How Much Does a Programming Language Cost a Coding Agent?

Tokens, Time and Correctness with Claude Opus 5.5
across 17 Languages, Three Front-End Frameworks
and Four Specialised Tracks

Arthur De Freitas Abeilice
Universidade Federal de Minas Gerais (UFMG)
abeilice@ufmg.br https://github.com/afa7789/TKNbenchmark

November 2026

Abstract

We measure how much a coding agent spends to solve the same tasks in 17 programming languages and in React, Vue and Svelte. Every case (language × task × repetition) was run by its own isolated Claude Opus 5.5 agent with an identical prompt, and checked live by hidden tests, a clean build and, for front ends, a real browser. Across 282 agent runs, 3 failed verification and 3 of them were fixed by a repair agent that received the failure report. The language barely changed correctness, but it changed the cost: from the cheapest (Python, US$ 0.69 for four tasks) to the most expensive (Assembly) the gap was 4.8×. Python, TypeScript and JavaScript were the three cheapest. Final code size tracks cost closely (Spearman $\rho = 0.85$) but not completely: Zig cost 2.9× the cheapest, more than C, whose code is larger, because the agent had to read the standard library's source to learn a recently changed API. TypeScript's code was 1.26× JavaScript's in `cl100k` tokens, close to the 1.31× reported for snippets, yet the agent spent the same on both. Rust's build artifacts live per project, so parallel agent worktrees multiply them. For front ends we reproduce the published ordering (Svelte < Vue < React) with a different tokenizer. Four specialised tracks (scientific computing, theorem proving, smart contracts and zero-knowledge circuits; 10 more languages) passed every hidden check, and within each track the languages cost about the same. We rank the languages twice, by cost to write and by the speed of the final code (points per task, 100 for the fastest): the two rankings run against each other: the costlier a language was to write, the faster its code tended to be (Spearman $\rho = 0.63$ between cost and speed score), with Assembly first in speed and last in cost.

1 Motivation and hypotheses

Pereira et al. [1] asked how energy, time and memory relate when the same programs are written in 27 languages. Coding agents add a fourth resource to that list: the tokens spent to produce, test and fix the program. The series *The Tokenomics of Web Development* [11, 12] measured, with `tiktoken`, the token size of hand-written idiomatic snippets: Angular costs 38% more than Svelte, TypeScript 31% more than JavaScript. An agent, however, does more than emit code: it reads the specification, compiles, fails, reads error messages, looks things up and tries again. We wanted to know whether the choice of language changes that whole bill. A conversation between two developers, one of them running Rust + React in a server for chat bots, supplied the hypotheses:

H1 (Zig) Zig spends fewer tokens than Rust, compiles faster and is simpler for an LLM to get right on the first try.

H2 (Rust disk) Rust produces very large `target/` directories, which hurts when agents work in parallel worktrees.

H3 (conciseness) More concise languages cost the agent fewer tokens, as the series showed for static snippets.

H4 (front end) Svelte < Vue < React in tokens [11].

H5 (Rust + React) Rust + React is a good stack for people who program with agents.

2 Related work

Comparing programming languages. Empirical comparisons of languages solving the same problems go back at least to Prechelt [3], who compared seven languages on one task for program length, development effort, run time and memory. Nanz and Furia [4] compared eight languages on Rosetta Code for conciseness, size, speed and failures. Closest in spirit to this work, Pereira et al. [1, 2] ran the Computer Language Benchmarks Game in 27 languages and related energy, time and memory; we keep their design (the same problems across many languages, one resource measured at a time) and measure the tokens an agent spends instead. Ray et al. [5] and the reproduction by Berger et al. [6] show how hard it is to attribute outcomes to the language alone, a caveat that applies here too.

Language models and code across languages. HumanEval [7] measured the functional correctness of generated Python. MultiPL-E [8] translated it into 18 more languages and found that model performance depends on how common a language is and on its features, which matches our finding that the model's familiarity with a language version drives the cost.

Coding agents. SWE-bench [9] evaluates models on real GitHub issues from Python repositories; SWE-agent [10] showed that the agent–computer interface changes the outcome; DeepSWE [13] reports cost and steps alongside pass@1. These benchmarks hold the language fixed and vary the model. We hold the model and harness fixed and vary the language.

3 Method

Agents. Inspired by DeepSWE [13], each case runs its own headless Claude Code (`claude -p`): model `claude-opus-5-5`, effort `high`, tools `Bash`, `Read`, `Edit` and `Write`; no web, MCP, plugins or `CLAUDE.md`; an empty directory outside the repository. The prompt is the same for every language except for the language name and toolchain version (Assembly, the JVM languages, JavaScript, TypeScript and Python get one extra line: "only `.s`, libc calls allowed", or "`./app` may be a launcher script that runs `java`/`node`/`python3`"). The agent sees the specification, a few examples and the `check.mjs` script that runs them.

Tasks. Four back-end programs: `wordfreq` (bytes, hash maps, sorting, inputs up to 50 MB), `lru` (an LRU cache over one million commands), `calc` (a recursive-descent parser with `int64` overflow rules) and `kvhttp` (a concurrent HTTP key–value server, where third-party packages are allowed). The first three are standard-library only. For front ends, `ui` (the series' five components plus a client for `kvhttp`) and `shop` (a mini app with parent/child components, a global store of the agent's choice, derived state, a form, an effect with cleanup, context and slots). Three repetitions per case.

Specialised tracks. Languages that cannot run the main tasks get two tasks from their own domain, a verifier validated against reference solutions in every language of the track, and the same rounds. *Scientific computing* (GNU Octave, Julia, R): least-squares regression and dense linear algebra (log-determinant, solve, eigenvalues), compared to a relative tolerance of 10^{-6}. *Theorem proving* (Lean 4, Rocq, Dafny): four list lemmas over our own `append`/`rev`, and the correctness of insertion sort (sorted and count-preserving); the file must check with the language's own checker,

with no `sorry`/`Admitted`/`assume`/axioms and every definition and statement unchanged. *Smart contracts* (Solidity, Vyper): a token with allowances and a linear-vesting contract; the build emits ABI and bytecode, which the harness deploys on a fresh `anvil` chain and exercises with exact timestamps, expected reverts and exact events. *Zero-knowledge circuits* (Circom, Noir): a private range check and a 4×4 Sudoku; valid inputs must be provable and invalid ones (out of range, wrong clue, repeated row, column or box, huge field elements) must not.

Rounds. (1–2) Write. (3) Verify once, live: for back ends, a cold build of a clean copy with isolated caches and a hidden suite larger than the examples; for front ends, a Vite build and checks in Chromium with mocked network, testing only what the specification states literally plus the absence of runtime errors. (4) Repair: each failure gets a fresh agent with the failure report, up to three attempts. Each verifier was first validated against reference implementations.

Metrics. Tokens and cost come from Claude Code's final event (Opus 5.5 list price: US$ 4 and 20 per million input and output tokens, cache included). Final code was counted with Opus 5.5's own tokenizer, as the input-token difference between a prompt with the code and an empty one, and also with `cl100k_base` to compare with the series. We also count compiler errors and reads of the toolchain's own sources or docs, binary performance (median of five runs) and server throughput with 64 concurrent clients. Each performance benchmark gives a language 100× the fastest time divided by its own (for throughput, its own divided by the best), so 100 is the fastest; the speed score is the geometric mean over the four, and the speed index is 100/score, how many times slower than the fastest. Disk is split into per-project artifacts and shared global caches, so that the cost of N worktrees is $N \times \text{project} + \text{cache}$.

4 Results

We rank the back ends twice. By cost to write, Table 1 and Figure 1; costs are the sum of the per-task medians over the four tasks. By speed of the final code, Table 4, built from the measurements in Tables 2 and 3. Of the 282 runs, 3 failed round 3, all of them Odin `calc` on the same hidden random case; the repair round fixed each on the first attempt.

Table 1: Per language, summing the medians of the 4 tasks. "Solved": passed the hidden suite in round 3. "1st try": the first run of the examples already passed. "Code": Opus 5.5 tokens of the final code. "Stdlib": reads of the toolchain's own sources/docs.

Language	Solved	1st try	US$	Output	Turns	Min	Code	Errors	Stdlib
Python	12/12	92%	0.69	12 719	17	2.8	6 632	0.0	0
TypeScript	12/12	100%	0.72	14 127	16	2.7	8 437	0.0	0
JavaScript	12/12	100%	0.73	14 259	17	2.9	6 522	0.0	0
Kotlin	12/12	100%	0.75	14 080	20	5.6	6 795	0.0	0
Rust	12/12	100%	0.83	17 277	18	5.1	11 721	0.0	2
Nim	12/12	75%	0.85	15 954	23	5.6	7 378	0.0	2
Go	12/12	100%	0.85	18 221	17	3.1	12 141	0.0	0
Java	12/12	83%	0.88	18 242	22	5.6	10 378	0.0	0
OCaml	12/12	92%	0.91	19 637	25	3.9	10 227	0.0	1
C++	12/12	100%	0.97	21 789	20	5.9	13 469	0.0	0
Haskell	12/12	75%	1.10	23 785	33	7.2	9 758	1.0	2
Odin	9/12	50%	1.21	24 656	37	6.0	11 040	2.0	0
C#	12/12	8%	1.28	28 325	36	7.7	12 075	2.0	1
C	12/12	100%	1.31	33 266	25	6.1	22 502	0.0	0
Swift	12/12	8%	1.58	36 578	39	11.3	15 070	6.0	4
Zig	12/12	100%	1.96	39 457	48	9.7	16 568	3.0	21
Assembly	12/12	58%	3.28	99 286	37	15.8	36 769	1.0	0

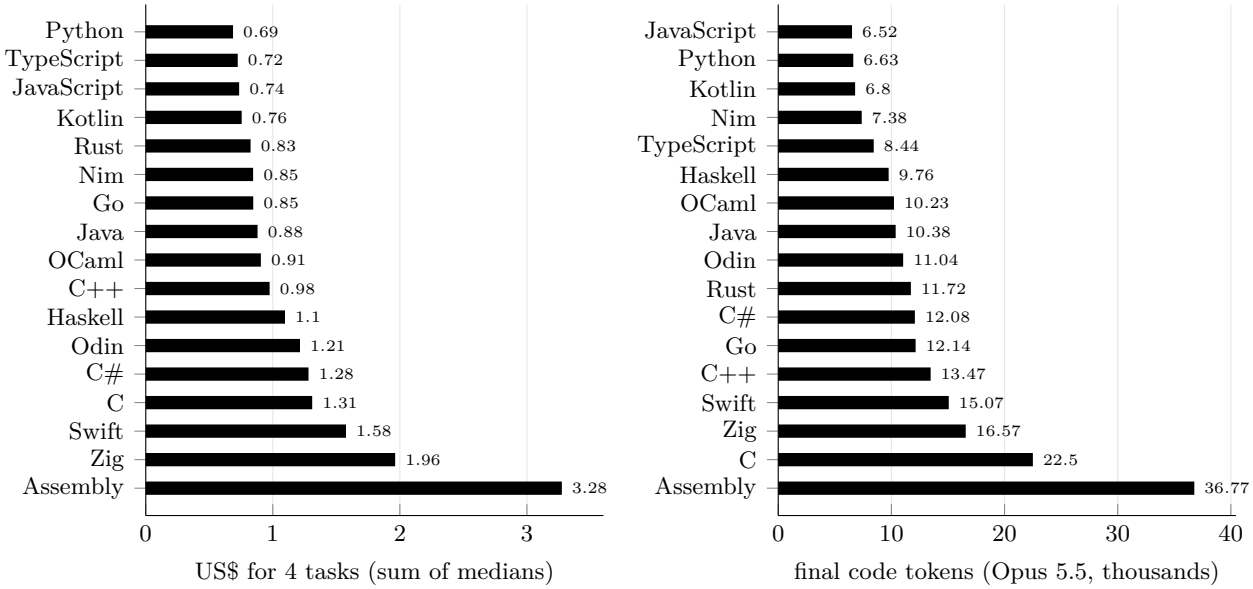


Figure 1: Agent cost (left) and final code size (right), per language.

5 Discussion

H1 (Zig): rejected in this setting. Zig cost US$ 1.96 against US$ 0.83 for Rust. It did not fail more often (100% passed on the first try, as Rust did); it spent the money learning. The agent made 21 reads of the Zig 0.16 standard library sources, whose I/O API changed recently, against 2 for Rust. Cold builds were not faster either: 5.4 s against 1.8 s on the standard-library tasks, because Zig compiles its standard library from source into an empty cache. A simple language is not enough; the model must know the version in use.

H2 (Rust disk): confirmed once dependencies and worktrees enter. Without dependencies Rust is light: 1.3 MB after the build, against 49 MB for Zig. For the HTTP server Rust and Haskell were the only languages whose agents pulled in packages (`tokio` and `hyper`/`axum`; `warp`), and Rust's release build reached 107 MB and 11.4 s. What matters is where those bytes live. `target/` is per project (76 MB for the release build alone), so every worktree repeats it, while Go keeps almost everything in its global cache (5.6 MB per project). With four worktrees Rust reaches 334 MB and Go 132 MB. After a full development loop (debug build, release build and test build) Rust's `target/` reached 206 MB per worktree, or 857 MB for four, for a server of about 200 lines. With hundreds of crates and incremental builds this becomes the gigabytes that people running parallel agents report. Rust is not alone, though: per project, the release builds of C# (223 MB in `bin/` and `obj/`) and Haskell (103 MB in `dist-newstyle/`, plus a shared cabal store; 1467 MB in total) were larger than Rust's, giving 937 MB and 1775 MB for four worktrees.

H3 (conciseness): confirmed for code, with caveats for cost. Across languages, final code size and agent cost are strongly associated: Spearman $\rho = 0.85$, Pearson $r = 0.92$, or $r = 0.71$ without Assembly, which is an outlier on both axes. The four cheapest languages (Python, TypeScript, JavaScript and Kotlin) include the three with the least code (JavaScript, Python and Kotlin). The more verbose Java cost more than Kotlin. C cost more than C++ because, lacking hash maps and strings in its standard library, it has to write more; Assembly was the extreme (4.8×). The series' headline comparison replicates for code but not for cost: TypeScript's final code was 1.26× JavaScript's in `cl100k` tokens (the series reports 1.31× for snippets) and 1.29× in Opus tokens, but the agent's cost ratio was 0.98×. Type annotations add tokens to the artefact, not to the process that writes it. The exceptions tell the other half of the story. Zig cost more than C, whose code is larger, because the agent paid for API uncertainty (21 standard-library reads);

Table 2: `kvhttp`: dependencies chosen by the agent, disk after a cold build (project + caches it created), build time and throughput.

Language	Dependencies (most common)	MB	Build s	req/s
Haskell	bytestring, containers, http-types, wai, warp	1467	145.6	23 610
C#	stdlib	277	9.5	22 434
Go	stdlib	115	3.1	24 521
Rust	bytes, http-body-util, hyper, hyper-util, tokio	107	11.4	17 513
Zig	stdlib	67	9.4	26 144
Kotlin	stdlib	5.8	4.3	21 422
OCaml	stdlib	3.5	0.3	24 709
Nim	stdlib	2.7	1.2	22 069
C++	stdlib	0.8	1.1	24 017
Odin	stdlib	0.3	1.7	23 677
Swift	stdlib	0.1	9.9	14 837
C	stdlib	0.0	0.1	27 776
Java	stdlib	0.0	0.4	22 763
Assembly	stdlib	0.0	0.1	22 051
Python	stdlib	0.0	0.0	27 272
JavaScript	stdlib	0.0	0.0	24 608
TypeScript	stdlib	0.0	0.0	25 275

Odin was the only language whose agents failed hidden tests. Swift and C# also cost more than their code size predicts, but for a reason outside the languages: on the test machine, which has Apple's Command Line Tools but no Xcode, SwiftPM aborts at start-up and .NET Native AOT cannot link, so the agents' first builds failed (only 8% and 8% passed on the first try) until they found a workaround (calling `swiftc` directly; a non-AOT build).

H4 (front end): confirmed. With a different component set, written by agents, and with Opus 5.5's tokenizer, the article's ordering holds and the ratios are close (Table 6). All three frameworks worked in every run. In the mini app the store choices were idiomatic: Context + `useReducer` in React, Pinia or `reactive()` in Vue, runes in Svelte.

H5 (Rust + React): defensible, not optimal. For the agent, Rust was among the cheapest back ends (US$ 0.83). React was the most expensive front end. The full feature (`kvhttp` server + client component) took 3 960 tokens in Rust + React, against 3 702 in Rust + Svelte and 8 619 in Zig + Vue. In performance, all native languages were within the same order of magnitude under this load (Table 2).

Cheap to write is not fast to run. The two rankings run against each other: the rank correlation between cost and speed score is $\rho = 0.63$ (Table 4), so the languages that cost more to write tend to produce faster code. In speed index (1.00 = fastest everywhere), the three cheapest languages for the agent produced the slowest code: JavaScript 5.65, TypeScript 6.14 and Python 6.52. Assembly (1.06), Zig (1.25) and C (1.32) were the fastest, and among the most expensive to write. Go (1.60) and Rust (2.44) were the best compromise: both in the cheaper half to write and the faster half to run.

Specialised tracks. All 60 runs passed their hidden checks on the first verification (Table 5). Within a track the languages cost about the same: Dafny, Lean 4 and Rocq between US$ 0.25 and 0.28; Vyper and Solidity US$ 0.41 and 0.42; Circom and Noir US$ 0.46 and 0.47. The widest gap

Table 3: Final binary performance (median of 5 runs), including process start-up (and JVM start-up for Java and Kotlin).

Language	wordfreq ms	lru ms	calc ms	lru RSS MB
Assembly	94	36	18	29
Zig	119	56	21	28
Odin	158	59	21	39
C	135	52	26	30
Go	169	80	26	20
Rust	244	180	31	42
Nim	308	245	39	58
OCaml	366	241	54	18
Haskell	1 100	218	89	117
Swift	194	525	91	8.5
C#	315	330	91	113
Java	182	290	149	138
Kotlin	230	349	149	152
JavaScript	837	350	188	262
TypeScript	861	365	251	285
C++	200	223	406	43
Python	607	346	516	184

was in scientific computing, where R (US$ 0.31) was cheaper than Julia and Octave (US$ 0.46 and 0.48). Noir passed its examples on the first try in only a third of the runs, but recovered at little cost. Theorem proving was the cheapest work in the whole study: the agent wrote short inductive proofs and the checkers answered in about a second. Solidity's cold build took 80–100 MB per task, mostly a `solc` download into the empty home directory (and, in one run, OpenZeppelin); this partly reflects installation, since Vyper's compiler was already installed.

6 Limitations

Round 3 itself found two harness bugs, both fixed before the repair round, with the affected cases re-verified. (i) The filter that separates source from build output dropped `bin/` directories, which are output in C# but source in dune (OCaml); the three OCaml servers appeared not to build. (ii) On macOS, the first launch of a freshly built binary goes through a system check that took over 3 s for the 48 MB Haskell binary; the verifier now launches each binary once before the suite. No failure attributed to an agent comes from these two cases.

Swift and C# are confounded by the test machine: SwiftPM crashes (a mismatched `llbuild` in the Command Line Tools) and Native AOT needs Xcode, so their costs include finding a workaround.

The specialised tracks have two small tasks each. The zero-knowledge checks are behavioural: they show that valid inputs are provable and that the listed invalid inputs are not, but a circuit could still be under-constrained for inputs the suite does not try.

The `kvhttp` load comes from a single Node.js client, which likely caps the measured throughput: every language scored between 53 and 100 there, so the speed ranking is decided mostly by the three command-line tasks. The score weighs the four tasks equally.

Three repetitions per case: small differences are noise. One model, one harness, one machine (macOS arm64). Tasks take minutes, not days; in large projects reading existing code weighs more, which should widen the advantage of concise languages. Cost uses list price and includes cache reads. The front-end check is a smoke test, not a full suite. Zig and Odin change their APIs often; another model with another training cut-off could reverse H1.

Table 4: Performance ranking. Points per task: 100× the fastest time divided by the language's (`kvhttp`: its throughput divided by the best), so 100 is the fastest. Score: geometric mean of the four. ×fastest: speed index, 100/score.

#	Language	wordfreq	lru	calc	kvhttp	Score	×fastest
1	Assembly	100	100	100	79	**94.4**	1.06
2	Zig	79	64	86	94	**80.0**	1.25
3	C	70	69	69	100	**76.0**	1.32
4	Odin	59	61	86	85	**71.8**	1.39
5	Go	56	45	69	88	**62.5**	1.60
6	Rust	39	20	58	63	**41.0**	2.44
7	Nim	31	15	46	79	**35.8**	2.79
8	OCaml	26	15	33	89	**32.7**	3.06
9	Java	52	12	12	82	**28.2**	3.54
10	C#	30	11	20	81	**26.9**	3.72
11	Kotlin	41	10	12	77	**25.0**	3.99
12	Swift	48	7	20	53	**24.3**	4.11
13	C++	47	16	4	86	**23.2**	4.31
14	Haskell	9	17	20	85	**22.2**	4.51
15	JavaScript	11	10	10	89	**17.7**	5.65
16	TypeScript	11	10	7	91	**16.3**	6.14
17	Python	15	10	3	98	**15.3**	6.52

Table 5: Specialised tracks. US$ is the sum of the per-task medians over the track's two tasks, so it is comparable within a track, not with Table 1.

Track	Language	Solved	Repaired	1st try	US$	Output	Code	Turns
Scientific computing	R	6/6	6/6	100%	0.31	4 329	2 135	10
	Julia	6/6	6/6	83%	0.46	9 867	5 386	10
	Octave	6/6	6/6	100%	0.48	8 596	1 923	20
Theorem proving	Dafny	6/6	6/6	100%	0.25	2 285	2 260	7
	Lean 4	6/6	6/6	83%	0.28	2 777	1 892	9
	Rocq	6/6	6/6	100%	0.28	2 944	2 043	8
Smart contracts	Vyper	6/6	6/6	100%	0.41	5 757	3 973	9
	Solidity	6/6	6/6	100%	0.42	5 947	3 432	8
Zero-knowledge circuits	Circom	6/6	6/6	83%	0.46	8 613	3 032	13
	Noir	6/6	6/6	33%	0.47	7 330	3 169	12

7 Conclusion

With Claude Opus 5.5 the language hardly decided whether the agent succeeded (only Odin failed a hidden test, and a repair agent fixed it on the first attempt); it decided how much success cost, by up to 4.8×. The cheapest were Python, TypeScript, JavaScript and Kotlin: widely used languages with little ceremony. Cost follows code size, as the series predicted for snippets, but also how well the model knows the language version and its standard library, and how smoothly the toolchain works on the machine. For people who program with agents the practical question is not only "which language is most concise" but "which language does the model already know well, in the version I use". Data, code and every solution are in the repository. Suggestions, corrections and pull requests that improve the method are welcome, at the repository or by email to `abeilice@ufmg.br`.

Table 6: Front end. Ratios to Svelte (= 1): the article [11] and our measurement.

	Article cl100k	Ours cl100k	Ours Opus	ui tokens	shop tokens	Store in shop (agent's choice)
React	1.254	1.217	1.264	2 599	3 348	Context + useReducer
Svelte	1.000	1.000	1.000	2 056	2 543	Svelte runes ($state)
Vue	1.115	1.090	1.108	2 278	2 581	Vue reactive(), Pinia

Use of generative AI

The benchmark harness, the agents under test and a first draft of this manuscript were produced with Claude (Anthropic) under the author's direction. The author defined the questions and hypotheses, reviewed the code, data and text, and takes full responsibility for the content.

References

[1] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes and J. Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proc. ACM SIGPLAN Int. Conf. on Software Language Engineering (SLE)*, 2017. doi:10.1145/3136014.3136031.

[2] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes and J. Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[3] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000.

[4] S. Nanz and C. A. Furia. A comparative study of programming languages in Rosetta Code. In *Proc. 37th Int. Conf. on Software Engineering (ICSE)*, pages 778–788, 2015.

[5] B. Ray, D. Posnett, V. Filkov and P. Devanbu. A large scale study of programming languages and code quality in GitHub. In *Proc. 22nd ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (FSE)*, pages 155–165, 2014.

[6] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek and J. Vitek. On the impact of programming languages on code quality: a reproduction study. *ACM Transactions on Programming Languages and Systems*, 2019.

[7] M. Chen et al. Evaluating large language models trained on code. *arXiv:2107.03374*, 2021.

[8] F. Cassano et al. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.

[9] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press and K. Narasimhan. SWE-bench: can language models resolve real-world GitHub issues? In *Proc. Int. Conf. on Learning Representations (ICLR)*, 2024.

[10] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan and O. Press. SWE-agent: agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[11] A. Kucherenko. *We Measured the LLM Token Cost of 5 Frontend Frameworks: Angular Costs 38% More Than Svelte*. HackerNoon, 2026.

[12] A. Kucherenko. *We Measured the LLM Token Cost of 5 Languages: TypeScript Costs 31% More Than JavaScript*. HackerNoon, 2026.

[13] Datacurve. *DeepSWE: a long-horizon software engineering benchmark*. `https://deepswe.datacurve.ai/`, 2026.