

Fortran (1957, IBM) byl první jazyk navržený pro vědce a inženýry. Jméno pochází z "Formula Translation" — matematické vzorce se poprvé daly zapsat téměř přesně tak, jak je vědec navyknul psát na papír. Výpočet trajektorie raketoplánu? Zapsatelný. Numerická integrace? Čitelná pro fyzika bez znalosti hardwaru.

```
! Fortran 77 – soucet prvku pole REAL A(100), SUMA SUMA = 0.0 DO 10 I = 1, 100 SUMA = SUMA + A(I) 10 CONTINUE WRITE(*,*) 'Soucet:', SUMA ! Python (1991): suma = sum(a) -- jeden radek!
```

C (1972, Bell Labs — Dennis Ritchie) šel jiným směrem: maximální výkon s minimální abstrakcí. C je "portable assembler" — kompiluje se téměř na raw strojový kód, ale je čitelný lidmi. Operační systém Unix byl přepsán z assembleru do C v roce 1973. Tím se Unix stal *přenositelným* — zlomový okamžik.

```
/* C: FizzBuzz (1972 stil) */ #include int main() { for (int i = 1; i <= 100; i++) { if (i % 15 == 0) printf("FizzBuzz\n"); else if (i % 3 == 0) printf("Fizz\n"); else if (i % 5 == 0) printf("Buzz\n"); else printf("%d\n", i); } return 0; }
```

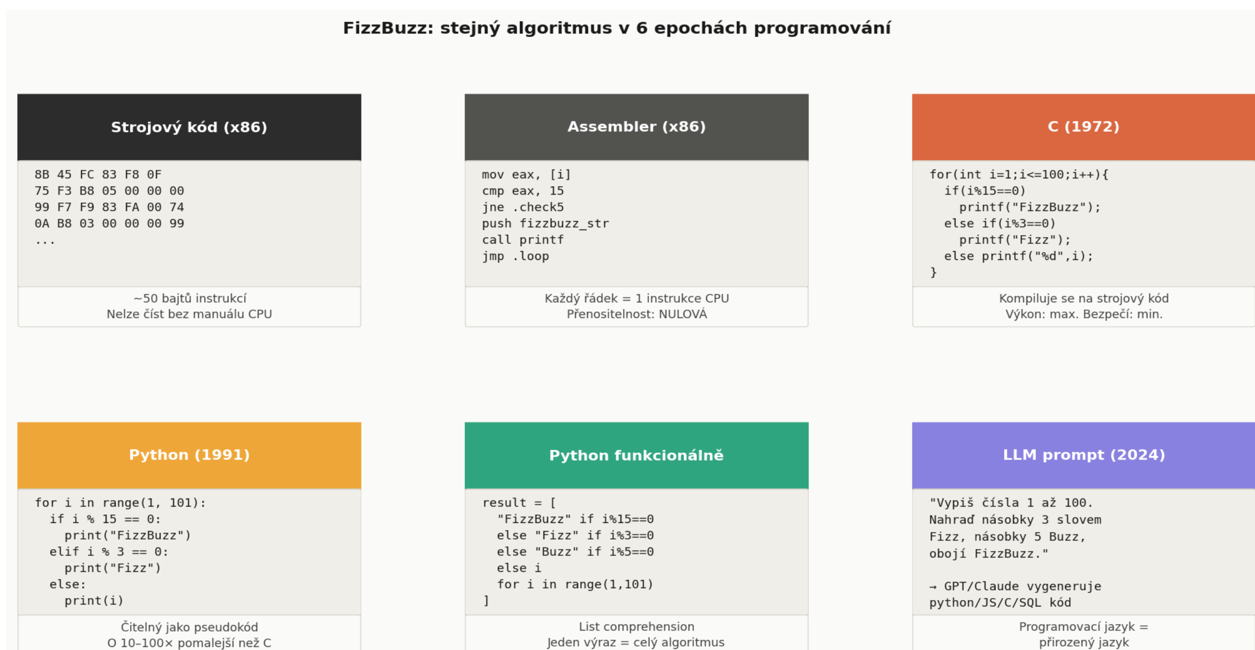
Paradox efektivity: C je pomalejší na psaní, ale rychlejší na výkon

C kompilátor generuje kód, který běží 10–100× rychleji než interpretovaný Python. Proto C a C++ stále pohánějí: jádra OS (Linux, Windows, macOS), databázové enginy (SQLite, PostgreSQL), herní enginy (Unreal, Unity) a samotné interpretery Pythonu a JavaScriptu. Abstrakce má svou cenu v CPU cyklech.

4 — Vysokoúrovňové jazyky: Python, Java, JavaScript

Demokratizace programování: Čitelnost jako primární hodnota. "Programs must be written for people to read, and only incidentally for machines to execute." — Abelson & Sussman, SICP (1984)

Python (Guido van Rossum, 1991) byl navržen s explicitním cílem: aby byl čitelný jako pseudokód. Odsazení je syntaxí, ne stylem. Jeden způsob jak věci dělat (PEP 20: "There should be one — and preferably only one — obvious way to do it"). Výsledek: dnes nejpoužívanější jazyk světa (TIOBE index 2024: #1), primární jazyk vědeckého výpočtu, AI, skriptování.



Obr. 4.1 — Stejný algoritmus (FizzBuzz) v 6 epochách: od strojového kódu přes assembler, C, Python, Python funkcionálně až po LLM prompt. Každá vrstva přidává čitelnost na úkor přímé kontroly nad hardwarem.

Java (Sun Microsystems, 1995) přinesla klíčové motto: "Write once, run anywhere." Bytecode kompilovaný do JVM (Java Virtual Machine) fungoval na jakémkoliv OS. OOP (objektově orientované programování) jako dominantní paradigma. Dnes pohání Android, enterprise systémy, Minecraft.

JavaScript (Brendan Eich, 1995 — za 10 dní) byl navržen jako skriptovací jazyk pro Netscape Navigator. Dnes běží na 98,7 % webů světa, na serverech (Node.js), v mobilních aplikacích (React Native), v desktopových aplikacích (Electron). Příběh o neúmyslném standardu.

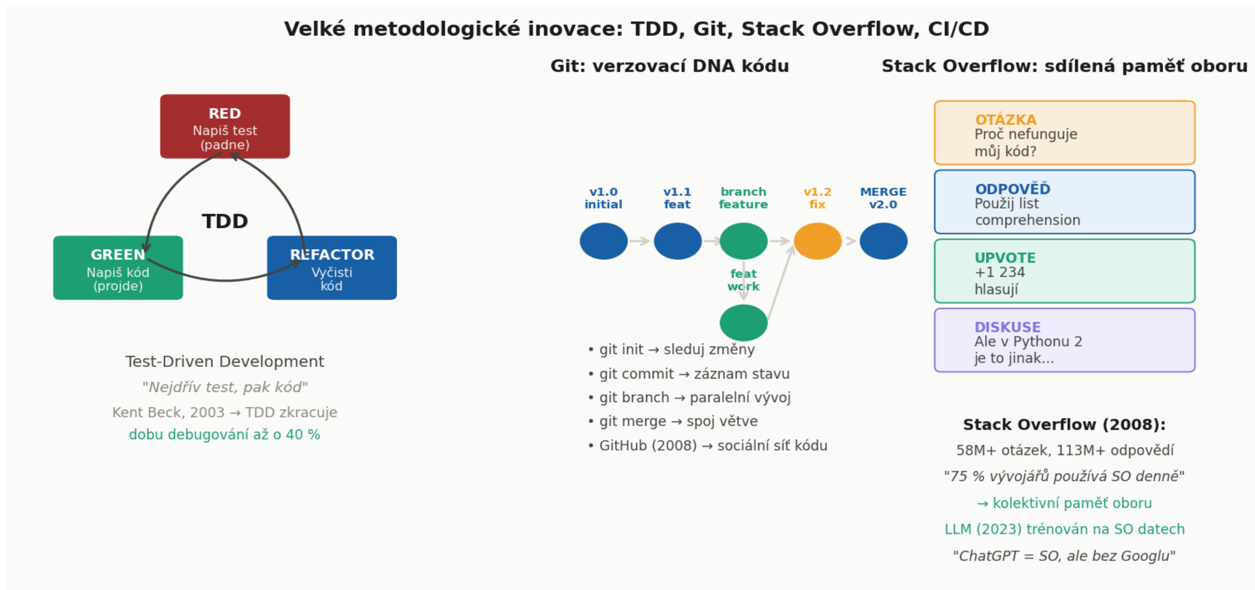
Abstraction leak: co se "skrývá" v Python příkazu

Jeden řádek Pythonu: seznam = sorted(data, key=lambda x: x["age"]) Za kulisami: CPython interpret → bytecode → JIT → CPU instrukce pro comparison sort (TimSort) → cache misses → branch prediction → stovky CPU cyklů. Abstrakce skrývá komplexitu — ale nekanceluje ji.

Obr. 4.2 — Abstrakční pyramida: od fyzické vrstvy tranzistorů po LLM. Každá vrstva skrývá komplexitu té pod ní — ale nezruší ji. Porozumění spodním vrstvám odlišuje seniorního inženýra od juniora.

5 — Metodologie, která přetrvá: TDD, Git, Stack Overflow

Klíčový insight: Jazyky se mění — přístupy k řešení problémů přetrvávají. TDD, verzování a sdílení znalostí budou relevantní i tehdy, kdy kód píše AI.



Obr. 5.1 — Tři velké metodologické inovace: TDD (test-first vývoj), Git (verzování a větvení) a Stack Overflow (kolektivní znalostní báze). Každá z nich zásadně změnila, jak lidé spolupracují na kódu.

Test-Driven Development (TDD) invertoval tradiční workflow. Místo "napište kód, pak otestujte" přišel Kent Beck (1999) s: "napište test, který selže — pak napište minimální kód, aby prošel — pak refaktorujte." Výzkum Carnegie Mellon (2008): TDD zkracuje dobu hledání chyb o 40–80 % při nákladu ~15 % pomalejšího psaní.

```
# TDD v praxi — Python/pytest # KROK 1: Nejdrive test (CERVENY — pada) def
test_fizzbuzz_15(): assert fizzbuzz(15) == "FizzBuzz" # NameError: fizzbuzz neexistuje #
KROK 2: Minimalni implementace (ZELENY — projde) def fizzbuzz(n): if n % 15 == 0: return
"FizzBuzz" if n % 3 == 0: return "Fizz" if n % 5 == 0: return "Buzz" return str(n) # KROK
3: Refactor — kod je uz otestovany, lze bezpecne menit
```

Git (Linus Torvalds, 2005) vznikl za týden — Torvalds potřeboval verzovací systém pro Linux kernel po rozkolu s BitKeeper. Klíčová inovace: *distributed* verzování. Každý vývojář má celou historii lokálně. Větvení a merging jsou laciné operace. **GitHub** (2008) přidal sociální vrstvu: fork, pull request, issues — open source spolupráce se stala masovou.

Stack Overflow (2008, Joel Spolsky & Jeff Atwood) vyřešil problém sdílení tacitní znalosti. Než SO existoval, programátoři bojovali s mailinglists, IRC, nedokumentovanými fóry. SO přinesl strukturu: otázka → nejlepší odpověď → hlasování komunity. Dnes: 58M+ otázek, 113M+ odpovědí. A ironie: LLM modely jako GPT a Claude byly trénovány na SO datech — SO je "zakonzervovaná" kolektivní paměť programátorské komunity.

Git a TDD v LLM éře: proč budou důležitější, ne méně

Když AI generuje 46 % kódu (GitHub, 2023), potřeba verzování, testů a review se ZVYŠUJE. AI generuje plausibilní kód — ne nutně správný. TDD zajistí, že AI-generovaný kód prochází specifikací. Git zajistí, že každou změnu lze vrátit. "Trust, but verify" — pro AI kód platí dvojnásob.

6 — Doménové jazyky: SQL, HTML, CSS, Regex

Klíčový přechod: Specializované jazyky pro specializované domény. Extrémní abstrakce za cenu omezené obecnosti.

Paralelně s obecnými jazyky vznikla celá rodina **DSL (Domain-Specific Languages)** — jazyků navržených pro jeden konkrétní problém. Tyto jazyky jsou vědomou volbou: obětují obecnost za *expresivitu* v dané doméně.

```
-- SQL (1974, IBM): Databazove dotazy SELECT jmeno, vek FROM uzivatele WHERE vek > 18 AND
mesto = 'Praha' ORDER BY vek DESC LIMIT 10; Nadpis Odstavec textu s odkazem. # Regex
(1951, Kleene): Vzory v textu ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ #
Overuje, zda retezec je validni e-mail
```

SQL (Edgar Codd & Donald Chamberlin, 1974) je pravděpodobně nejdéle aktivně používaný programovací jazyk vůbec. Navrháž deklarativní přístup: říkáš *co* chceš, ne *jak* to získat. Databázový engine sám rozhoduje o exekučním plánu. PostgreSQL, SQLite, MySQL, Oracle — vše staví na Coddových principech z roku 1970.

Deklarativní vs. imperativní: zásadní dělení paradigmat

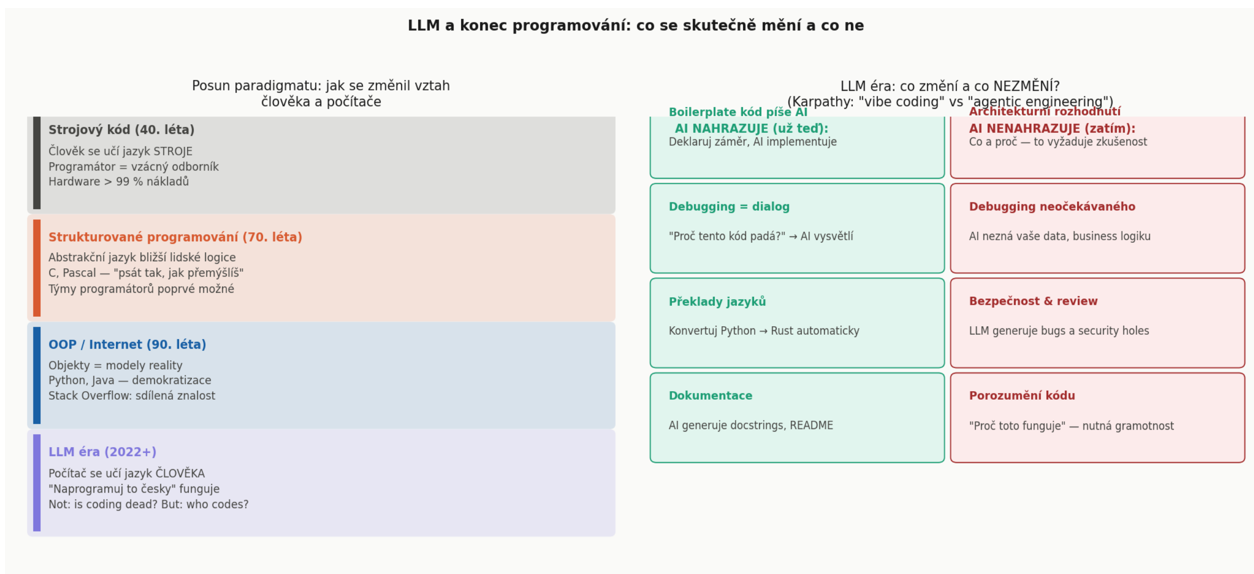
Imperativní (C, Python): říkáš "jak" — krok za krokem, sekvence instrukcí. Deklarativní (SQL, HTML, CSS): říkáš "co" — engine rozhodne jak. Funkcionální (Haskell, Erlang): matematické transformace bez vedlejšího stavu. LLM prompty jsou extrémně deklarativní: "Udělej X" — bez specifikace postupu.

7 — LLM: Přirozený jazyk jako programovací jazyk

Historický zlom: Poprvé v historii počítačů se stroj přizpůsobil jazyku člověka — ne naopak.

Každý krok v historii programování byl formou kompromisu: programátor se naučil více strojové logiky, výměnou za to, že stroj dokázal více. Fortran = naučte se napsát vzorec přesně. Python = naučte se čitelnou syntaxi. SQL = naučte se deklarativní logiku. Vždy šlo o *setkání na půl cesty*.

LLM modely (GPT-3, 2020; ChatGPT, 2022; Claude, Gemini, 2023) posunuly setkání zcela na stranu člověka. Programovacím jazykem je dnes čeština, angličtina, nebo jakýkoliv přirozený jazyk. Karpathy nazval tento přístup "vibe coding" — "plně se odevzdej vibes, přijmi exponenciály, zapomeň, že kód existuje."



Obr. 7.1 — Posun paradigmatu (vlevo): čtyři epochy vztahu člověka a stroje. Co LLM mění a co nemění (vpravo): AI nahrazuje boilerplate a překlady, ale architektura, bezpečnost a porozumění zůstávají lidskou zodpovědností.

GitHub Copilot (2022) reportoval, že **46 % nového kódu** na GitHubu generuje AI. V roce 2025 Anthropic reportoval, že 25 % Y Combinator W25 startupů píše 95 % kódu pomocí AI. Stack Overflow v roce 2023 zaznamenal první pokles návštěvnosti za 15 let — přímý důsledek přechodu na ChatGPT jako první zdroj odpovědí.

```
# Stejny algoritmus (FizzBuzz) jako LLM prompt: Uzivatel: "Vypis cisla 1 az 100. Nahrady
nasobky 3 Fizz, nasobky 5 Buzz, nasobky oboji FizzBuzz." LLM → vygeneruje validni
Python, JavaScript, SQL nebo C podle kontextu — bez znalosti syntaxe uzivatelem #
Priklady "LLM-first" workflow (2025): "Refaktoruj tento kod tak, aby byl testovatelny"
"Proc tato funkce vraci None misto listu?" "Preloz tento Python skript do Rustu"
"Vygeneruj unit testy pro vsechny edge cases"
```

Hallucination: největší omezení LLM kódu

LLM modely generují "plausibilní" kód — ale ne vždy správný. Typické problémy: zastaralé API (model byl trénován v 2023, knihovna změnila API), neexistující metody ("hallucinated functions"), subtilní logické chyby v edge cases. Proto TDD + Code review zůstávají kritické — ne jako check na člověka, ale jako check na AI.

8 — Konec profese, nebo zrození sportu?

Klíčová analogie: Šachový počítač Stockfish hraje lépe než Magnus Carlsen — a přesto šachy nikdy nebyly populárnější. Stejné principy platí pro programování.

Když Deep Blue v roce 1997 porazil Kasparova, média psala o "konci šachů". O 27 let později: Chess.com má přes 150 milionů registrovaných uživatelů, Magnus Carlsen vydělává desítky milionů dolarů a šachové streamy na Twitchi mají stovky tisíc diváků. AI nefunguje jako náhražka šachů — funguje jako jejich *akcelerátor*.

Obr. 8.1 — Programování jako šach a hudba: v každém oboru AI překonala lidský výkon, a přesto se lidská účast zvýšila. Klíč: "chci hrát, ne jen sledovat" — lidská potřeba tvořit přetrvá automatizaci.

Paralelní trendy v hudbě: AIVA, Suno.ai a MuseNet generují hudbu nerozlišitelnou od průměrné lidské tvorby. Účast na jam sessions, prodej hudebních nástrojů a zájem o výuku instrumentů přesto od 2020 roste. Lidé hrají pro *radost z tvoření*, ne pro soutěž s algoritmem.

Programování dnes: **Advent of Code** (každoroční programátorská soutěž) má v roce 2024 přes 300 000 účastníků. **Codeforces** a **LeetCode** reportují rekordní registrace. **Modding** her (Minecraft, Stardew Valley) přivedl k programování miliony lidí, kteří "neprogramují" — ale programují. Open source projekty na GitHubu: 420 milionů repozitářů (2024).

"Karpathyho dělení" jako klíč k budoucnosti

"Vibe coding" (throwaway) vs "agentic engineering" (produkční, s dohledem) — Karpathy, 2026.
Vibe coding = sport, kreativita, explorace. Agentic engineering = profese s AI jako nástrojem.
Obojí vyžaduje jiné dovednosti — ale obojí vyžaduje porozumění, co stroj vlastně dělá.

Srovnávací tabulka: 80 let programovacích jazyků

Každá epocha přinesla novou abstrakci a nový kompromis mezi výkonem a čitelností. Tato tabulka shrnuje klíčové charakteristiky každé generace od strojového kódu po LLM.

Jazyk / Éra	Rok	Paradigma	Ukázka kódu	Výkon	Čitelnost	Kde dnes?
Strojový kód	1945	Imperativní (instrukce CPU)	`8B 45 FC 83 F8 0F`	★★★★★	★☆☆☆☆	Embedded, firmware
Assembler	1949	Imperativní (mnemoniky)	`MOV AX, BX ADD AX, 1`	★★★★☆	★★☆☆☆	OS jádra, drivery
Fortran	1957	Procedurální (vědecký)	`DO 10 I=1,N SUMA=SUMA+A(I)`	★★★★☆	★★★☆☆	HPC, simulace
COBOL	1959	Procedurální (business)	`MOVE A TO B ADD X TO Y`	★★★☆☆	★★★☆☆	Bankovnictví, legacy
C	1972	Imperativní (systémový)	`for(i=0;i<n;i++) sum+=a[i];`	★★★★★	★★★☆☆	OS, databáze, jádra
C++	1983	OOP + proced.	`class Foo { int x; };`	★★★★☆	★★☆☆☆	Hry, finance, HPC
SQL	1974	Deklarativní (dotazy)	`SELECT * FROM users WHERE id=1`	★★★☆☆	★★★★☆	Každá databáze
HTML / CSS	1991	Deklarativní (struktura/styl)	`Nadpis .cls{color:red}`	N/A	★★★★☆	Každý web
Python	1991	Multi-paradigma	`for x in lst: print(x)`	★★★☆☆	★★★★★	AI/ML, věda, web
Java	1995	OOP (JVM)	`public class Main {}`	★★★★☆	★★★☆☆	Android, enterprise
JavaScript	1995	Multi-paradigma	`const f = x => x*2`	★★★☆☆	★★★☆☆	98 % webů světa
Rust	2010	Systémový (bezpečný)	`fn main() { let x=5; }`	★★★★★	★★☆☆☆	WebAssembly, OS
Go	2009	Procedurální (konk.)	`func f() { go worker() }`	★★★★☆	★★★★☆	Cloud, microservices
LLM prompt	2022	Deklarativní (přir. jazyk)	`"Udělej mi tabulku..."`	Variabilní	★★★★★	Vše — roste rychle

Metodologie: co přetrvá i v době AI

Přístup	Vznik	Klíčová myšlenka	Měřitelný přínos	Relevance v LLM éře
TDD	1999 Kent Beck	Test napíšeš PŘED kódem — nepiš víc než je nutné	−40 % debug +15 % čas psaní	VYŠŠÍ — AI generuje kód, testy ověří správnost
Git	2005 Torvalds	Každá změna = snapshot, větvení zdarma	420M+ repozitářů, standard průmyslu	VYŠŠÍ — AI mění hodně kódu najednou, rollback nutný
Code review	70. léta IBM	Peer kontrola — 4 oči vidí víc než 2	−60 % bugů v produkci (Google 2014)	KRITICKÁ — AI generuje plausibilní ale chybný kód

Přístup	Vznik	Klíčová myšlenka	Měřitelný přínos	Relevance v LLM éře
Stack Overflow	2008 Spolsky & Atwood	Kolektivní Q&A; — nejlepší odpověď nahoru	58M+ otázek, zdroj LLM tréninku	TRANSFORMUJE SE — ChatGPT bere část trafficu
CI/CD	2000 XP komunita	Každý commit = automatický build + testy + deploy	Deploy týdně → několikrát denně	ROSTE — AI PR musí projít stejným pipeline
Dokumentace	Vždy existovala	Komentáře, README, docstrings = živá paměť projektu	Snižuje bus factor, umožňuje onboarding	AI generuje doc — ale správnost = lidský úkol

9 — Závěr: Co si odnést do budoucnosti

Prošli jsme 80 lety: od fyzické nutnosti binárního kódu, přes první abstrktorní vrstvy assembleru, přes demokratizaci Pythonu, až po LLM éru, kdy programovacím jazykem je čeština. Co z toho plyne prakticky?

1. Abstrakce skrývá, ale nekanceluje komplexitu

Python `sum(data)` skrývá sort algorithm, cache misses a CPU cycles. Znát spodní vrstvy odlišuje seniora od juniora — a v LLM éře: znát, co AI může a nemůže, odlišuje efektivního uživatele od naivního.

2. Metodologie přetrvá jazyky

TDD, Git, code review existovaly před LLM a budou existovat po něm. AI generovaný kód potřebuje testy víc, ne méně. "Trust, but verify" — to platí pro kolegu i pro GPT-5.

3. Programovací gramotnost ≠ znalost syntaxe

Číst kód, chápat algoritmy, rozumět datovým strukturám — to jsou přenositelné dovednosti. Syntaxe Pythonu je googleable; schopnost navrhnout správný algoritmus není.

4. LLM je nástroj, ne náhrada za myšlení

GitHub Copilot generuje kód — ale ne architekturu. ChatGPT odpoví na otázku — ale nerozumí vašemu businessu. Efektivní využití LLM vyžaduje schopnost formulovat správné otázky a zhodnotit odpovědi.

5. Programování jako tvorba má hodnotu mimo pracovní trh

Šachy přežily Stockfish. Hudba přežila Suno.ai. Programování přežije GitHub Copilot — jako sport, umění a intelektuální radost. "Krásný algoritmus" je estetický zážitek, ne jen funkční výsledek.

> *"The real danger is not that computers will begin to think like men, but that men will begin to think like computers."*

— Sydney J. Harris, 1965 — stále aktuální v éře LLM

Kde se dozvědět víc

Knihy

- Abelson & Sussman — Structure and Interpretation of Computer Programs (SICP, 1984)
- Kernighan & Ritchie — The C Programming Language (1978)
- Kent Beck — Test-Driven Development: By Example (2002)
- Scott Patterson — The Quants; Joel Spolsky — Joel on Software (eseje online)

Online kurzy a zdroje

- cs.harvard.edu/CS50 — nejlepší úvod do CS zdarma (David Malan)
- docs.python.org/tutorial — oficiální Python tutorial
- learngitbranching.js.org — interaktivní Git vizualizace
- roadmap.sh — vizuální roadmapy pro backend, frontend, DevOps

O LLM a budoucnosti

- Andrej Karpathy — youtube.com/karpathy (Neural Networks: Zero to Hero)
- Simon Willison — simonwillison.net (nejlepší LLM novinky)
- arxiv.org — vědecké papers (hledej "LLM coding", "AI software engineering")

Od bitů k LLM — Březen 2026 · Vytvořeno s Claude Sonnet 4.6 · Styl: Feynman × Tufte