

# Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering

Tal Ridnik, Dedy Kredo, Itamar Friedman  
CodiumAI

{tal.r, dedy.k, itamar.f}@codium.ai

## Abstract

*Code generation problems differ from common natural language problems - they require matching the exact syntax of the target language, identifying happy paths and edge cases, paying attention to numerous small details in the problem spec, and addressing other code-specific issues and requirements. Hence, many of the optimizations and tricks that have been successful in natural language generation may not be effective for code tasks. In this work, we propose a new approach to code generation by LLMs, which we call AlphaCodium - a test-based, multi-stage, code-oriented iterative flow, that improves the performances of LLMs on code problems. We tested AlphaCodium on a challenging code generation dataset called CodeContests, which includes competitive programming problems from platforms such as Codeforces. The proposed flow consistently and significantly improves results. On the validation set, for example, GPT-4 accuracy (pass@5) increased from 19% with a single well-designed direct prompt to 44% with the AlphaCodium flow. Many of the principles and best practices acquired in this work, we believe, are broadly applicable to general code generation tasks.*

Full implementation is available at: <https://github.com/Codium-ai/AlphaCodium>

## 1. Introduction

With a sparse reward signal, code generation tasks require searching in the huge structured space of possible programs. Correct solutions to the same problem can look significantly different, and judging if a partial or incorrect solution is useful is a difficult challenge - a single-character edit can completely alter the solution's behavior. Due to the unique nature of code generation tasks, common prompting techniques that have been optimized for natural language tasks [4, 13, 10], may not be as effective when applied to code generation.

Recent large-scale transformer-based language mod-

els [12] have successfully generated code that solves simple programming tasks [2, 1]. However, real-world code problems are often different in nature - they are more nuanced, and can be defined by a long natural language task description (i.e., spec), that contains multiple details and rules that the solution code must address.

The introduction of CodeContests [8], a dataset curated from competitive programming platforms such as Codeforces [9], enabled the evaluation of models and flows on more challenging code problems, which usually include a lengthy problem description. A private test set, with more than 200 unseen tests per problem, enables to evaluate the generated code comprehensively, and to reduce false positive rates to a minimum.

The primary work addressing the CodeContests dataset was AlphaCode [8], a code generation system developed by DeepMind, that utilizes a fine-tuned network specifically for competitive programming tasks. AlphaCode generates a very large number of possible solutions (up to 1M), that are then processed and clustered, and among them a small number ( $\sim 10$ ) is chosen and submitted. While the results of AlphaCode are impressive, the need to fine-tune a model specifically for code-oriented tasks, and the heavy computational brute-force-like load, makes it impractical for most real-life usages. CodeChain [7] is another work to tackle competitive programming tasks, which introduced a novel inference framework to improve code generation in LLMs through a chain of sub-module-based self-revisions.

In this paper, we present AlphaCodium, a code-oriented flow that revolves around an iterative process where we repeatedly run and fix a generated code against input-output tests. Two key elements for AlphaCodium flow are (a) generating additional data, such as problem reflection and test reasoning, to aid the iterative process, and (b) enrichment of public tests with additional AI-generated tests. The proposed flow, which is depicted in Figure 1, is divided into two main phases: a pre-processing phase where we reason about the problem in natural language, and an iterative code generation phase where we generate, run, and fix a code solution against public and AI-generated tests.

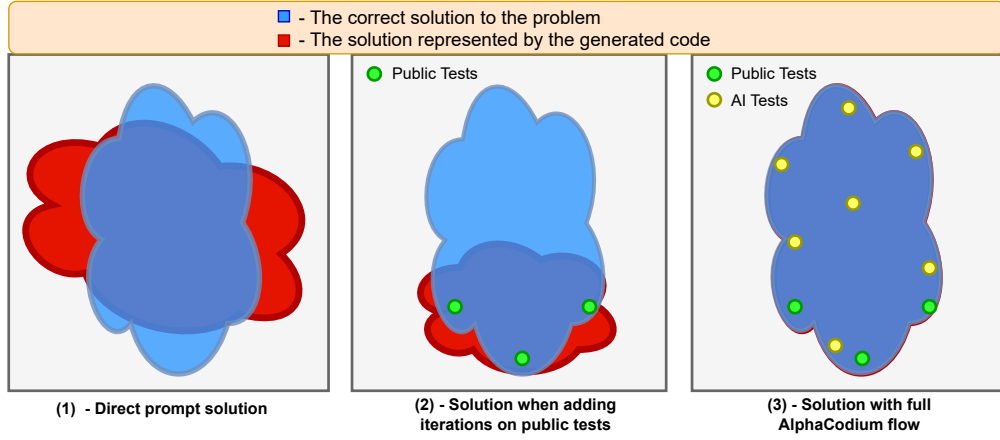
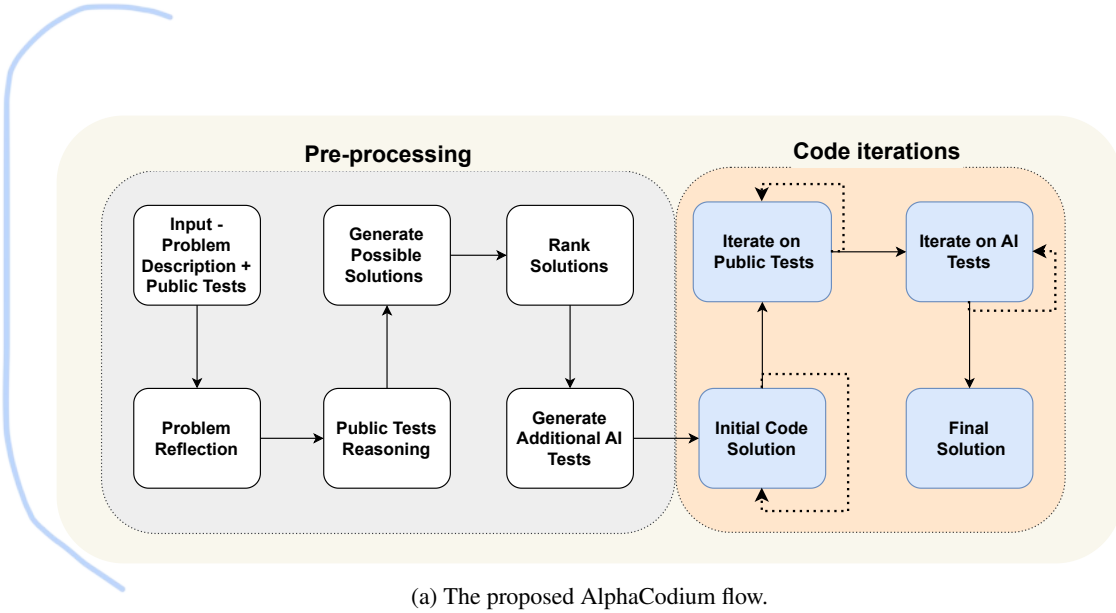


Figure 1: **Illustration of AlphaCodium flow contribution** - with direct prompt, the model struggles to solve code problems. Iterating on public tests stabilizes and improves the solution but leaves "blind spots" because the public tests are not comprehensive. The full AlphaCodium flow, which includes a pre-processing phase as well as iterations on public and AI-generated tests, allows the solution to be further improved, leading to increased solve ratio.

A key observation when designing AlphaCodium flow is that generating additional useful tests is easier than generating a correct code solution. Adding specific tests requires mainly understanding the problem, some insight, and basic brute-force or logical reasoning. There is no need to fully "solve" the problem when generating additional tests.

AlphaCodium flow also utilizes novel code-oriented design concepts, tricks, and best practices, such as: (1) YAML structured output; (2) bullet point analysis to encourage semantic reasoning; (3) generating modular code; (4) soft decisions with double validation; (5) encouraging exploration and postponing direct decisions; (6) test anchors.

AlphaCodium's flow, when compared against a well-

designed single prompt, consistently and significantly improves the performance of LLMs on CodeContests problems. This is true both for open-source models (DeepSeek [3]) and closed-source models (GPT [5]). For GPT-4 on the validation set, for example, the pass@5 accuracy improved from 19% to 44%. AlphaCodium also outperforms previous works, while having a significantly smaller computational budget - it achieves superior results over AlphaCode, for example, with four orders of magnitude fewer LLM calls.

We believe many of the principles and best practices used in this work broadly apply to general code generation tasks. In addition, we argue that utilizing harder and more

complicated benchmarks like CodeContests dataset will allow the community to better evaluate LLMs, compared to more simplistic benchmarks, which are common today, like HumanEval [2].

## 2. CodeContests Dataset

CodeContests [8] is a challenging code generation dataset introduced by Google’s DeepMind, involving problems curated from competitive programming platforms such as Codeforces [9]. The dataset contains 10K code problems that can be used to train LLMs, as well as a validation and test set to assess the ability of LLMs to solve challenging code problems.

In this work, instead of training a dedicated model, we focused on developing a code-oriented flow, that can be applied to any LLM pre-trained to support coding tasks, such as GPT [5] or DeepSeek [3]. Hence, we chose to ignore the train set, and focused on the validation and test sets of CodeContests, which contain 107 and 165 problems, respectively. Figure 2(a) depicts an example of a typical problem from CodeContests dataset. Each problem consists of a description and public tests, available as inputs to the model. The goal is to generate a code solution that produces the correct output for any (legal) input. A private test set, which is not available to the model or testers, is used to evaluate the submitted code solutions.

### What makes CodeContests a good dataset for evaluating LLMs on code generation tasks:

1) CodeContests, unlike many other competitive programming datasets [6, 2], utilizes a comprehensive private set of tests to avoid false positives - each problem contains  $\sim 200$  private input-output tests the generated code solution must pass.

2) LLMs generally do not excel at paying attention to small details, because they typically transform the problem description to some “average” description, similar to common cases on which they were trained. Real-world code problems, on the other hand, frequently contain minor details that are critical to their proper resolution. A key feature of the CodeContests dataset is that the problem descriptions are, by design, complicated and lengthy, with small details and nuances (see a typical description in Figure 2(a)). We feel that adding this degree of freedom of problem understanding is beneficial since it simulates real-life problems, which are often complicated and involve multiple factors and considerations. This is in contrast to more common code datasets such as HumanEval [2], where the problems are easier, and presented in a concise manner. An example of a typical HumanEval problem appears in appendix A.

Figure 2(b) depicts the model’s introspection on the problem presented in Figure 2(a). Note that proper self-reflection makes the problem clearer and more coherent.

This illustrates the importance of problem understanding as part of a flow that can lead, with high probability, to generating a correct code solution.

## 3. The Proposed Flow

### 3.1. Overview

Due to the complicated nature of code generation problems, we observed that single-prompt optimizations, or even chain-of-thought prompts, have not led to meaningful improvement in the accuracy of LLMs on CodeContests. The model struggles to understand and comprehend the problem, and continuously produces wrong code, or a code that passed public tests but fails to generalize to unseen private tests. Common flows, that are suitable for natural language tasks, may not be optimal for code-generation tasks, which include an untapped potential - repeatedly **running** the generated code, and validating it against known examples.

Instead of common prompt engineering techniques used in NLP, we found that to solve CodeContest problems it was more beneficial to employ a dedicated code-generation and testing-oriented flow, that revolves around an iterative process where we repeatedly run and fix the generated code against input-output tests (see Figure 2(a) for examples of such tests). Two key elements for this code-oriented flow are (a) generating additional data in a pre-processing stage, such as self-reflection and public tests reasoning, to aid the iterative process, and (b) enrichment of the public tests with additional AI-generated tests.

In Figure 1(a) we present AlphaCodium flow for solving competitive programming problems. The flow is divided into two main phases:

- The *pre-processing* phase represents a linear flow where AlphaCodium reasons about the problem, in natural language,
- The *code iterations* phase includes iterative stages where AlphaCodium generates, runs, and fixes a solution code against certain tests.

### 3.2. Flow stages

In this section, we will review the different stages used in AlphaCodium flow (Figure 1(a)):

---

**Problem reflection** Describe the problem, in bullet points, while addressing the problem goal, inputs, outputs, rules, constraints, and other relevant details that appear in the problem description.

---

Problem Description	Public Tests	Private Tests
<p>Mr. Chanek's city can be represented as a plane. He wants to build a housing complex in the city. There are some telephone poles on the plane, which is represented by a grid a of size <math>(n + 1) \times (m + 1)</math>. There is a telephone pole at <math>(x, y)</math> if <math>a_{x,y} = 1</math>. For each point <math>(x, y)</math>, define <math>S(x, y)</math> as the square of the Euclidean distance between the nearest pole and <math>(x, y)</math>. Formally, the square of the Euclidean distance between two points <math>(x_1, y_1)</math> and <math>(x_2, y_2)</math> is <math>(x_2 - x_1)^2 + (y_2 - y_1)^2</math>. To optimize the building plan, the project supervisor asks you the sum of all <math>S(x, y)</math> for each <math>0 \leq x \leq n</math> and <math>0 \leq y \leq m</math>. Help him by finding the value of <math>\sum_{y=0}^m \sum_{x=0}^n S(x, y)</math>.</p> <p>...</p>	<p><b>Input:</b>  2 2  101  000  000  <b>Output:</b>  18  ...</p>	<p><b>Input:</b>  5 4  10010  00000  01000  00001  00100  00010  <b>Output:</b>  36  ...</p>

(a) A typical CodeContests problem.

Self-reflection on the problem
<ul style="list-style-type: none"> <li>- The problem is about a city represented as a plane with a grid of size <math>(n + 1) \times (m + 1)</math> where some points have telephone poles.</li> <li>- The task is to calculate the sum of the squares of the Euclidean distances from each point in the grid to its nearest telephone pole.</li> <li>- The grid is represented by a matrix where 1 indicates the presence of a telephone pole and 0 indicates its absence.</li> <li>- The input consists of the size of the grid <math>(n, m)</math> and the grid itself.</li> <li>- The output is a single integer, the sum of the squares of the Euclidean distances from each point to its nearest telephone pole.</li> </ul>

(b) An AI-generated self-reflection on the problem.

Figure 2: **Problem description and reflection** - an example of a typical CodeContests problem, with AI-generated self-reflection on the problem. While the original description is lengthy and complicated, proper self-reflection makes the problem clearer and more coherent, leading to improved code solutions.

**Public tests reasoning** Explain why each test input leads to the output.

**Generate possible solutions** Generate a list of 2-3 possible solutions to the problem, described in natural language.

**Rank solutions** Rank the possible solutions and choose the “best solution”, in terms of correctness, simplicity, and robustness. (not necessarily take the “most efficient” solution).

**Generate additional AI tests** Generate an additional 6-8 diverse input-output tests for the problem. Try to cover cases and aspects not covered by the original public tests.

**Initial code solution** The goal of this stage is to generate an initial code solution to the problem. It is essential that this code will reasonably “close” to the correct code, so the

run-fix iterations in the next stages will have a better chance of succeeding. The stage flow:

- Choose a potential solution. Generate a corresponding code, and run it on selected public and AI tests.
- Repeat this process until the tests pass, or until a try-limit is reached.
- The first code that passes the tests, or the code with the closest output (see appendix D), will be used as the base code for the next steps.

**Iterate on public tests** Start from the base code. Iteratively run it on the public tests. If the code fails on a specific test, try to fix it, given the error message.

**Iterate on AI-generated Tests** Continue the run-fix iterations on the AI-generated tests. Use “test anchors” (see section 4).

---

### 3.3. Additional insights

In this section, we will offer additional insights and share intuitions about the proposed flow.

Firstly, the flow relies on knowledge accumulation - trying to progress from easy to hard, gaining knowledge and insight along the way to help with the more difficult stages. For example, the output of the first step, *problem reflection*, can be utilized as prompt input to more difficult steps like *generate possible solutions*. The pre-processing phase's outputs are used to aid the most challenging and critical phase, code iterations, where we try to generate code that correctly solves the problem.

Another key observation in designing AlphaCodium is that for AI, generating more tests is easier than generating a full solution code. Generating additional tests requires mainly understanding the problem and basic brute-force or logical reasoning. There is no need to fully "solve" the problem in order to generate additional useful input-output test pairs. This is in contrast to generating a correct solution code, which requires a complete algorithmic solution, equivalent to correctly solving any possible pair of input-output tests. As a result, we can generate more AI tests, and then leverage them to improve the code creation phase, as described in Figure 1(b). We further amplify the contribution of these additional tests by asking the model to focus on aspects not addressed by the original public tests, such as large inputs, edge cases, and so on.

Also note that some steps can be combined into a single LLM call, and the flow in Figure 2(a) is a conceptual flow, emphasizing the process's high-level steps. In practice, structured output (see section 4) enables to combine multiple stages into a single LLM call, in order to save resources, or because a model performs better when doing specific tasks concurrently.

## 4. Code-Oriented Design Concepts

In this section we will present additional design concepts, tricks, and best practices we found beneficial when trying to solve code generation problems. AlphaCodium flow proposed in Figure 1 extensively uses these design concepts.

**YAML Structured output:** the usage of structured output - asking the model to generate an output in YAML format, equivalent to a given Pydantic class - is a key component in our proposed flow. An example of such instruction (*possible solutions* stage) appears in Figure 3.

Structured output eliminates the majority of the hassle and dark knowledge required for "prompt engineering"

and instead allows complicated tasks to be presented in a straightforward, code-like manner. It also makes it possible to obtain complex answers that involve several stages, representing a logical and methodical thinking process.

While newer versions of GPT models [5] have built-in support for JSON-style output, we argue that YAML output is far more suitable specifically for code generation tasks, see appendix B.

**Semantic reasoning via bullet points analysis:** when asking an LLM to reason about an issue, better results are obtained when demanding the output to be in bullet points format. Bullet points encourage an in-depth understanding of the problem, and force the model to divide the output into logical semantic sections, leading to improved results. For example, with self-reflection on a problem in bullet points (Figure 2(b)), each bullet point represents a semantic understanding of a different part of the problem - general description, goals and rules, input structure, and output structure.

**LLMs do better when generating a modular code:** when LLMs are asked to generate a single lengthy function, we observed poor results - the code often contains bugs or logical mistakes. Even worse, a single monolithic code hurts the ability to perform iterative fixing - the model struggles to pinpoint and fix problems, even when given the error message. When clearly asking the model to: "*divide the generated code into small sub-functions, with meaningful names and functionality*", we observe a better-produced code, with fewer bugs, and higher success rates for the iterative fixing stages.

**Soft decisions with double validation:** LLMs tend to struggle with code tasks that require them to think, reason, and make strict, non-trivial decisions. Let's take for example the task of generating additional tests for a problem. Quite often, some of the tests the model generates will be plain wrong. With a *double validation* process, we add an extra step where, given the generated output, the model is asked to re-generate the same output, but correct it if needed. For example, given the generated AI tests as input, the model is asked to re-generate the same tests, while correcting wrong output, if exists. We found that this step of double validation, while encouraging the model to be critical and to reason, is more effective than asking a direct yes/no question: "is this test correct?"

**Postpone decisions, try to avoid direct questions, and leave room for exploration:** when we ask the model direct questions regarding complicated issues, we consistently see hallucinations and wrong answers. To counter this, we adopt a flow of gradual data accumulation, from easier tasks to harder ones:



```

...
Your goal is to present possible solutions to the problem.
Make sure that each solution fully addresses the problem goals, rules, and
constraints.

The output must be a YAML object equivalent to type $PossibleSolutions, according to
the following Pydantic definitions:

class Solution(BaseModel):
    name: str = Field(description="The name of the solution")
    content: str = Field(description="A description of the solution")
    why_it_works: str = Field(description="Why this solution is correct. Be specific\
and detailed regarding the problem rules and goals")
    complexity: str = Field(description="The complexity of the solution")

class PossibleSolutions(BaseModel):
    possible_solutions: List[Solution] = Field(max_items=3, description="A list of\
possible solutions to the problem. Make sure each solution fully addresses the\
problem rules and goals, and has a reasonable runtime - less than three seconds\
on a modern computer, given the problem constraints for large inputs.")

```

Figure 3: Example for a prompt with structured output (*generate possible solutions* stage)

- Start with the easiest tasks - self-reflection on the problem, and reasoning about public tests.
- Move to generating additional AI tests, and possible solutions to the problem
- Only after we acquire the model’s answers to the tasks above, we move to actual code generation, and run-fix iterations.

As another example, instead of choosing a single algorithmic solution to the problem, we prefer to rank several possible solutions, and give priority, but not exclusiveness, to the top-ranked solution when generating initial code. Since the model can be wrong, it’s better to avoid irreversible decisions, and leave room for exploration and code iterations with different possible solutions.

**Test anchors:** even with double validation, some AI-generated tests will be wrong. This makes iterations on them challenging - when a test fails, how can we know if it is because the code is wrong, or because the test is wrong? When we ask the model directly “who is wrong”, we often see hallucinations, and may end up with a wrongly fixed code. To tackle this problem, we utilized a technique of *test anchors*:

- Iterate first on the public tests, which we know are correct. When finished, set all the passed tests as anchor

tests.

- Now iterate on the AI-generated tests, one by one. If a test passes, add it to the list of test anchors
- If a test fails, assume it’s because the code is incorrect, and try to fix the code. However, demand that the fixed code will also pass all the test anchors already acquired. As a result, the test anchors will protect us against an incorrectly fixed code.

Another optimization for test anchors is to sort the AI-generated tests from easy to hard. That way, there are more chances that the iterative process will acquire anchors at the beginning of the process, which can be used as protection later when iterating on the more complicated AI tests.

**What did not work for us:** In appendix C we present additional tricks and methods we tried, which have not led to improved results.

## 5. Results

### 5.1. Direct prompt vs. AlphaCodium flow

In Table 1 we compare AlphaCodium results to the results obtained with a single well-designed direct prompt. The metric being used is pass@k, defined as the percentage of problems solved by using k generated solutions per

model	set	method	score (pass@5)
DeepSeek -33B [3]	validation	Direct	7%
		AlphaCodium	<b>20%</b>
	test	Direct prompt	12%
		AlphaCodium	<b>24%</b>
GPT-3.5	validation	Direct prompt	15%
		AlphaCodium	<b>25%</b>
	test	Direct prompt	8%
		AlphaCodium	<b>17%</b>
GPT-4	validation	Direct prompt	19%
		AlphaCodium	<b>44%</b>
	test	Direct prompt	12%
		AlphaCodium	<b>29%</b>

Table 1: **Comparison of AlphaCodium flow results to direct prompt on various models.**

problem. As can be seen, AlphaCodium flow consistently and significantly improves the performance of LLMs on CodeContests problems. This is true both for open-source (DeepSeek) and close-source (GPT) models, and for both the validation and test sets. For GPT-4 on the validation set, for example, the pass@5 score improves from 19% to 44% - x2.3 improvement.

## 5.2. Comparison to previous works

In Table 2 we compare AlphaCodium results to other methods from the literature. As can be seen, when comparing AlphaCodium to CodeChain with the same model (GPT-3.5) and the same metric (pass@5), AlphaCodium consistently does better.

When comparing AlphaCodium to AlphaCode work, we need to take into account that AlphaCode uses a different generation methodology - fine-tuning an (unknown) model specifically for code problems, generating a very large number of code solutions, clustering them, and submitting K solutions from the top clusters. pass@10@100K, for example, means the 100K (!) solutions were generated and clustered, and 10 solutions were finally chosen and submitted. AlphaCode used a fine-tuned model, and utilized a brute-force-like approach with a significantly higher number of LLM calls. Yet, the top results achieved by AlphaCodium are better

Note that neither AlphaCode nor CodeChain papers [8, 7] released a reproducible open-source solution for CodeContests, including end-to-end generation and evaluation scripts. There are subtleties when evaluating results. For ex-

model	set	method	score
GPT-3.5	validation	AlphaCodium (pass@5)	<b>25%</b>
		CodeChain (pass@5)	17%
	test	AlphaCodium (pass@5)	<b>17%</b>
		CodeChain (pass@5)	14%
GPT-4	validation	AlphaCodium (pass@5)	<b>44%</b>
AlphaCode		AlphaCode (pass@10@1K)	17%
		AlphaCode (pass@10@100K)	24%
GPT-4	test	AlphaCodium (pass@5)	<b>29%</b>
AlphaCode		AlphaCode (pass@10@1K)	16%
		AlphaCode (pass@10@100K)	28%

Table 2: **Comparison of AlphaCodium to other works from the literature.**

ample - how to treat problems with multiple solutions, how to address tolerance issues, timeouts, and more. We compare to the numbers reported in the papers, but release a full reproducible code and evaluation script of AlphaCodium, to enable future comparisons to be more reliable and consistent.

## 5.3. Computational effort and comparison to AlphaCode and AlphaCode2

With AlphaCodium flow we perform ~15-20 LLM calls per solution, so a pass@5 submission involves ~ 100 LLM calls.

AlphaCode did not report how many LLM calls were done per run [8]. Let’s assume one call per run was done (unknown, could be more), then a pass@10@100K (i.e. ten submissions, curated from 100,000 generated solutions) involves 1M LLM calls, four orders of magnitude more than AlphaCodium. Yet, the top results obtained by AlphaCodium are better.

Recently, a new work called AlphaCode2 [11] was published, where a Gemini-Pro model was fine-tuned and evaluated on code programming problems. The paper also reported results on a CodeContests benchmark, but on an up-

dated variant that they did not release to the public. According to AlphaCode2 report: “AlphaCode2 requires about 100 samples to reach the level of performance of AlphaCode with a million samples, making it over 10000× more sample efficient.” Hence both AlphaCode2 and AlphaCodium are four orders of magnitude more efficient than AlphaCode, LLMs call-wise. But, AlphaCode2 utilized a modern foundation model (Gemini-Pro) that was heavily fine-tuned specifically for CodeContests competition, while AlphaCodium uses general-purpose models as-is, and improves their performances without extra data and an expensive training phase.

## 6. Conclusions

In this paper, we introduced AlphaCodium, a code-oriented flow that iteratively runs and fixes a generated code against input-output tests. The flow is divided into two main phases: a pre-processing phase, where AlphaCodium reasons about the problem in natural language, and a code iterations phase, in which AlphaCodium iterates on public and AI-generated tests.

AlphaCodium also utilizes additional design concepts, tricks, and best practices we found beneficial for code generation: structured output in YAML format, generating modular code, semantic reasoning via bullet point analysis, soft decisions with double validation, encouraging exploration, and test anchors.

We tested AlphaCodium on a challenging code generation dataset called CodeContests. The proposed flow consistently and significantly improves results of various closed-source and open-source models. AlphaCodium also outperforms previous works from the literature, while having a significantly smaller computational budget.

## References

- [1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. [1](#)
- [2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. [1](#), [3](#)
- [3] DeepSeek. Deepseek coder: Let the code write itself, 2023. [2](#), [3](#), [7](#)
- [4] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*, 2023. [1](#)
- [5] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020. [2](#), [3](#), [5](#)
- [6] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021. [3](#)
- [7] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*, 2023. [1](#), [7](#)
- [8] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. [1](#), [3](#), [7](#)
- [9] Mike Mirzayanov, Oksana Pavlova, Pavel MAVRIN, Roman Melnikov, Andrew Plotnikov, Vladimir Parfenov, and Andrew Stankevich. Codeforces as an educational platform for learning programming in digitalization. *Olympiads in Informatics*, 14(133-142):14, 2020. [1](#), [3](#)
- [10] Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolo Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, et al. Can generalist foundation models outcompete special-purpose tuning? case study in medicine. *arXiv preprint arXiv:2311.16452*, 2023. [1](#)
- [11] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023. [7](#)
- [12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. [1](#)
- [13] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022. [1](#)



# Appendices

## A. Typical HumanEval code problem

```
/*
Check if in given vector of numbers, are any two numbers closer to each other than
given threshold. >>>
has_close_elements({1.0, 2.0, 3.0}, 0.5) false >>>
has_close_elements({1.0, 2.8, 3.0, 4.0, 5.0, 2.0}, 0.3) true
*/
#include<stdio.h>
#include<vector>
#include<math.h>
using namespace std;
bool has_close_elements(vector<float> numbers, float threshold){
```

## B. Why YAML output is better suited for code generation tasks than JSON output

While newer GPT versions<sup>1</sup> have inherent support for JSON-style output, we argue that YAML output is far better for code generation. Why - generated code often contains single-quote, double-quote, special characters, and so on. LLMs will struggle to *validly* place these characters inside a JSON format, since a JSON output needs to be surrounded with initial double quotes (see Figure 4 (a)). In contrast, YAML output with block scalar<sup>2</sup> must only obey indentation. Any text or code with proper indentation will be a legal one (see Figure 4 (b)).

In addition, as can be seen in Figure 4, since YAML format doesn't need curly brackets, quotations or escape characters, its output has fewer tokens than JSON, hence reducing cost and inference time, and resulting in increased quality as the model needs to pay attention to fewer tokens that are not essential.

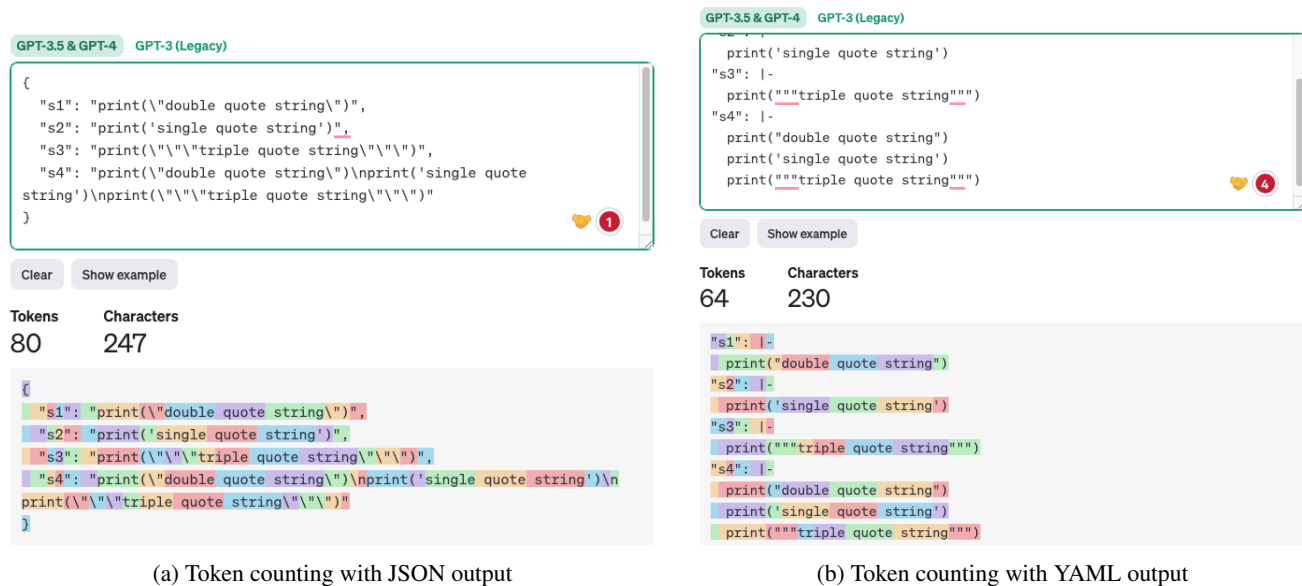


Figure 4: Comparison of the same output, once in JSON format, and once in YAML format. Taken from OpenAI [playground](#).

<sup>1</sup><https://platform.openai.com/docs/guides/text-generation/json-mode>

<sup>2</sup><https://yaml-multiline.info/>

## C. What didn't work for us

1. **Injecting the failed execution trace to the prompt:** In addition to giving the model the error message when doing iterative fixing, we also tried giving it the trace of the last X (50) lines executed. We have not observed improvement from this.
2. **Injecting the last K failed code solutions to the prompt:** When doing iterative fixing, we tried injecting the last K failed code solutions to the prompt, in order to steer the model in different directions. We have not observed improvement from this.
3. **Injecting the last git patch diff to the prompt:** When doing iterative fixing, we also tried to give the last applied code patch diff to the prompt. No improvement was seen.
4. **Complicated single-stage prompts:** we have not observed any significant improvement in results when trying to manipulate and optimize a single-stage prompt, or a chain of non-iterative prompts. The model still struggles to understand the lengthy problem description, tends to ignore specific details, and consistently produces wrong code.

## D. Estimating distance between tests' outputs

When we run a solution code against an input test, the code generates an output. We compare this output to the expected output, and end up with a boolean answer: *pass* or *failed*. However, it is also beneficial to try and **estimate** in some sense the distance between the generated output and the correct output. For this matter, we employed the following logic:

- If the test output is a number, calculate the L2 distance.
- If the test output is an array of numbers, calculate the sum of L2 distances between corresponding array cells.
- If the test output is an array of strings, compare each cell separately (boolean comparison), and return the number of non-identical cells.

This methodology enables us to produce a distance between each generated output and the correct output on CodeCon-tests.