Computation and Performance Studies
Volume 1 · Number 1 · 2026

What an Input-Length Budget Misses: Traced Reference Execution on HumanEval

Spencer Franklin[1] · Nathan Reed[2] · Benjamin Lewis[3]

[1] Iceland · [2] Canada · [3] United States

Original publication: October 28, 2026 | Article 614

Abstract

Short coding inputs can still trigger substantial execution work. We extracted unique literal calls from HumanEval test text and traced screened canonical functions under fixed guards. Of 1,088 attempted calls, 1,084 completed and four reached the source-line limit. Completed calls executed between one and 54,713 Python source lines, with a median of nine. Input byte length had an imprecise positive association with log line count; reference-code size and a loop indicator also contributed to a model with R-squared 0.330. These are implementation-specific tracing results, not elapsed-time measurements. They suggest that input length alone gives an incomplete description of this evaluation-work proxy, while leaving hardware cost and model-generation expense unmeasured.

Bounded execution of screened canonical functions on directly literal test inputs. The study does not measure coding-model token use, wall-clock speed, or repair correctness.

What an Input-Length Budget Misses: Traced Reference Execution on HumanEval

Bounded execution of screened canonical functions on directly literal test inputs. The study does not measure coding-model token use, wall-clock speed, or repair correctness.

Abstract

Short coding inputs can still trigger substantial execution work. We extracted unique literal calls from HumanEval test text and traced screened canonical functions under fixed guards. Of 1,088 attempted calls, 1,084 completed and four reached the source-line limit. Completed calls executed between one and 54,713 Python source lines, with a median of nine. Input byte length had an imprecise positive association with log line count; reference-code size and a loop indicator also contributed to a model with R-squared 0.330. These are implementation-specific tracing results, not elapsed-time measurements. They suggest that input length alone gives an incomplete description of this evaluation-work proxy, while leaving hardware cost and model-generation expense unmeasured.

1. A coding evaluation has more than one resource budget

Generating a program and evaluating a program consume different resources. A prompt may be short while its supplied input leads to many iterations of a reference implementation. Conversely, a long literal string can be processed by one compact built-in operation. Counting input bytes therefore cannot by itself define the amount of computation performed by a test. This article examines one directly observable proxy for that computation in a public coding resource.

The research question is how literal-input size and reference-code structure relate to executed Python source lines. We trace the canonical functions supplied with HumanEval, introduced in Chen et al., 2021, rather than model-generated programs. The data concern reference execution under a declared interface and guard policy. They cannot establish the cost of producing code, the quality of an assistant's edits, or the correctness of an unobserved candidate program.

The coding-assistant preference setting in Tao and Kamata, 2026 motivates distinguishing the resources used to obtain an evaluation signal from the learning procedure consuming that signal. We do not reproduce its fine-tuning or create preference pairs. The empirical contribution is narrower: an auditable execution record can show how much variation a simple input-length budget fails to describe, without claiming an improvement in a coding model.

2. Define the work proxy before interpreting it

We use the tracing interface documented in Python Software Foundation, 2026 to count source-line events whose code belongs to the captured canonical functions. Repeatedly executing the same line contributes repeatedly. Calls and generator or comprehension execution can therefore add events without adding new source text. Python library implementation lines outside the captured source filename are not counted. This is a specified instrumentation measure, not every instruction executed by the machine.

The distinction is substantial. A single source line can call sorting, regular-expression processing, or another built-in with considerable implementation work. The hardware-performance perspective in Williams et al., 2009 concerns effects not recoverable from our event counter. Technical-debt discussion in Sculley et al., 2015 similarly reminds us that a convenient local measure can omit important system

behavior. We consequently report no speedup, memory saving, or financial cost from a lower traced count.

Numerical accuracy and execution expense are also separate. The finite-arithmetic issues discussed by Higham, 2002 and Goldberg, 1991 do not disappear when a program executes few lines. We do not compare numerical error or choose a cheaper approximation. The endpoint is how the supplied reference source runs on selected supplied inputs. Its adequacy as a budget proxy must remain distinct from correctness, arithmetic stability, and application utility.

3. Select literal calls without executing the test harness

The captured HumanEval file contains 164 tasks, each with a prompt, a canonical solution, an entry-point name, and test text. We parse the combined prompt and canonical solution as Python syntax. The test text is parsed separately to locate calls to its candidate placeholder. Only arguments directly recoverable as Python literals are eligible. Generated arguments, variable-dependent arguments, and surrounding test loops are not executed to create more cases.

That restriction gives a reproducible, bounded interface but changes the evaluated population. Some benchmark behavior is exercised only through generated or indirect inputs. The analysis therefore does not claim complete hidden-test coverage or a full HumanEval pass rate. Evaluation concerns raised by Liu et al., 2023 are relevant precisely because a subset of test evidence should not be advertised as semantic adequacy of a task or solution.

We remove repeated literal inputs within the same task using a canonical serialized representation of positional and keyword arguments. An input must serialize to no more than two thousand ASCII JSON bytes, and each numeric value must have absolute magnitude at most ten thousand. These limits are declared execution guards, not descriptions of every valid input in the task's prose specification. They exclude sixteen unique candidate inputs after deduplication.

The extracted subset contains 1,146 literal-call occurrences after the source gate. Forty-two repeated inputs are removed, leaving 1,104 unique inputs before the guard exclusion and 1,088 attempted calls afterward. Reporting these stages avoids treating every assertion occurrence as an independent observation or silently describing omitted dynamic tests as executed. The same source snapshot and extraction rules define the dataset available for reproduction.

4. Screen the reference source and bound its execution

The source gate permits reviewed function definitions, imports from a small standard-module set, descriptive strings, and literal module constants. Dynamic evaluation, file access, and system access are excluded. One task, HumanEval/160, is rejected because its reference uses dynamic expression evaluation. A module string asking for more tests is treated only as data stored in the source; it is not followed as an instruction to change the experiment.

Arguments are copied before each call, so a function cannot mutate an input that is later reused. The trace stops a call after 100,000 executed source lines, with an additional two-second trace guard. Generator results are materialized under an output cap. The experiment invokes only the screened canonical functions, not a repository-wide evaluation driver or newly generated code. These choices keep the measured activity aligned with the declared reference-execution question.

Four calls reach the source-line budget. The other 1,084 complete without a recorded exception, and none reaches the trace-time guard. We retain all attempted-call records, including the censored ones, while fitting the cost regression to completed calls only. The resulting outcome distribution is conditional on

completion under the guard. A budget-reaching call is not assigned its unknown eventual cost or described as a proof of nontermination.

Repository-extraction practices in Bird et al., 2009, empirical-review guidance in Kitchenham et al., 2002, and the code-review context of Bacchelli and Bird, 2013 support explicit accounting of source units and exclusions. Defect categories in Herzig et al., 2013 concern another object; our canonical executions are not a sample of real software faults. The bounded benchmark interface can inform an evaluation-work description while leaving industrial program behavior and failure mechanisms outside its scope.

5. Record source structure as retrospective information

For each task, we count syntax-tree nodes in its canonical entry function. This is a compact source-size measure. We also record whether that function contains a loop or comprehension and whether it directly calls itself. These are properties of the supplied reference implementation, not features available from a user's prompt alone. A system choosing an evaluation budget before seeing any implementation would need another model to obtain comparable information.

The control-flow metric discussed by McCabe, 1976 is relevant background, but our loop indicator is much coarser and is not a cyclomatic-complexity calculation. Syntax-node count is likewise not a proof of asymptotic time complexity. Two implementations of the same mathematical function can have different trees, line layouts, and built-in calls. The analysis describes the captured implementation rather than an invariant computational requirement of the underlying task.

Only two tasks with completed calls have a directly self-recursive reference. We therefore do not report a separate clustered recursion coefficient as if many independent recursive tasks supported it. The flag remains in the processed data for transparent description. This choice preserves a limitation of the available task set instead of allowing a large number of calls within two tasks to create a misleading impression of broad independent support.

6. Model completed-call line counts

The response is the logarithm of one plus executed source lines. Predictors are the logarithm of one plus serialized input bytes, the logarithm of reference syntax-node count, and the loop/comprehension indicator. The transformation moderates a strongly skewed positive count distribution. Count-model perspectives in Cameron and Trivedi, 1998 offer other approaches to such outcomes, but no Poisson or negative-binomial comparison is fitted in this article.

Calls from one task share a reference implementation and many aspects of test design. We use task-clustered covariance following the dependence considerations in Cameron and Miller, 2015, with 160 task clusters, the recorded correction, and a t reference using 159 degrees of freedom. Heteroskedasticity correction in White, 1980 alone would not account for that repeated implementation. Related tasks can still share algorithmic structure across clusters, which remains an external-validity limit.

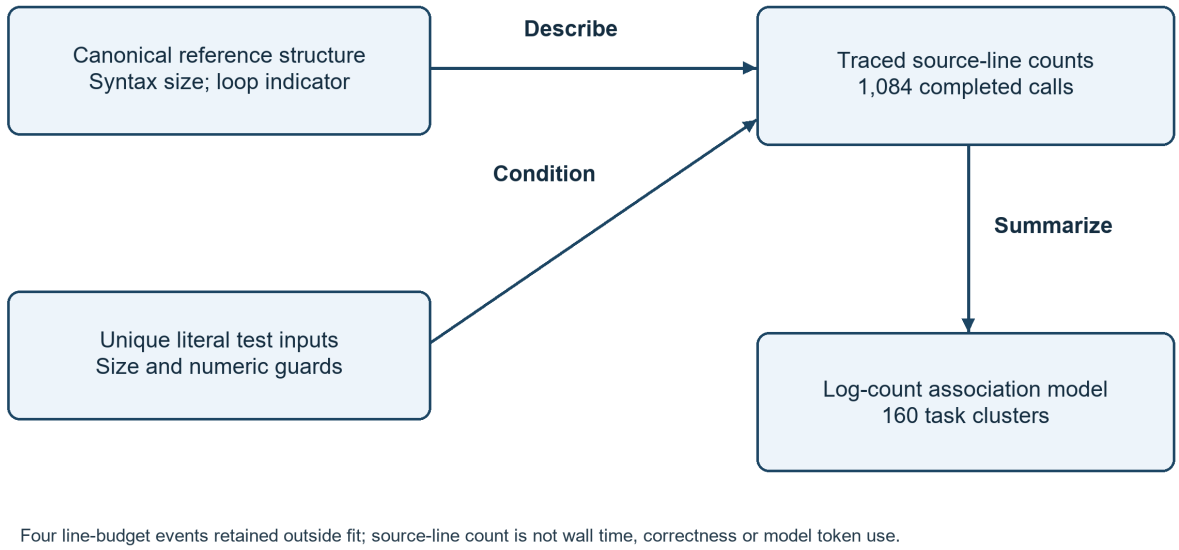


Figure 1. Screened canonical source and unique guarded literal inputs produce completed-call line counts for a task-clustered association model.

Table 1. Input-size and reference-structure associations with log transformed executed source lines.

Model	Term	Estimate (SE)	95% CI	N
M1	Log(1 + input bytes)	0.3246 (0.1643)	[0.0002, 0.6490]	1084
M1	Log reference syntax nodes	0.8868 (0.1369)	[0.6164, 1.1571]	1084
M1	Loop/comprehension present	1.0388 (0.1800)	[0.6833, 1.3944]	1084
M1: fit	R-squared = 0.3301	-	-	1084

Model labels: M1 = Reference execution-line association.

This is an explanatory association within the captured completed-call sample. It is not a held-out resource predictor. Validation issues in Roberts et al., 2017 and model-selection concerns in Varma and Simon, 2006 would matter for using the coefficients to budget new tasks. A future prediction study should separate tasks during development and evaluation rather than distribute calls from the same implementation randomly across both sides.

7. Wide execution variation remains under small input guards

Completed calls execute between one and 54,713 source lines, with a median of nine. The large range exists even after the declared numeric and serialized-size guards. It shows that the selected inputs do not imply a uniform source-execution burden. The four capped calls also indicate that the completed-call maximum is not a bound on every attempted reference's eventual work.

The log-input-size coefficient is 0.32462, with standard error 0.16426 and interval 0.00020 to 0.64903. The interval is wide and sits close to zero at its lower endpoint. We describe an imprecise positive association, rather than presenting the estimate as a sharply established resource law. Because both response and input use one-plus transformations, the coefficient should not be treated as an exact elasticity at every small count.

The log-syntax-size coefficient is 0.88676, with interval 0.61641 to 1.15711. The loop indicator has coefficient 1.03884, with interval 0.68330 to 1.39439. Model R-squared is 0.3301. These estimates indicate that source structure adds descriptive information in the selected sample, while leaving substantial variation unexplained. They do not establish that adding a loop to an arbitrary program causes a fixed multiplicative change in its runtime.

8. Coding-model evaluation needs additional resource measurements

Systems such as Li et al., 2023, Guo et al., 2024, Guo et al., 2020, Fried et al., 2022, and Le et al., 2022 provide different code-generation or representation settings. None is run here. A canonical reference can differ greatly from a model-generated implementation in algorithm choice, formatting, imported libraries, and error behavior. The current coefficients cannot be assigned to those models as estimated generation or execution costs.

Line counts are also formatting-sensitive. Splitting an expression across source lines or implementing a loop through a built-in can change the counter without an equivalent change in physical work. Measurement-error reasoning in Carroll et al., 2006 helps clarify the proxy's limits: the observed counter is precise under its instrumentation rule, but the relationship between that counter and a desired hardware-cost construct remains imperfect. We do not repair that gap by calling the proxy compute without qualification.

Scoring principles in Gneiting and Raftery, 2007 distinguish the measured outcome from a quality objective. A function can complete cheaply and still give an incorrect answer on another input. We do not execute the surrounding assertions to assign correctness labels, so completion here means completion under the guard, not a benchmark pass. This preserves the difference between acquiring a resource measurement and certifying the semantic adequacy of the reference or a candidate.

9. What can be learned from a bounded execution record

Influence analysis motivated by Cook, 1977 could identify tasks with unusually large leverage or residuals. Such tasks have not been removed to improve the fit. Guidance in Wasserstein and Lazar, 2016 favors presenting the imprecise input coefficient and the proxy limitations alongside the more clearly separated structure coefficients. Multiplicity considerations in Benjamini and Hochberg, 1995 would apply if many source metrics and extraction policies were searched, rather than treating one selected regression as a universal account of computational cost.

An operational evaluation budget could combine more direct measurements: elapsed time, memory, input construction, and the consequences of a timeout. It would also need to account for generated programs that fail or behave unexpectedly. The present study supplies none of those missing outcomes by assumption. It contributes a preserved call-level record showing why a single easily counted input quantity is an incomplete description of one reproducible execution-work proxy.

The attempted-call record documents literal extraction, deduplication, input guards, the line-event definition, completion restriction, and the fitted subset (Gebru et al., 2021). Runtime observability also appears in Yan, 2026, whose stated topic is production-failure detection through distributed tracing. Our tracer observes canonical Python source-line events within guarded standalone calls, rather than interservice context, latency, or failures in deployment. Its counts must remain bound to that instrumentation boundary. Source structure describes part of their variation beyond input bytes; translating the relationship into hardware budgets or fault causes needs evidence absent from this experiment.

References

1. Bird, Christian; Rigby, Peter C.; Barr, Earl T.; Hamilton, David J.; German, Daniel M.; Devanbu, Premkumar. (2009). The promises and perils of mining git. https://doi.org/10.1109/MSR.2009.5069475.
2. White, Halbert. “A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity.” *Econometrica*, vol. 48, no. 4, 1980, pp. 817-838. https://doi.org/10.2307/1912934.
3. Varma, Sudhir and Richard Simon. “Bias in error estimation when using cross-validation for model selection.” *BMC Bioinformatics*, vol. 7, no. 1, 2006. https://doi.org/10.1186/1471-2105-7-91.
4. Yan, Biyun. (2026). From Observability to Causality: Detecting Production Failures in Microservices with Distributed Tracing. 2026 9th International Conference on Computer Information Science and Application Technology (CISAT), 1522-1525. https://doi.org/10.1109/CISAT71030.2026.11682113.
5. Li, Raymond; Allal, Loubna Ben; Zi, Yangtian; Muennighoff, Niklas; Kocetkov, Denis; Mou, Chenghao; Marone, Marc; Akiki, Christopher; Li, Jia; Chim, Jenny; Liu, Qian; Zheltonozhskii, Evgenii; Zhuo, Terry Yue; Wang, Thomas; Dehaene, Olivier; Davaadorj, Mishig; Lamy-Poirier, Joel; Monteiro, João; Shliazhko, Oleh; Gontier, Nicolas; Meade, Nicholas; Zebaze, Armel; Yee, Ming-Ho; Umapathi, Logesh Kumar; Zhu, Jian; Lipkin, Benjamin; Oblokulov, Muhtasham; Wang, Zhiruo; Murthy, Rudra; Stillerman, Jason; Patel, Siva Sankalp; Abulkhanov, Dmitry; Zocca, Marco; Dey, Manan; Zhang, Zhihan; Fahmy, Nour; Bhattacharyya, Urvashi; Yu, Wenhao; Singh, Swayam; Luccioni, Sasha; Villegas, Paulo; Kunakov, Maxim; Zhdanov, Fedor; Romero, Manuel; Lee, Tony; Timor, Nadav; Ding, Jennifer; Schlesinger, Claire; Schoelkopf, Hailey; Ebert, Jan; Dao, Tri; Mishra, Mayank; Gu, Alex; Robinson, Jennifer; Anderson, Carolyn Jane; Dolan-Gavitt, Brendan; Contractor, Danish; Reddy, Siva; Fried, Daniel; Bahdanau, Dzmitry; Jernite, Yacine; Ferrandis, Carlos Muñoz; Hughes, Sean; Wolf, Thomas; Guha, Arjun; von Werra, Leandro; de Vries, Harm. (2023). StarCoder: may the source be with you!. arXiv:2305.06161. https://arxiv.org/abs/2305.06161.
6. Carroll, Raymond J., David Ruppert, Leonard A. Stefanski, and Ciprian M. Crainiceanu. “Measurement Error in Nonlinear Models.” 2006. https://doi.org/10.1201/9781420010138.
7. Cameron, A. Colin; Trivedi, Pravin K.. (1998). Regression Analysis of Count Data. Cambridge University Press. ISBN 9780521632010. https://www.cambridge.org/9780521632010.
8. Kitchenham, Barbara A.; Pfleeger, Shari Lawrence; Pickard, Lesley M.; Jones, Peter W.; Hoaglin, David C.; El Emam, Khaled; Rosenberg, Jarrett. (2002). Preliminary guidelines for empirical research in software engineering. https://doi.org/10.1109/TSE.2002.1027796.
9. Herzig, Kim; Just, Sascha; Zeller, Andreas. (2013). It’s not a bug, it’s a feature: How misclassification impacts bug prediction. https://doi.org/10.1109/ICSE.2013.6606585.
10. Python Software Foundation. (2026). Python 3.12 documentation: sys.settrace. Documentation snapshot captured 7 November 2026. https://docs.python.org/3.12/library/sys.html#sys.settrace.
11. Goldberg, David. (1991). What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys. https://doi.org/10.1145/103162.103163.
12. Sculley, D., et al. "Hidden Technical Debt in Machine Learning Systems." *Advances in Neural Information Processing Systems*, vol. 28, 2015. https://papers.nips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html
13. Le, Hung; Wang, Yue; Gotmare, Akhilesh Deepak; Savarese, Silvio; Hoi, Steven C. H.. (2022). CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780. https://arxiv.org/abs/2207.01780.
14. Guo, Daya; Ren, Shuo; Lu, Shuai; Feng, Zhangyin; Tang, Duyu; Liu, Shujie; Zhou, Long; Duan, Nan; Svyatkovskiy, Alexey; Fu, Shengyu; Tufano, Michele; Deng, Shao Kun; Clement, Colin; Drain, Dawn; Sundaresan, Neel; Yin, Jian; Jiang, Daxin; Zhou, Ming. (2020). GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv:2009.08366. https://arxiv.org/abs/2009.08366.
15. Tao, Hongmo; Kamata, Hiroyuki. (2026). Teaching Coding Assistants What to Change: Fine-Tuning from Pairwise Preferences. Proceedings of the Second Workshop for REsearch on Agent Language Models (REALM 2026), 1363-1374. https://doi.org/10.18653/2026.realm-1.95.
16. Roberts, David R., et al. “Cross-validation strategies for data with temporal, spatial, hierarchical, or phylogenetic structure.” *Ecography*, vol. 40, no. 8, 2017, pp. 913-929. https://doi.org/10.1111/ecog.02881.
17. Fried, Daniel; Aghajanyan, Armen; Lin, Jessy; Wang, Sida; Wallace, Eric; Shi, Freda; Zhong, Ruiqi; Yih, Wen-tau; Zettlemoyer, Luke; Lewis, Mike. (2022). InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999. https://arxiv.org/abs/2204.05999.

18. Cameron, A. Colin, and Douglas L. Miller. “A Practitioner’s Guide to Cluster-Robust Inference.” *Journal of Human Resources*, vol. 50, no. 2, 2015, pp. 317-372. https://doi.org/10.3368/jhr.50.2.317.

19. Higham, Nicholas J.. (2002). Accuracy and Stability of Numerical Algorithms: Second Edition. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718027.

20. Guo, Daya; Zhu, Qihao; Yang, Dejian; Xie, Zhenda; Dong, Kai; Zhang, Wentao; Chen, Guanting; Bi, Xiao; Wu, Y.; Li, Y. K.; Luo, Fuli; Xiong, Yingfei; Liang, Wenfeng. (2024). DeepSeek-Coder: When the Large Language Model Meets Programming -- The Rise of Code Intelligence. arXiv:2401.14196. https://arxiv.org/abs/2401.14196.

21. Cook, R. Dennis. (1977). Detection of Influential Observation in Linear Regression. Technometrics. https://doi.org/10.1080/00401706.1977.10489493.

22. Benjamini, Yoav and Yosef Hochberg. “Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing.” *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 57, no. 1, 1995, pp. 289-300. https://doi.org/10.1111/j.2517-6161.1995.tb02031.x.

23. Liu, Jiawei; Xia, Chunqiu Steven; Wang, Yuyao; Zhang, Lingming. (2023). Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210. https://arxiv.org/abs/2305.01210.

24. Gebru, Timnit, et al. "Datasheets for datasets" *Communications of the ACM*, 2021. https://doi.org/10.1145/3458723.

25. Bacchelli, Alberto; Bird, Christian. (2013). Expectations, outcomes, and challenges of modern code review. 2013 35th International Conference on Software Engineering (ICSE). https://doi.org/10.1109/icse.2013.6606617.

26. Williams, Samuel; Waterman, Andrew; Patterson, David. (2009). Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM. https://doi.org/10.1145/1498765.1498785.

27. Wasserstein, Ronald L.; Lazar, Nicole A.. (2016). The ASA Statement on p -Values: Context, Process, and Purpose. The American Statistician. https://doi.org/10.1080/00031305.2016.1154108.

28. Gneiting, Tilmann; Raftery, Adrian E. (2007). Strictly Proper Scoring Rules, Prediction, and Estimation. Journal of the American Statistical Association. https://doi.org/10.1198/016214506000001437.

29. Chen, Mark; Tworek, Jerry; Jun, Heewoo; Yuan, Qiming; Pinto, Henrique Ponde de Oliveira; Kaplan, Jared; Edwards, Harri; Burda, Yuri; Joseph, Nicholas; Brockman, Greg; Ray, Alex; Puri, Raul; Krueger, Gretchen; Petrov, Michael; Khlaaf, Heidy; Sastry, Girish; Mishkin, Pamela; Chan, Brooke; Gray, Scott; Ryder, Nick; Pavlov, Mikhail; Power, Alethea; Kaiser, Lukasz; Bavarian, Mohammad; Winter, Clemens; Tillet, Philippe; Such, Felipe Petroski; Cummings, Dave; Plappert, Matthias; Chantzis, Fotios; Barnes, Elizabeth; Herbert-Voss, Ariel; Guss, William Hebgen; Nichol, Alex; Paino, Alex; Tezak, Nikolas; Tang, Jie; Babuschkin, Igor; Balaji, Suchir; Jain, Shantanu; Saunders, William; Hesse, Christopher; Carr, Andrew N.; Leike, Jan; Achiam, Josh; Misra, Vedant; Morikawa, Evan; Radford, Alec; Knight, Matthew; Brundage, Miles; Murati, Mira; Mayer, Katie; Welinder, Peter; McGrew, Bob; Amodei, Dario; McCandlish, Sam; Sutskever, Ilya; Zaremba, Wojciech. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374. https://arxiv.org/abs/2107.03374.

30. McCabe, T.J.. (1976). A Complexity Measure. IEEE Transactions on Software Engineering. https://doi.org/10.1109/tse.1976.233837.