Zero-Shot vs. Chain-of-Thought Prompting for Python Code Generation: Separating Execution, Functional Correctness, and Judged Quality

Sean Lester S. Miranda

College of Computer Studies, AMA Computer University, Philippines

Abstract

Zero-shot and chain-of-thought (CoT) prompting can produce different program structures, validation logic, and explanations, but those differences need not translate into improved correctness. We present a small case study using ChainZCode, a code-evaluation toolkit with a retained collection of 180 Python implementations: 30 algorithmic tasks, three models (GPT-4o, Claude 3.5 Sonnet, and Gemini 2.0 Flash), and two styles. The evaluation separates retrospective external callable checks under explicit task contracts, whole-script completion, and five-criterion model-assisted quality scores. Of the 180 implementations, 178 pass the callable suites and 177 complete as whole scripts. GPT and Claude pass all 30 callable suites in each style; Gemini passes 29 in each, with different failed tasks. The failures distinguish a maximum-subarray convention, a syntax error, and defects in generated self-tests. Paired CoT-minus-zero-shot judge-score differences are -0.124, $+0.180$, and -0.497 points out of 20 for GPT, Claude, and Gemini, respectively. Gemini's difference becomes $+0.171$ when the comparison requires two completed assessments, a sign reversal showing why failure policy and assessment coverage must accompany quality averages. Process completion, checked functional behavior, and judged quality capture different outcomes in this case study. The study contains one saved implementation per condition; its conclusions concern these programs rather than variation across newly sampled generations.

Keywords: Python programs; zero-shot prompting; chain-of-thought; functional testing; model-assisted evaluation.

1 Introduction

Prompting a large language model to explain an algorithm before producing code may affect how it handles inputs, organizes a solution, or constructs tests. Whether those changes improve the resulting program is an empirical question. An implementation can execute normally while violating an input/output requirement, and a correct function can be packaged with a faulty demonstration or test. Model-assisted quality ratings provide another view of the code, but are distinct from observed execution and externally checked behavior.

Execution-based benchmarks provide a foundation for evaluating generated code. HumanEval measures functional correctness against tests [2], while APPS covers programming problems with a wider range of requirements and difficulty [6]. EvalPlus demonstrates that additional tests can expose previously missed errors and change apparent model rankings [11]. These findings motivate comparisons that identify both what is being measured and which test conditions support a reported success.

ChainZCode is the project's Python framework for organizing and evaluating model-generated code. Here the name also identifies the associated collection of saved solutions and evaluation materials. This case study uses that collection to compare zero-shot and CoT-style solutions across 30 data-structure and algorithm tasks. The collection contains one saved implementation for each task and prompting style from GPT-4o, Claude 3.5 Sonnet, and Gemini 2.0 Flash, for 180 implementations in total. Tasks include searching, sorting, arithmetic, arrays, strings, trees, shortest paths, and dynamic programming. The retained zero-shot prompt examples contain direct programming requests; the CoT examples additionally ask for an algorithmic outline and its implementation. The available evidence supports comparisons within this collection, with model variants and generation-provenance limits specified in the study design.

The comparison uses three complementary outcomes. First, external reference checks evaluate callable behavior under explicit input domains and return conventions. Second, whole-script execution includes the generated implementations' demonstrations and assertions. Third, the recorded quality rubric scores execution correctness, output correctness, input handling, completeness, and requirement compliance. Keeping these outcomes separate makes it possible to distinguish an algorithmic counterexample from a malformed script, a faulty generated test, or an unavailable judge assessment.

We address three research questions:

1. **RQ1: Functional behavior and execution.** How do zero-shot and CoT implementations compare on external callable checks and whole-script completion across the matched tasks?
2. **RQ2: Failure modes and task contracts.** Which implementation, formatting, and generated-test failures occur, and how do input-domain or API conventions affect their interpretation?
3. **RQ3: Judged quality and sensitivity.** How do the paired quality scores differ between styles, and how sensitive are those differences to missing assessments and failure penalties?

The contributions are a task-matched comparison of the saved implementations, a distinction between callable behavior and complete-script outcomes, and a transparent analysis of quality-score inclusion rules. The main methodological finding is that failure handling can reverse a paired judge-score difference, while callable and whole-script outcomes expose different failure modes.

2 Related work

Functional evaluation. HumanEval [2] evaluates programs synthesized from function descriptions, and APPS [6] covers natural-language programming problems with functional tests. MBPP [1] adds short Python tasks described in natural language. EvalPlus [11] shows that additional test inputs can uncover missed defects and change model rankings. These studies motivate our use of external reference checks alongside script execution. Our task collection provides a smaller setting for inspecting how evaluation criteria affect comparisons between prompting styles. LiveCodeBench [7] collects new contest problems over time to address contamination and evaluates several code-related capabilities. BigCodeBench [16] tests library use and complex instructions beyond isolated textbook algorithms. These broader benchmarks motivate extending the task domains considered here.

ClassEval [4] examines class-level generation and shows why standalone method performance need not transfer to interdependent class behavior. The present collection contains a few tree classes but primarily consists of isolated algorithm exercises. Its tree checks include structural properties, while multi-file integration and repository maintenance remain outside scope.

Prompting strategies. Wei et al. [13] studied prompting with intermediate reasoning steps. The CoT-style instruction used in the retained examples asks for an algorithmic plan followed by code, without few-shot demonstrations. It is an instruction-based adaptation rather than a replication of the original protocol. A planning instruction can affect comments, validation logic, and self-tests as well as the algorithm itself, so those outcomes should be examined separately. Structured CoT (SCoT) [9] organizes intermediate plans around sequence, branch, and loop structures. Self-planning code generation [8] separates planning from implementation. Our retained instructions share the plan-before-code idea, but do not reproduce either protocol.

AlphaCode [10] combines large-scale candidate sampling with behavioral filtering for competitive programming. The present evaluation does not measure candidate-selection budgets. InCoder [5] supports infilling and synthesis in surrounding code context. The isolated tasks considered here provide a complementary, narrower setting for comparing directly requested and explicitly planned solutions.

Quality ratings and resource measurements. Zheng et al. [14] examine position, verbosity, and self-enhancement biases in LLM-based judging of chat responses. Such findings motivate treating model-assisted ratings as a separate measurement rather than a replacement for functional tests. We analyze a five-criterion code rubric and explicitly identify unavailable assessments and execution-failure penalties. ICE-Score [15] evaluates code using instructed LLM assessments, including comparisons with execution success and human preferences. CodeJudge [12] uses guided analysis to assess semantic correctness without executing test cases. The archived rubric here was not calibrated as either metric; these studies provide context for keeping judge assessments distinct from observed test outcomes. Coignion et al. [3] study the performance of LLM-generated Leetcode solutions and discuss measurement reliability. Resource-efficiency comparisons require a clearly specified workload and timer; the present analysis focuses on correctness, script completion, and rubric ratings.

3 Study design

3.1 Tasks and retained programs

The dataset contains 30 task identifiers, DSA-01 through DSA-30, with one Python program for each combination of task, model, and prompting style. The three models are GPT-4o, Claude 3.5 Sonnet, and Gemini 2.0 Flash; the two styles are zero-shot and CoT, giving $30 \times 3 \times 2 = 180$ retained programs. The task collection includes searching, sorting, arithmetic, string manipulation, array operations, binary search trees, shortest paths, and dynamic programming.

Examples include binary search, merge sort, palindrome checking, maximum-subarray sum, array rotation, Dijkstra's algorithm, and the knapsack problem. These tasks permit direct comparison of common algorithmic behaviors. The prompts are short, and several leave conventions such as normalization, tie-breaking, or return representation unspecified. The external contracts in Section 4 make those choices explicit for the present evaluation.

The comparison unit is the retained program associated with a model–style–task combination. Within each model, the same 30 task identifiers form matched pairs. Multiple test cases and local execution repetitions measure coverage and stability of those programs.

3.2 Prompting styles and model variants

The retained zero-shot examples contain a task request without worked demonstrations or an explicit planning instruction. The CoT-style instruction requests problem analysis, selection and justification of data structures and algorithms, and a solution outline embedded as a multiline code comment. It also requests a standalone function or class with suitable invocation or test inputs. This describes the available prompt examples; complete provider request logs are not retained for every program.

We identify the code-generation models as GPT-4o, Claude 3.5 Sonnet, and Gemini 2.0 Flash. Throughout the results, GPT, Claude, and Gemini denote these respective variants, following our study records; dated API snapshots, decoding settings, and per-request generation timestamps are not retained. The archived evaluator source separately configures `gemini-2.0-flash-lite` as the quality judge. This judge is distinct from the Gemini 2.0 Flash code generator; stored responses do not independently confirm a dated judge endpoint.

3.3 Evaluation datasets

The functional evaluation uses all 180 retained source programs and the retrospective contracts described in Section 4.

The rubric analysis uses a separate set of archived assessments dated 22 May 2025, keyed by model, style, and task identifier. It contains 179 of 180 expected records: 177 completed assessments, one execution-failure zero penalty, and one malformed judge response. GPT–CoT DSA-20 has no assessment; the two Gemini–CoT error records are DSA-28 (execution-failure penalty) and DSA-27 (malformed response). These ratings and the new functional checks are complementary analyses. Historical records do not retain per-request source hashes, so we do not use matched model–style–task keys alone to claim a calibrated relationship between a judge score and the newly measured behavior of an identical historical input file.

4 Evaluation methods

4.1 External functional contracts

Each task has an explicit callable contract and reference checks. The contracts, API adapters, and cases were developed retrospectively after inspecting the retained source. Reference computations are separately implemented or use standard-library functions; they do not reuse generated demonstration assertions. Examples include exhaustive interval sums for maximum subarray, Bellman–Ford relaxation for shortest paths, exhaustive subset optimization for small knapsack instances, and complete substring enumeration for pattern matching. Randomly constructed inputs use the recorded seed `20261103`, with a fixed task-specific offset.

Input domains address ambiguities consistently. Palindrome checks use lowercase ASCII alphanumerics, avoiding additional punctuation and case-normalization requirements. The maximum-subarray contract requires a nonempty interval. String-rotation checks exclude the two-empty-string case. The missing-number primary domain is an internally missing value from $1, \ldots, N$; arbitrary integer starting points are evaluated as separate exploratory probes. The contracts and exclusions define the scope of each pass result.

Adapters reconcile function names and interface conventions: returned versus in-place lists, index versus value pairs, graph layouts, and first-match versus all-match string APIs. Array-rotation direction follows the declared source convention; unspecified variants must use a consistent direction.

No adapter changes an algorithm body, repairs syntax, or fabricates output. Binary-search-tree checks inspect structural properties as well as traversal results.

A program passes its callable suite only if every primary check succeeds. Syntax errors and unavailable callable interfaces remain distinct outcomes and are retained in the program denominator. A fixed-output FizzBuzz task has one full-sequence assertion; other tasks have multiple cases. Assertion counts therefore describe test coverage rather than independent sample size. Positive reference/interface controls and selected deliberately incorrect implementations check whether the suites accept known valid behavior and reject chosen defects. These controls do not establish test completeness.

4.2 Callable and whole-script execution

Both execution modes use the same retained source bytes. Callable mode first parses the complete file, retaining syntax failures. It then loads permitted imports, function/class definitions, docstrings, and literal constants from the abstract syntax tree. Top-level demonstrations, loops, calls, and assertions are omitted so that external checks control the inputs and expected outputs.

Whole-script mode executes the complete unchanged source with its main block, demonstrations, and assertions. Completion means the script finishes without a recorded syntax or runtime failure. The two modes answer different questions: a callable can satisfy an external contract while its complete script fails a generated test or demonstration.

Each program runs in three fresh processes per mode, giving $180 \times 2 \times 3 = 1{,}080$ worker runs. The runner records outcome labels, per-case results, and source/test hashes. It uses isolated Python startup, a credential-free environment, a temporary working directory, restrictions on external I/O, permitted imports, and CPU, memory, and wall-time limits. These controls support consistent local evaluation of the inspected collection; they are not a security assessment of arbitrary generated code.

4.3 Model-assisted quality rubric

The archived rubric scores five criteria from 0 to 20: execution correctness e, output correctness o, input handling h, implementation completeness c, and requirement compliance r. The archived 60/40 split between the three-criterion correctness group and the two-criterion adherence group reduces exactly to an equally weighted arithmetic mean of all five criteria:

$$Q = 0.6\frac{e+o+h}{3} + 0.4\frac{c+r}{2} = \frac{e+o+h+c+r}{5}. \tag{1}$$

Each criterion contributes 20%. Q is out of 20, while $5Q$ is the raw total out of 100. Component ranges and stored arithmetic are validated before aggregation. A high rubric score is a model-assisted assessment, not a measured fraction of independently passed functional tests.

The primary comparison retains the recorded execution-failure zero penalty and excludes unavailable or malformed assessments. Each model's zero-shot and CoT means use exactly the same available task IDs. A conditional sensitivity analysis instead requires two completed judge assessments, excluding the failure penalty as well. A parsing placeholder is never interpreted as an ordinary code-quality rating.

4.4 Paired comparisons and sensitivity

For callable behavior and whole-script completion, we compare task-level pass counts within each model and report discordant task pairs. Every retained program is counted once, using repetition

1; repetitions 2 and 3 check the consistency of its outcome. The finite checks characterize the saved programs under their stated contracts.

For rubric scores, the mean CoT-minus-zero-shot difference on paired tasks P_m is

$$\overline{\Delta}_m = \frac{1}{|P_m|} \sum_{i \in P_m} (Q_{m,i,\mathrm{CoT}} - Q_{m,i,\mathrm{ZS}}). \tag{2}$$

We report score differences in points on the 0–20 scale, alongside the number of tasks with higher, equal, or lower CoT scores. Exact leave-one-task-out ranges show how deleting a task changes the observed difference. They describe sensitivity to task inclusion; they are not confidence intervals. We report descriptive comparisons without significance tests.

Runtime and memory records are not used for between-style efficiency rankings because the retained instrumentation does not supply a common, workload-controlled measure of algorithm-only performance.

5 Results

5.1 RQ1: functional outcomes

Table 1 reports callable-suite outcomes and whole-script completion for the 180 retained implementations. Overall, 178 implementations pass their callable suites (98.89%). GPT and Claude pass all 30 tasks in both styles; Gemini passes 29 of 30 in each style. Each style therefore has 89 callable-suite passes among 90 implementations. The two Gemini failures occur on different tasks, so the equal aggregate rates do not imply identical behavior. The denominator includes the source that fails syntax checking.

Table 1: Callable-suite and whole-script outcomes by model and prompting style.

Model	Style	Callable suite	Whole script
GPT	ZS	30/30	30/30
GPT	CoT	30/30	30/30
Claude	ZS	30/30	30/30
Claude	CoT	30/30	30/30
Gemini	ZS	29/30	30/30
Gemini	CoT	29/30	27/30

A callable pass satisfies every primary retrospective check; full-source syntax errors remain failures. Whole-script completion includes generated demonstrations and assertions. Each artifact has the same outcome in three fresh processes per mode; denominators count artifacts once, not repeated runs or assertions. ZS denotes zero-shot.

Whole-script completion succeeds for 177 of 180 implementations (98.33%). All 90 zero-shot scripts complete, compared with 87 of 90 CoT scripts. The three unsuccessful scripts are Gemini CoT solutions. One contains a syntax error, while the other two fail their generated demonstration tests. Section 5.2 examines these cases separately from the external callable checks.

All outcome labels and per-case pass/fail results are stable across three fresh processes per implementation and evaluation mode. The resulting 1,080 worker runs check repeatability of the saved programs; reported denominators count each implementation once. Positive reference/interface controls pass, and the predefined incorrect controls are rejected.

The matched callable outcomes give a zero CoT-minus-zero-shot pass-rate difference within each model. GPT and Claude have no discordant tasks; Gemini has one CoT-only pass and one zero-shot-only pass. Removing one Gemini task produces differences between approximately -3.45 and $+3.45$ percentage points. This describes sensitivity to task inclusion. The high pass rates also leave limited scope for distinguishing styles on these primary contracts.

5.2 RQ2: failures and specification choices

Table 2 summarizes the observed counterexamples and generated-test failures. The Gemini zero-shot maximum-subarray solution returns zero for $[-1]$, whereas the declared nonempty-subarray contract gives -1. Other all-negative arrays reveal the same convention difference. The brief original prompt does not resolve whether an empty subarray is permitted, so the result identifies a mismatch with the stated evaluation contract rather than an unqualified violation of the original request.

The Gemini CoT missing-number source, DSA-28, contains a leftover Markdown fence and fails full-source parsing at line 75. Neither evaluation mode repairs the source. This failure therefore affects both callable-suite and whole-script outcomes.

Table 2: Counterexamples and generated-test defects.

Task	Artifact	Evidence and interpretation
DSA-06	Gemini ZS	$[-1]$ returns 0; the nonempty reference gives -1. This fails the declared retrospective convention; an empty-subarray convention would differ.
DSA-28	Gemini CoT	A leftover Markdown fence causes a syntax error at line 75. Neither execution mode repairs the source.
DSA-03	Gemini CoT	A malformed generated test entry unpacks `.a` into input `.` and expected value `a`. The assertion fails despite a callable-suite pass.
DSA-17	Gemini CoT	The generated test expects `Leading` (7 letters), although the returned `trailing` has 8. This is a faulty expected answer.

These examples describe the saved artifacts under explicitly documented conditions.

The additional whole-script failures arise in generated test data. In DSA-03, the palindrome test list contains a bare `.a` entry followed by a Boolean instead of a test-case pair. Unpacking that string produces input `.` and expected value `a`, causing the assertion to fail. In DSA-17, the longest-word test expects `Leading`, although the returned `trailing` is longer (8 versus 7 letters). Both callable implementations pass the external primary suites. These remain script-level failures, but their immediate causes are faulty generated tests rather than the corresponding callable computations. The distinction explains why the lower CoT script completion count does not translate directly into a lower callable pass rate.

The exploratory missing-number probes extend the primary $1, \ldots, N$ input contract to sequences beginning at other integers. Each zero-shot solution passes zero of seven probes, while GPT and Claude CoT each pass seven of seven. Gemini CoT has no executable probe outcomes because its source fails parsing. These results reveal different assumptions about the sequence origin. They are reported separately from the primary rates because the original prompt does not specify that broader domain.

5.3 RQ3: judge ratings and sensitivity

Table 3 compares the archived judge ratings on matched tasks: 29 GPT pairs, 30 Claude pairs, and 29 Gemini pairs. On the 0–20 composite scale, the mean CoT-minus-zero-shot differences are -0.124, $+0.180$, and -0.497 points, respectively. GPT has seven higher CoT scores and six lower scores; Claude has 11 higher and seven lower; Gemini has 13 higher and six lower. The remaining pairs are ties. Thus more positive task-level changes can coexist with a negative mean when one failure receives a large penalty.

Table 3: Paired judge ratings and task-deletion sensitivity.

Model	Pairs	Zero-shot	CoT	Δ	LOO range
GPT	29	19.552	19.428	-0.124	$[-0.150, -0.071]$
Claude	30	19.207	19.387	$+0.180$	$[+0.131, +0.262]$
Gemini	29	19.407	18.910	-0.497	$[-0.564, +0.171]$

Scores are out of 20; Δ is CoT minus zero-shot. LOO is the exact range after deleting one paired task, not a confidence interval. The execution-failure zero penalty is retained; missing and malformed assessments are excluded from pairing.

Failure treatment particularly affects Gemini. Requiring two completed judge assessments reduces its comparison to 28 pairs, with means of 19.414 for zero-shot and 19.586 for CoT: a difference of $+0.171$ points.[1] GPT and Claude remain unchanged. Conversely, including both Gemini error rows as zero scores yields a CoT mean of 18.280 across all 30 archived records. The primary paired analysis retains the explicit execution-failure penalty, excludes the malformed judge response, and leaves the absent GPT assessment unfilled. Table 3 also reports exact leave-one-task-out ranges to make the influence of individual tasks visible.

Input handling is the lowest-rated criterion in every completed-assessment condition, with means from 17.33 to 18.36. This pattern identifies a common area for closer testing despite the high composite scores. The ratings and functional checks nevertheless measure different outcomes: the former reflect the archived rubric, while the latter apply explicit external contracts to the saved programs. Their results show mixed style associations and do not support a uniform CoT advantage across these evaluation dimensions. Historical per-request source hashes were not retained, so the archived ratings and the new functional results remain separate evidence series.

6 Discussion

6.1 Comparative interpretation

The zero-shot and CoT conditions achieve the same primary callable-suite pass counts within each model: GPT and Claude each pass 30 of 30 tasks in both styles, while Gemini passes 29 of 30 in each. Within this collection, the CoT condition therefore shows no aggregate increase in callable-suite success. The passing counts also approach the ceiling of the evaluation, leaving little room for this binary measure to distinguish the conditions. They should be interpreted alongside the criterion scores and task-level outcomes.

Equal counts do not imply identical behavior. Gemini's zero-shot failure concerns the declared nonempty-subarray convention, whereas its CoT failure is a source syntax error. These different cases cancel in the aggregate paired pass-rate difference. GPT and Claude have no discordant

[1] Unrounded means give 0.171; subtracting the rounded means gives 0.172.

callable outcomes on the checked tasks. The comparison therefore identifies the behavior of these saved solutions and the role of the declared evaluation contracts.

The judge-based comparison adds a different perspective. Mean paired CoT-minus-zero-shot differences are -0.124 points for GPT, $+0.180$ for Claude, and -0.497 for Gemini on the 0–20 scale when the recorded execution-failure penalty is retained. Requiring two completed assessments changes Gemini's difference to $+0.171$ points. The primary result includes an observed pipeline failure, while the conditional result describes only pairs with completed assessments. Their different directions show why coverage and penalty rules need to accompany a reported quality average. Neither policy supports a uniform improvement from CoT across all three models.

Callable and whole-script outcomes further separate solution behavior from submission behavior. Gemini CoT completes 27 whole scripts, compared with 29 callable-suite passes, because its generated palindrome and longest-word tests contain defects. One malformed test entry and one incorrect expected answer cause failures even though the corresponding functions pass the external primary checks. Conversely, normal script completion alone does not establish that all outputs satisfy an external contract. Retaining both measurements helps distinguish an implementation error, a formatting failure, and a faulty generated test.

For practical evaluation, these results favor matched task specifications, external expected outputs, and separate records for execution and rubric judgments. An explicit solution outline can expose assumptions and provide material for review, but its value should be checked against the required behavior. The present comparison does not measure generation cost or developer productivity, so the observed score differences do not establish a cost or workflow advantage for either prompting style.

6.2 Scope and generalizability

The evaluation covers 180 retained Python solutions across 30 tasks, with one solution per model/style/task combination. Three local repetitions assess execution stability; they do not sample generation variability. Contracts and adapters were developed retrospectively through source inspection, using separately implemented reference calculations. Suite outcomes apply to the specified input domains and checked cases; complexity and security properties require separate assessment.

The tasks are common textbook exercises. Similar problems or solutions may have appeared in model training data, creating a contamination or memorization risk that this collection cannot quantify. High pass rates are compatible with both prior exposure and effective problem solving; they do not distinguish the two. Time-filtered problems such as those in LiveCodeBench [7] and broader library-based tasks such as BigCodeBench [16] would help assess transfer beyond this setting.

The model variants are identified in Section 3.2. Dated API snapshots, decoding settings, and per-request generation timestamps are not retained, so provider-version and collection-time effects cannot be isolated. Judge scores complement the external checks, with no independent human-calibration estimate available for those ratings. The configured Gemini judge also assesses Gemini programs, so same-provider self-enhancement or style preferences could affect those scores [14]. The archive has no independent judge comparison that could estimate the direction or magnitude of such bias.

6.3 Future work

Extensions include broader task collections, repeated generations from recorded endpoints, and independent assessment of judge ratings. A prospective collection can freeze task contracts before generation, balance generation order across styles, and retain full request/response metadata. These additions would extend the comparison to variation across generated programs and more complex software contexts.

7 Conclusion

ChainZCode compares zero-shot and CoT Python solutions using external callable checks, whole-script execution, and model-assisted quality ratings. Across the 180 retained programs, 178 pass the callable suites and 177 complete as scripts. Callable pass counts are equal between styles within each model, while paired judge-score differences are mixed and sensitive to failure treatment. Task-level inspection identifies a contract-specific counterexample, a syntax error, and two faulty generated self-tests that explain the differing outcomes. The findings support reporting execution, functional behavior, and judged quality separately when evaluating generated code.

Code and data availability

The evaluated programs, task prompts, contracts, reference checks, adapters, evaluation worker, and per-case results are available at `https://doi.org/10.5281/zenodo.23894180`.

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. arXiv:2108.07732, 2021. `https://arxiv.org/abs/2108.07732`.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. arXiv:2107.03374, 2021. `https://arxiv.org/abs/2107.03374`.

[3] Tristan Coignion, Clément Quinton, and Romain Rouvoy. A performance study of LLM-generated code on Leetcode. arXiv:2407.21579, 2024. `https://arxiv.org/abs/2407.21579`.

[4] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. arXiv:2308.01861, 2023. `https://arxiv.org/abs/2308.01861`.

[5] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. arXiv:2204.05999, 2022. `https://arxiv.org/abs/2204.05999`.

[6] Dan Hendrycks, Steven Basart, Saurav Kadavath, et al. Measuring coding challenge competence with APPS. arXiv:2105.09938, 2021. `https://arxiv.org/abs/2105.09938`.

[7] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida I. Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *Proceedings of the Thirteenth*

International Conference on Learning Representations (ICLR), 2025. https://arxiv.org/abs/2403.07974.

[8] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):182:1–182:30, 2024. https://doi.org/10.1145/3672456.

[9] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. arXiv:2305.06599, 2023. https://arxiv.org/abs/2305.06599.

[10] Yujia Li, David Choi, Junyoung Chung, et al. Competition-level code generation with AlphaCode. arXiv:2203.07814, 2022. https://arxiv.org/abs/2203.07814.

[11] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. arXiv:2305.01210, 2023. https://arxiv.org/abs/2305.01210.

[12] Weixi Tong and Tianyi Zhang. CodeJudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051. Association for Computational Linguistics, 2024. https://aclanthology.org/2024.emnlp-main.1118/.

[13] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. arXiv:2201.11903, 2022. https://arxiv.org/abs/2201.11903.

[14] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. arXiv:2306.05685, 2023. https://arxiv.org/abs/2306.05685.

[15] Terry Yue Zhuo. ICE-Score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2232–2242. Association for Computational Linguistics, 2024. https://aclanthology.org/2024.findings-eacl.148/.

[16] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*, 2025. https://arxiv.org/abs/2406.15877.