Whiteboard then Code: Team-Refined Code Generation via Iterative Automatic Feedback from Peer LLMs

Nicholas Asker
nga2120@columbia.edu

Abstract

Although state-of-the-art large language models (LLMs) match the performance of human programmers on some coding tasks, they still struggle on difficult programming problems. To improve model performance on coding challenges, this work explores initially creating a content plan – Whiteboarding – before generating code, on the hypothesis that encouraging the generator to think deeply about a high-level solution could help overcome flawed reasoning. Further, it looks at how performance changes when the generator is allowed to iteratively improve the quality of its whiteboarded content plans through Self-Refine. These results are then compared to Team-Refine, a novel extension of Self-Refine intended to simulate human collaboration. In particular, it proposes that distinct peer LLMs, or teammates, provide the feedback instead of the generator itself during the Whiteboard phase, with each teammate bringing a unique background or area of expertise (e.g., efficiency, readability, correctness) to the table. In light of the suggestions, the generator is asked to adjust its plan and then implement the solution in code, using the finalized plan as a blueprint. This study focuses on the effects of collaboration applied during planning (the Whiteboard phase), leaving study of how collaboration fares at other points in the process, such as at code-implementation time (the Code phase), to future work.

1 Introduction

While the performance of top LLMs such as GPT-4 on many coding tasks is competitive with human programmers (Bubeck et al., 2023), they still struggle on difficult programming tasks (OpenAI, 2023), including most of the problems categorized as hard on LeetCode. To improve model performance on coding challenges, this work tests the idea of initially creating a content plan – Whiteboarding – before generating code. The hypothesis is that, by having the generator first output a detailed plan of attack through chain-of-thought prompting, it might be possible to tap into recent gains outside the coding domain, specifically on complex reasoning and problem-solving tasks, to crack open previously unsolvable coding problems. In particular, before doing any coding, encouraging the generator to think deeply about a high-level solution could help overcome flawed reasoning, plausibly one of most limiting factors on challenging programming problems.

Next, experiments test how the generator performs after it is first allowed to iteratively improve the quality of its whiteboarded content plan through Self-Refine (Madaan et al., 2023b), to see whether adjustments – changes to the algorithm, data structures, or the ordering of the steps – can similarly help overcome incorrect initial thinking.

Finally, the Self-Refine approach is extended to peer LLMs, or teammates, each from distinct backgrounds and/or specialized to focus on a particular aspect of generated plans. The idea that this simulated collaboration might lead to performance gains is based on the age-old notion that diverse teams can do better than individuals — especially when composed of specialists from different problem-relevant domains. Each specialized teammate in turn provides specific feedback on the generator's initial content-plan, with their area of expertise determining their perspective (e.g., efficiency, readability, correctness, coherence). In light of the suggestions, the generator outputs another plan, and this process continues iteratively until some condition is met (a set number of iterations, or until a plurality of teammates are satisfied with the current generation). Finally, the generator is asked to implement the solution in code, using the finalized plan as its roadmap.

This study exclusively focuses on the effects of collaboration applied during planning (the Whiteboard phase), leaving study of how collaboration fares at other points in the process, such as at code-

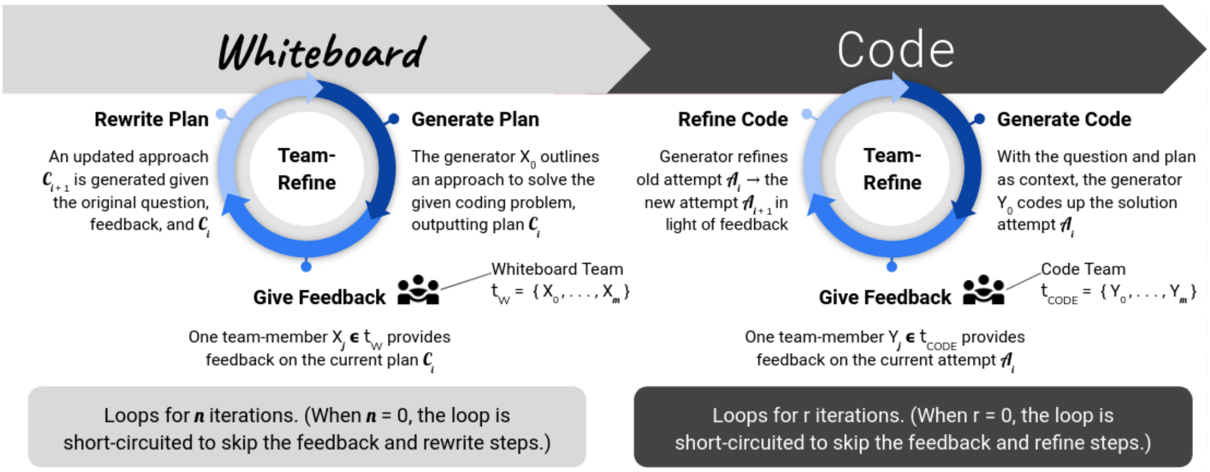


Figure 1: An overview of the Whiteboard-then-Code framework with Team-Refine. First, the Whiteboard phase results in a technical approach, written in words rather than code, to solving the problem. Then, the Code phase outputs a coded-up solution attempt. Team composition and number of iterations can be configured for each of the two Team-Refine cycles. The configurations across phases can differ – for example, the team-members constituting the teams need not be the same in both phases.

implementation time (the Code phase), to future work.

My contributions are as follows:

1. The novel concept of Whiteboarding: inference-time content-planning for code-generation tasks.

2. A framework for (team-)refining in the Whiteboard phase: leveraging automatic feedback in various forms (both from the generator itself and from a diverse set of peer LLMs) to improve plans.

3. Negative findings in the limited setting tested, in particular, that neither (1) Whiteboarding nor (2) Team-Refine in the Whiteboard phase led to performance gains over the baseline, for the particular (relatively small) model used as the generator, on the particular dataset used in this work. While confirming general trends in the literature that smaller models tend to under-perform (especially on automatic evaluation and reasoning tasks), it leaves open the possibility that different manifestations of the framework – using different (larger) models, team specifications, prompts, etc. – may still succeed in other code-generation settings (e.g., software development applications).

The GitHub repository for this project[1] is publicly available and includes the scripts and auxiliary files used for generation and evaluation, generator outputs, and experimental results.

2 Related Work

Using large pre-trained language models to generate code given natural language input has been a task of interest over the last several years, with exemplars such as GPT-3 (Brown et al., 2020), Codex (Chen et al., 2021), and GPT-4 (OpenAI, 2023; Bubeck et al., 2023). The newest open models (Rozière et al., 2023) have proven quite capable in this domain, as well.

This work borrows high-level ideas from content-planning (Su et al., 2021; Tian and Peng, 2022) and lessons from the domain of logical reasoning including chain-of-thought (CoT) prompting (Wei et al., 2023; Kojima et al., 2023) to elicit complex reasoning and better solutions from the generator. Furthermore, it takes inspiration from the idea of process-based, as opposed to outcome-based, feedback (Lightman et al., 2023), so that teammates deliver more targeted and actionable suggestions.

In addition, this project leverages ideas from recent research in automatic feedback and self-improvement. It extends the Self-Refine framework (Madaan et al., 2023b), which provides a task-agnostic method to improving the quality of generations through iterative automatic feedback. Importantly, this is achieved at inference-time, without making parameter updates or relying on reinforcement learning (RL). Adjacent work has similarly avoided relying on expensive fine-tuning or RL,

[1] https://github.com/N-G-Asker/whiteboard-then-code

instead using a form of iteration to achieve gains (Shinn et al., 2023), and there has been much research concerning automatic evaluation techniques (Fu et al., 2023) and other forms of LLM interaction (Perez et al., 2022).

To simulate human collaboration in particular, this work builds on work simulating human evaluators with peer LLMs (Dubois et al., 2023).

3 Data

APPS is a benchmark dataset of coding problems written in natural language, each paired together with a set of Python test cases (Hendrycks et al., 2021).[2] The degree of difficulty ranges widely, including basic and interview-like questions as well as problems sourced from college competitions, and many have proven difficult for state-of-the-art models.

There are $5,000$ questions in both the test and training sets, with an average length of 293.2 words. The number of test cases for each questions varies, but on average there are 21.2 test cases per question in the test split.[3] Figure 2 depicts an interview-level problem from the training split and demonstrates that questions can be quite long, complex, and involve extensive mathematical notation.

This work focuses on the lowest difficulty level – "introductory" questions – but even these can be surprisingly challenging, with reference solutions involving non-trivial data structures and algorithms. In particular, only the first 650 questions in the test split rated at this difficulty level are used for evaluation, and introductory problems in the *training* split are treated as a development split, reserved for cross-validation.

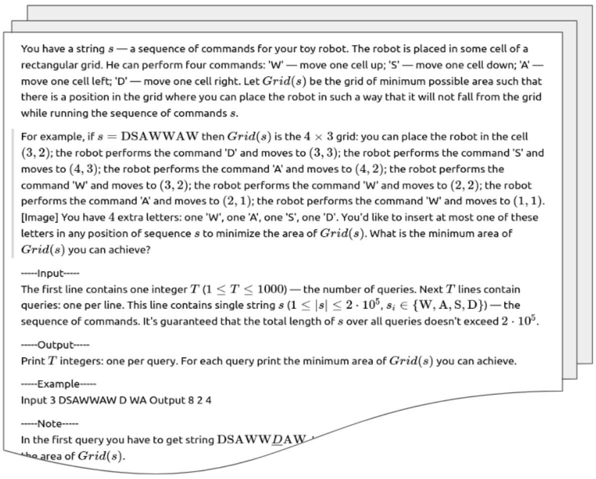
4 Methods

For the **generator** model X_0, *Code Llama Instruct 7B* (Rozière et al., 2023) was used, specifically an 8-bit quantized checkpoint.[4] Code Llama is the state-of-the-art among open models fine-tuned for general-purpose programming, as of August 2023. The choice to use the smallest version (seven billion parameters "7B") along with quanitization was made due to resource constraints. In addition,

[2]The data is available in a convenient form here: https://huggingface.co/datasets/codeparrot/apps. For the full download, go here: https://github.com/hendrycks/apps

[3]I confirmed these statistics, reported here: https://huggingface.co/datasets/codeparrot/apps

[4]https://huggingface.co/TheBloke/CodeLlama-7B-Instruct-GPTQ



You have a string s — a sequence of commands for your toy robot. The robot is placed in some cell of a rectangular grid. He can perform four commands: 'W' — move one cell up; 'S' — move one cell down; 'A' — move one cell left; 'D' — move one cell right. Let $Grid(s)$ be the grid of minimum possible area such that there is a position in the grid where you can place the robot in such a way that it will not fall from the grid while running the sequence of commands s.

For example, if $s = \mathrm{DSAWWAW}$ then $Grid(s)$ is the 4×3 grid: you can place the robot in the cell $(3, 2)$; the robot performs the command 'D' and moves to $(3, 3)$; the robot performs the command 'S' and moves to $(4, 3)$; the robot performs the command 'A' and moves to $(4, 2)$; the robot performs the command 'W' and moves to $(3, 2)$; the robot performs the command 'W' and moves to $(2, 2)$; the robot performs the command 'A' and moves to $(2, 1)$; the robot performs the command 'W' and moves to $(1, 1)$. [Image] You have 4 extra letters: one 'W', one 'A', one 'S', one 'D'. You'd like to insert at most one of these letters in any position of sequence s to minimize the area of $Grid(s)$. What is the minimum area of $Grid(s)$ you can achieve?

-----Input-----

The first line contains one integer T ($1 \leq T \leq 1000$) — the number of queries. Next T lines contain queries: one per line. This line contains single string s ($1 \leq |s| \leq 2 \cdot 10^5$, $s_i \in \{\mathrm{W}, \mathrm{A}, \mathrm{S}, \mathrm{D}\}$) — the sequence of commands. It's guaranteed that the total length of s over all queries doesn't exceed $2 \cdot 10^5$.

-----Output-----

Print T integers: one per query. For each query print the minimum area of $Grid(s)$ you can achieve.

-----Example-----

Input 3 DSAWWAW D WA Output 8 2 4

-----Note-----

In the first query you have to get string $\mathrm{DSAWW}\underline{D}\mathrm{AW}$

the area of $Grid(s)$.

Figure 2: Example interview-level question from the APPS training split. Note that questions can be quite long and involve extensive mathematical notation.

I picked the model that was fine-tuned to receive instructions, since content-planning and collaboration require more than the simple code completion offered by the non-instruct models.

For all questions, the generator first produces a high-level roadmap (even more abstract than pseudocode) before generating the solution in code. Figure 3 outlines the process.

The Self-Refine framework is also applied at the content-planning stage, to see if self-revision and correction at the level of the plan can lead to performance improvements in the code the generator ultimately produces. It is then extended into a **Team-Refine** framework for emulating human collaboration. Figure 1 provides an overview, with the details below.

Denote the m peer LLMs comprising these teams as **teammates** $X_1, \ldots, X_m$. (Write X_0 for the generator itself.) Teammates X_i are also instantiated from the Code Llama 7B Instruct checkpoint. Despite using the same model class M, teammates with distinct "personas" are created to simulate a diverse set of specialists *in the room*, by using different prompts. This choice builds on Dubois et al. (2023) simulating evaluation annotators. If the model class M_i used for each X_i were distinct from one another and from the generator (that is, for all $0 \leq i, j \leq m,\ i \neq j: \ M_i \neq M_j$), persona prompting would not be necessary, but might still add value.

There are various kinds of teams t possible in the framework:

- When $t = \{X_0\}$, the generator X_0 is "alone"

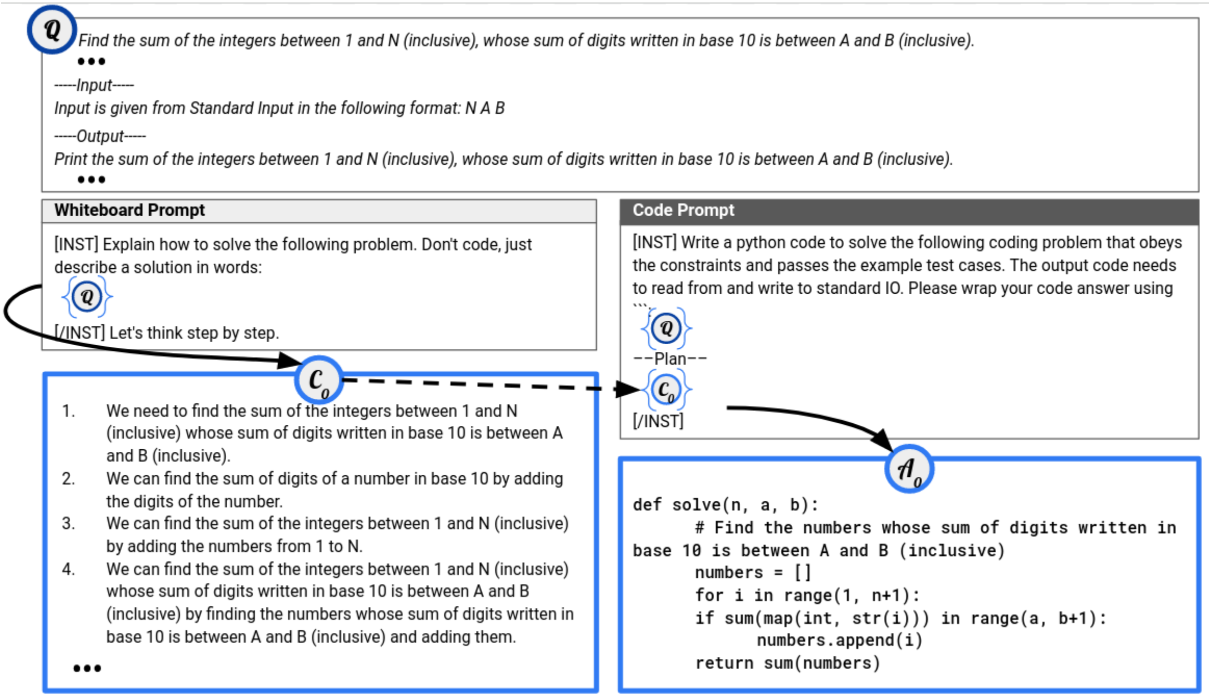


Figure 3: Example generation using the Whiteboard-then-Code framework, without iterative improvement nor collaboration.

(but Self-Refine may optionally be applied). This serves as baseline (or experimental control) against which the performance of other teams can be compared.

- When $t = \{X_0, \dots, X_m\}$, each teammate X_i comprising the group is different. In particular, this is intended to simulate collaboration amongst distinct experts (each bringing a different perspective) and in this work is achieved by prompting each LLM instance differently, for example, to focus on particular evaluation aspects, to borrow terminology from (Fu et al., 2023). For example, one teammate X_1 may focus on algorithmic efficiency (space and time complexity) (Madaan et al., 2023a), another X_2 on readability or simplicity, and a third X_3 on logical correctness and/or coherence.

The framework allows for two possible **phases**, or intervention points at which Team-Refine can be applied.

1. **Whiteboard**: Results in a technical approach, written in words rather than code, to solving the problem. This may include (1) high-level details such as which data structures and algorithms are involved, in addition to chain-of-thought reasoning; and (2) low-level details such as a step-by-step procedure for implementing the solution (bordering on pseudocode), and consideration of test and edge cases. The generator is prompted to give its best step-by-step outline for solving the problem: the initial content plan C_0. n iterations of feedback and refining culminate in content plan C_n.

2. **Code**: Results in a coded-up solution attempt. Denote as A_0 any first attempt generated by leveraging the final content plan C_n as additional context (appended to the question/instruction prompt).[5] r feedback-refine iterations culminate in attempt A_r, that is, the finalized solution (synthetic program). Previous works on self-feedback applied to coding appear to focus on this phase, for example in Nijkamp et al. (2023) and Madaan et al. (2023b). Collaboration in this development phase is reminiscent of code reviews, pull requests, and pair or team programming.

Intuitively, phases simulate the two parts of a technical interview for a software-engineering position, or, more abstractly, different steps in the software-development lifecycle (SDLC).

[5]For evaluation with the pass@k metric, the generator would produce k such attempts per question: $A_0^{(1)}, \dots, A_0^{(k)}$.

In each phase, use the term **collaboration strategy** S to refer to a potential configuration of Team-Refine. In particular, it defines a pair $S = (t,\ l)$, where t is a set of team-members and l is a list indicating which particular team-member gives feedback each round. The number of iterations of Team-Refine is given by the length of l.[6]

In this work, S_{code} is held fixed for all experiments. In particular, feedback-refine is always skipped in the Code phase such that coding attempt A_0 constitutes the final submission, and there is only a single team-member – the generator itself (the same model as the plan generator):

$$S_{\text{code}} = (\{X_0\},\ None)$$

In contrast, collaboration strategies in the Whiteboard phase S_W vary from experiment to experiment:

1. $S_W^{(1)} = None$
2. $S_W^{(2)} = (\{X_0\},\ None)$
3. $S_W^{(3)} = (\{X_0\},\ [X_0])$
4. $S_W^{(4)} = (\{X_0,\ X_1\},\ [X_1])$
5. $S_W^{(5)} = (\{X_0\},\ [X_0,\ X_0])$
6. $S_W^{(6)} = (\{X_0,\ X_1\},\ [X_1,\ X_1])$
7. $S_W^{(7)} = (\{X_0,\ X_1,\ X_2\},\ [X_1,\ X_2])$

5 Experiments

Overall, the experiments are designed to test both the viability of Whiteboarding for the code-generation task, and the efficacy of different configurations of Team-Refine.

5.1 Inference

Each experiment carries out the collaboration strategy defined in its particular specification S. The pool of questions is kept constant: the first 650 introductory-level questions in the test split of the APPs dataset. The setting is zero-shot (so no prompts provide demonstrations/examples).

For generating code, following Rozière et al. (2023, Appendix E) I decoded using nucleus sampling (p = 0.95) with temperature set to 0.6. (The randomness that sampling introduces is essential to the pass@k benchmark, to ensure output diversity across the k attempts.) For simplicity, greedy decoding was used for all plan, feedback, and rewrite generations.

Generations were cut off after 256, 128, and 320 new tokens respectively for plans, feedback, and rewrites. This was done to limit excessively long context lengths to achieve reasonable generation times, and based on observations during cross-validation that generations exceeding these lengths were often redundant or completely degenerate (e.g., repeating the same phrase line after line).

5.2 Prompt Designs

All prompts are zero-shot. Prompt templates are the same across all experiments excepting feedback prompts, which are altered in turn to study the addition of distinct teammates. A high-level overview of designs follows, with full prompt templates available in Appendix A.

- **Plan Prompts**: For prompts requesting plans, the generator was asked to explain how to solve the given question, without coding. Experiment 2 directly studies whether CoT-prompting applied to these templates influences performance.

- **Feedback Prompts**: All prompts for feedback have the same structure, but, to elicit distinct personalities, I leveraged the special System tokens available with Code Llama models fine-tuned for instructions. In particular, a message informing the system of its persona/role was placed between a pair of opening and closing system tokens, right before the normal feedback instruction (which remained the same in all experiments). Two such personas are featured in this work, apart from the default "personality" of the base model (without any special system/meta message):
  - *Practical Software Engineer*: "You are a practical software engineer focused on implementation and testing."
  - *Algorithms Expert*: "You are an algorithms expert focused on correctness and efficiency, including time complexity and space complexity."

- **Rewrite Prompts**: For prompts soliciting rewrites, I took advantage of Code Llama Instruct's dialogue capability. Specifically, the

[6] An empty pair $S = None$ indicates an empty configuration – that phase is skipped entirely.

context handed to the generator included the original plan and corresponding feedback formatted as a conversation history between a user and the model itself. While limited cross-validation suggested the model responded better to rewrite prompts formatted in this manner than ones where the original plan and feedback were packed back-to-back in one long user instruction, it is not clear this would hold true in general for other models.

5.3 Experiment Details

Experiment 1: $S_W^{(1)} = None$

This is the baseline intended to capture how the generator performs without the Whiteboard phase – just coding.

I attempted to replicate Code Llama Instruct 7B's zero-shot results on introductory-level questions in the APPS dataset, as reported by Rozière et al. (2023, Appendix E) in the Code Llama release paper. I used the exact same zero-shot prompt template for code generation (see Figure 13 of the Code LLama paper). Figure 9 in the appendix reproduces this template.

Experiment 2: $S_W^{(2)} = (\{X_0\},\ None)$

This study introduces the Whiteboard phase before the Code phase. In particular, this experiment tests two designs for the plan prompt:

(A) Without CoT prompting: The APPS question comes after "Explain how to solve the following problem. Don't code, just describe a solution in words".

(B) With CoT prompting: Identical to the prompt above, except the phrase *"Let's think step by step"* is appended to the prompt (Kojima et al., 2023). (Results from cross-validation suggested that, for this particular generator, the best placement for the phrase was after the special closing instruction token.[7])

For concrete visuals of the plan prompt used in Experiment 2B, see Figures 11 and 16 in the appendix.

[7]While Kojima et al. (2023) recommends appending it to the original prompt, it was ambiguous how to implement this for models with instruct tokens. For example, an alternative is to include it inside the instruction boundary such that the model interprets it as part of the user dialogue, instead of the beginning of its own response.)

This experiment also required adapting the code prompt to include the whiteboarded plan as additional context. Various candidate designs were considered in small-scale trials (each roughly involving between five and 50 APPS questions). I determined the most promising by manually reviewing code outputs for quality and doing automatic evaluation on them. Figures 10 and 22 show the winning design.

Figure 3 gives a high-level outline of this experiment.

Experiment 3: $S_W^{(3)} = (\{X_0\},\ [X_0])$

This study examines a single round of feedback-refine in the Whiteboard phase, with **self** feedback – that is, no special prompting to elicit an expert/peer persona was done.

This experiment required designing the feedback and rewrite prompts. The best performing pair of feedback and rewrite prompts were chosen through extensive cross-validation (for which the first 100 introductory-level questions in the train split were used). Appendix Figure 12 illustrates the base feedback prompt design, while Figures 15 and 20 show that of the rewrite prompt.

Experiment 4: $S_W^{(4)} = (\{X_0,\ X_1\},\ [X_1])$

This study again looks at a single round of feedback-refine in the Whiteboard phase, but this time with the feedback provided by a (single) peer LLM, prompted as an expert.

Two distinct personas were tested: (A) "practical software engineer" and (B) "algorithms expert". Figures 13 and 14 show the template, while Figure 18 shows an instantiated version of (A).

Appendix B exemplifies Experiment 4A: focusing on a particular question as an example, it shows all artifacts involved, including every instantiated prompt, intermediate generation (plans, feedback, and rewrites), and code generation, providing a full-circle view of the Whiteboard-then-Code framework with Team-Refine.

Experiment 5: $S_W^{(5)} = (\{X_0\},\ [X_0,\ X_0])$

This study tests the effect of twice iterating the feedback-rewrite process in the Whiteboard phase, where the feedback is self-provided.

This is achieved by reusing the re-written plans C_1 the generator produced in Experiment 3, and passing them in as input to the exact same script, resulting in twice-reworked plans C_2.

Experiment 6: $S_W^{(6)} = (\{X_0,\ X_1\},\ [X_1,\ X_1])$

This study tests the effect of twice iterating the feedback-rewrite process in the Whiteboard phase, where this time, the feedback comes from a (single) peer in both iterations. The peer LLM is prompted as an expert, specifically, to assume the persona of an algorithms expert (from 4B). (This persona was used over the other that was tested in 4 because it performed better during evaluation in raw-score terms, although the difference was not statistically significant.) The peer's prompting is identical both rounds (that is, the feedback persona is invariant).

Experiment 7:
$S_W^{(7)} = (\{X_0,\ X_1,\ X_2\},\ [X_1,\ X_2])$

This study tests the effect of twice iterating the feedback-rewrite process in the Whiteboard phase, but this time, the feedback comes from a different peer in each iteration. Both the first and second peers X_1 and X_2 are prompted as experts, but their prompting differs to elicit different behaviors and focus on different aspects of the plan generations. X_1 assumes the persona of a practical software engineer, whereas X_2 assumes the persona of an algorithms expert.

5.4 Secure Evaluation Setup

Because executing LLM-generated code poses a security risk, I built a sandbox following Chen et al. (2021). In particular, I only ran generated code inside of a containerized runtime isolated with gVisor from the host computer. Moreover, external networking was disabled on the container. For additional details, see Appendix C.

6 Results

Table 1 gives the generator's zero-shot performance on the first 650 introductory-level questions in the test split of the APPS dataset. The benchmark pass@k metric is used for five values of k, $\{1, 2, 3, 4, 5\}$ corresponding to the number of tries the generator is granted to pass the test cases. Due to resource constraints, this scale deviates from the task literature – for example, Rozière et al. (2023) reports metrics with substantially larger k-values – $k \in \{5, 10, 100\}$.

Figure 4 depicts pass@5 performance on APPS on all experiments, showing significance with Wilson Confidence Intervals at a 95% confidence level. Interested readers are directed to Appendix section D for confidence intervals corresponding to all entries in Table 1.

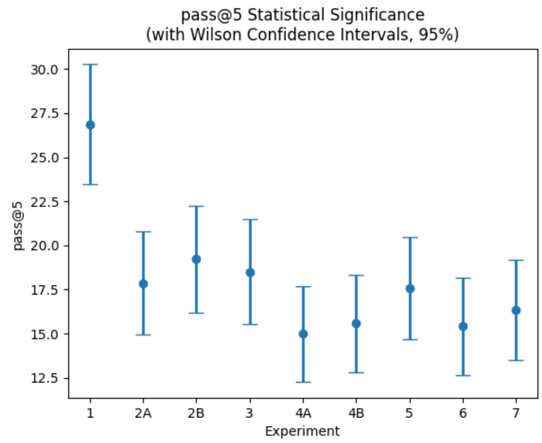


Figure 4: pass@5 performance on APPS, with error bars depicting Wilson Confidence Intervals at a 95% confidence level. For all experiments, true performance is estimated based on the success rate achieved on the first 650 samples of introductory level questions from the test split of the APPS dataset. See Table 2 in the appendix for pass@k confidence intervals for all values of k by experiment.

Specific discussion on each experiment follows:

Experiment 1 – The results of the Code Llama paper were successfully replicated: the pass@5 metric reported in the paper for the generator of 24.9% is very close to the value achieved here of 26.73%.

Experiment 2 – Performance relative to the baseline significantly declined, but only slightly so, with the introduction of content-planning. The prompt-template with CoT prompting performed slightly better than the version without it, but not significantly so.

Experiment 3 – Layering on single-round self-refine led to significantly worse performance than the baseline, but about the same performance as without any Self-Refine.

Experiment 4 – The feedback given by both of the peer personas tested ("practical software engineer" and "algorithms expert") resulted in comparable performances, and again, both were significantly below the baseline. For both 4A and 4B, the raw scores were noticeably lower than Experiment 3's (Self-Refine, no peers), but this was not significant.

Experiment 5 – Twice iterating the self-feedback self-rewrite process led to no significant change from doing so for a single iteration.

Experiment 6 – Twice iterating the feedback-refine process from the same peer (the algorithms expert 4B) led to little noticeable change from do-

Experiment	Pass@1	Pass@2	Pass@3	Pass@4	Pass@5
1	**16.84**	**21.18**	**23.63**	**25.35**	**26.73**
2 A.	11.06	13.95	15.62	16.80	17.67
2 B.	11.03	14.55	16.47	17.88	19.05
3	10.03	13.69	15.78	17.20	18.31
4 A.	8.28	11.02	12.66	13.84	14.77
4 B.	8.49	11.35	13.08	14.34	15.38
5	8.95	12.14	14.26	15.94	17.38
6	8.12	11.14	13.02	14.31	15.23
7	8.58	11.77	13.71	15.11	16.15

Table 1: pass@k on APPS introductory questions, by experiment, for $k \in \{1, 2, 3, 4, 5\}$. pass@k simply reports the percentage of questions on which at least one of the k allowed solution attempts is correct. Correctness is a strict all-or-nothing measure: an attempt is only considered correct if it passes all annotated test cases for the given question. Experiment 1 successfully replicated Instruct Code Llama 7B's zero-shot pass@5 performance on introductory problems (Rozière et al., 2023, Appendix E). This baseline – without Whiteboarding or Team-Refine – eclipses the performance of all other collaboration strategies studied here, and for all values of k, this observation is statistically significant. (See D).

ing so for a single iteration.

Experiment 7 – Feedback from each of the two peers in turn (two rounds total) did not lead to gains over any of the single-round experiments, nor any of the other experiments.

6.1 Analysis

That the Whiteboard step failed to bring gains in the APPS setting with this particular generator is possibly owing to the model's small size. Specifically, for whiteboarding to work on coding challenges, it is likely that the model's complex reasoning skills would need to be very high, yet smaller models tend to be worse in this domain. Moreover, adding the plan as another long segment of text to the context may have confused the model at points. APPS questions are already long and dense to begin with, and the benchmark is unforgiving in the sense that correct answers require painstaking attention to minor details. For example, passing test cases often hinges on whether the generator heeds a command to "read from standard input and write to standard output." Short but crucial instructions like these may have been buried/lost by adding more to the code prompts, thereby distracting/throwing off the generator.

While at first surprising that the generator's performance across experiments stagnated – despite being given the chance to adjust its whiteboarded plans (even multiple times for later experiments) – it is consistent with findings from Madaan et al. (2023b, See section 4 Analysis, subsection: Does SELF-REFINE work with weaker models?) that Self-Refine is liable to break down with weaker models. Indeed, inspection of generated outputs reveal that the version of Code Llama used in this work often had similar problems to those the authors identify as afflicting weaker models, including failure to properly follow both the feedback and refine prompts, and tending to generate hallucinations and rambling repetitive content.

The following Section 7 Error Analysis discusses the above sticking points in more detail.

7 Error Analysis

Manually inspecting sample outputs to identify potential patterns causing low-quality generations led to the following observations.

7.1 Whiteboard Phase

Superficially, most plans are appropriately structured, but some are unhelpful. In the vast majority of cases, the model appeared to produce a document resembling instructions or a roadmap during the Whiteboard phase. These plans usually enumerated a series of steps, often in the form of a numbered list, and mention variables, functions, data structures, and algorithms that should be involved in the ultimate implementation. In many cases, it lays out its reasoning and gives justification for certain choices. In other cases, it seems to copy too much from the question instructions, resulting in unhelpful blueprints that merely restate the question. Figure 5 exemplifies this error occurring in Experiment 2B.

The model occasionally disobeys the instruction to avoid code. Figure 6 provides four example cases where this occurs in Experiment 2B. When asked to create the content plan, the generator often could not resist writing Python code, despite being explicitly told not to. This occurred in roughly 10% of plan generations. In a few of these cases, the model was just providing an instructional snippet, for example, to demonstrate a subroutine it had just mentioned. But for most of these cases, the generator was writing the entire solution. This was likely another source of confusion for the model when it came time to code: code already present in the context in addition to the question. This issue was made worse by the fact that, more often than not when the generator began writing code, it was cut off mid-line by the token limit – so such codes were frequently unfinished.

7.2 Feedback-Refine

The root cause of issues for Team-Refine appears to be a breakdown at the feedback step. In particular, it seems that recommendations were frequently unhelpful, or the feedback contained too much data. It is thus understandable how the quality of the generator's rewritten plans (and subsequently, its code generations) might degrade, and possibly suffer even worse from the errors identified in 7.1.

In giving feedback, actual suggestions are output most of the time, but they're of limited usefulness. Generated feedback sometimes included dramatic suggestions that could prove helpful, like a overhaul to the approach's algorithm. But a lot of the time, suggestions were vague, minor, and in the worst of cases, misguided. Figure 7 illustrates this sticking point by juxtaposing examples of good and bad feedback from Experiment 3.

For feedback, the model frequently ignores instructions to provide only one suggestion. This potentially overwhelms the generator when it is tasked with refining in light of this feedback. Figure 8 illustrates this common error type with examples from Experiment 3.

8 Conclusions, Limitations, and Future Work

In summary, this project introduced both (1) the concept of Whiteboarding, inference-time content-planning for code-generation tasks, and (2) a framework for (team-)refining via automatic feedback in various forms (both from the generator itself and from a diverse set of peer LLMs) to improve plans. Neither (1) nor (2) led to performance gains over the baseline, for the particular (relatively small) model used as the generator, on the particular dataset used in this work.

The negative results validate findings from the Self-Refine paper that smaller models do not enjoy performance gains from the iterative process, and the more general observation that weaker models struggle to follow instructions. Naturally, it would be interesting to see follow-on work using a larger model to carry out the Whiteboard-then-Code framework. In the same vein, having different underlying models serve as teammates (with or without persona prompting) may offer a way forward to improve results. As far as open models are concerned, a good candidate is Llama 2 (Touvron et al., 2023).

Another interesting direction would be exploring in-context learning within the Whiteboard-then-Code framework: How would including several demonstrations in the prompt templates (for plans, feedback, and rewrites) affect the quality of (intermediate and coding) outputs?

One major challenge in this work was prompt design: for each template, countless permutations in phrasing and structure are possible. While prompt design is a source of headaches in general across the field, the pains may be particularly acute in this framework, since it requires different templates for plans, feedback, rewrites, and each persona involved. While some amount of cross-validation was done at various points to decide on a prompt that seems to work the best (on the development split), the sheer number of possibilities made it impossible to satisfactorily treat the larger issue here. It is left to future work to develop a more systematic approach to deciding on prompt designs when using the Whiteboard-then-Code framework, so as to optimize model outputs. (Because the "right" design likely depends on the idiosyncrasies of the particular model being prompted, I believe general guidance on how to decide would be the most helpful contribution on this front.)

Generated Plan (Error Analysis – Unhelpful plans that just restate the question)

1. We need to check if the last character in A and the initial character in B are the same.
2. We need to check if the last character in B and the initial character in C are the same.

If both conditions are true, we print YES. Otherwise, we print NO.

So, we need to write a program that checks if the last character in A and the initial character in B are the same, and if the last character in B and the initial character in C are the same. If both conditions are true, we print YES. Otherwise, we print NO.

(a) In this example, the generator largely copies from the question or restates it, instead of giving an algorithmic approach or implementation plan as expected.

Question (Error Analysis – Unhelpful plans that just restate the question)

You are given three strings A, B and C. Check whether they form a word chain. More formally, determine whether both of the following are true:
- The last character in A and the initial character in B are the same.
- The last character in B and the initial character in C are the same.

If both are true, print YES. Otherwise, print NO.

-----Constraints-----
- A, B and C are all composed of lowercase English letters (a - z).
- $1 \leq |A|, |B|, |C| \leq 10$, where $|A|$, $|B|$ and $|C|$ are the lengths of A, B and C, respectively.

-----Input-----
Input is given from Standard Input in the following format:
A B C
-----Output-----
Print YES or NO.
-----Sample Input-----
rng gorilla apple
-----Sample Output-----
YES
They form a word chain.

(b) The corresponding question for which the above output was generated.

Figure 5: The generator has a tendency to regurgitate information already present in the question, occasionally producing unhelpful plans. Such duplicate information may distract the generator in the Code phase.

Generated Plan (Error Analysis – plan includes code)

1. First, we need to get the input from the user.
2. Then, we need to calculate the value of a + a^2 + a^3.
3. Finally, we need to print the result.

Let's start with the first step. We can use the input() function to get the input from the user.

Next, we need to calculate the value of a + a^2 + a^3. We can use the power operator to calculate the value of a^2 and a^3.

Finally, we need to print the result. We can use the print() function to print the result.

Here is the code:

```
a = input()
a_plus_a_squared_plus_a_cubed = a + a**2 + a**3
print(a_plus_a_squared_plus_a_cubed)
```

Note that we need to convert the input to an integer before we can use it in the calculation. We can use the int() function to convert the input to an integer.

Here is the complete code:

```
a = int(input())
a_
```

(a) The generator hardly starts to code up a solution before hitting the token limit.

Generated Plan (Error Analysis – plan includes code)

1. We have a digit sequence S of length 4.
2. We are wondering which of the following formats S is in:
   * YYMM format: the last two digits of the year and the two-digit representation of the month (example: 01 for January), concatenated in this order
   * MMYY format: the two-digit representation of the month and the last two digits of the year, concatenated in this order
3. If S is valid in only YYMM format, print YYMM; if S is valid in only MMYY format, print MMYY; if S is valid in both formats, print AMBIGUOUS; if S is valid in neither format, print NA.

Let's break down the problem into smaller parts.

1. Check if the last two digits of the year are valid.
2. Check if the two-digit representation of the month is valid.
3. If both conditions are true, print AMBIGUOUS.
4. If only one condition is true, print the corresponding format.
5. If neither condition is true, print NA.

Now, let's write the code

(b) The generator wants to code, but run up on the token limit right before doing so.

Figure 6: The first and second of four examples where the generator produces code during the Whiteboard phase, in spite of being directed not to do so in the prompt. (Figure continues on next page.)

Generated Plan (Error Analysis – plan includes code)

1. First, we need to sort the restaurants by their city names in lexicographical order.
2. Then, we need to sort the restaurants in each city by their scores in descending order.
3. Finally, we need to print the identification numbers of the restaurants in the order they are introduced in the book.

Let's write the code in Python:

```
# Sort the restaurants by their city names in lexicographical order
restaurants.sort(key=lambda x: x[0])

# Sort the restaurants in each city by their scores in descending order
for city in restaurants:
	city.sort(key=lambda x: x[1], reverse=True)

# Print the identification numbers of the restaurants in the order they are introduced in the book
for restaurant in restaurants:
	print(restaurant[2])
```

In this code, we first sort the restaurants by their city names in lexicographical order using the `sort()` method. Then, we sort the restaurants in each city by their scores in descending order using the `sort()` method again. Finally, we print

(c) After stating its plan of attack, the generator implements it in code, and even has time to start a summary afterward before hitting the max_new_tokens limit.

Generated Plan (Error Analysis – plan includes code)

1. First, we need to count the number of times each string appears in the input. We can do this by creating a dictionary where the keys are the strings and the values are the number of times they appear.
2. Next, we need to find the string that appears the most number of times. We can do this by keeping track of the maximum number of times a string appears and the string that appears that number of times.
3. Finally, we need to print all the strings that appear the maximum number of times in lexicographical order. We can do this by sorting the strings in the dictionary by their values (i.e., the number of times they appear) and then printing them in order.

Here's an example of how this solution might look in code:

```
# Read in the input
n = int(input())
strings = []
for i in range(n):
	strings.append(input())

# Create a dictionary to keep track of the number of times each string appears
counts = {}
for string in strings:
	if string in counts:
	counts[string] += 1
	else:
	counts[string] =
```

(d) The generator has enough remaining tokens to write a fair amount of code, but does not enough to finish.

Figure 6: (Continued from previous page) The third and fourth examples where the generator produces code during the Whiteboard phase, in spite of being directed not to do so in the prompt.

Generated Feedback (Error Analysis – Potentially Helpful Algorithm Overhaul)

One suggestion to help improve your approach is to use a more efficient algorithm to find the LCM of all the divisors in the list. One such algorithm is the Euclidean algorithm, which has a time complexity of $O(log\ min(d_1, d_2, \dots, d_n))$. This is much faster than the naive approach of simply multiplying all the divisors together, which has a time complexity of $O(n^2)$. By using the Euclidean algorithm, you can find the LCM of all the divisors in the list in $O(log\ min(

(a) Good Feedback Example: The generator suggests a potentially helpful algorithm overhaul.

Generated Feedback (Error Analysis – Misguided suggestion)

Your approach is good, but you can simplify the calculations by noticing that A + B = A - (-B) = A + (-B) = A - B, and A - B = A + (-B) = A - (-B) = A + B. This means that the largest number among A + B, A - B, and A * B is A + B. You can also use the fact that A * B = -A * (-B) = -A * B = -A + B to simplify the calculations. Additionally, you can use the fact that A - B

(b) Bad Feedback Example: Incorrect arithmetic muddles the attempt at giving a suggestion. This could easily throw the generator off in the rewrite stage and ultimately result in incorrect coding.

Figure 7: A side-by-side view of good vs. degenerate feedback.

Generated Feedback (Error Analysis – More than one suggestion)

Your approach is good, but it can be improved by considering the following:

1. You can start by finding the minimum number of operations to obtain $A \le B$ such that B is divisible by A.
2. You can then find the minimum number of operations to obtain $A \le B \le C$ such that B is divisible by A and C is divisible by B.
3. You can use a greedy approach to solve this problem, where you start with the smallest number and try to increase it by 1 as much

(a) At least two, arguably three suggestions are given here.

Generated Feedback (Error Analysis – More than one suggestion)

Your approach is good, but it can be improved by considering the following:

1. Instead of applying the algorithm to the entire array, you can apply it to each element separately. This will reduce the time complexity of the algorithm.
2. You can use a hash table to store the final values of each element after applying the algorithm. This will reduce the space complexity of the algorithm.
3. You can use a dynamic programming approach to solve the problem. This will reduce the time complexity of the algorithm and also reduce the space complexity.
4. You can use a greedy approach to solve the problem. This

(b) Four pieces of feedback are given here; the third and fourth pieces of advice appear to be in direct conflict with one another.

Figure 8: The first and second of four examples where multiple pieces of feedback are produced per prompt, in spite of being asked to produce a single suggestion. (Figure continues on next page.)

Generated Feedback (Error Analysis – More than one suggestion)

One suggestion to help improve your approach is to consider the following:

* Instead of considering all possible combinations of microtransactions and special offers, consider only the combinations that are relevant to the current day.
* Use a dynamic programming approach to solve the problem, where you keep track of the minimum day when Ivan can order all microtransactions he wants for each type of microtransaction and each day.
* Use a hash table to store the information about the special offers, where the key is the type of microtransaction and the value is the day when the offer starts.
* Use a hash table to store the information

(c) At least three distinct pieces of advice are given. The fourth point, which gets cut off, appears slated to simply repeat the previous one.

Generated Feedback (Error Analysis – More than one suggestion)

Your approach is good, but there is a small mistake in the formula for finding the maximum height of the decorated trees. Instead of subtracting the height of the tallest tree that is not decorated from the height of the tallest tree, you should subtract the height of the tallest tree that is not decorated from the height of the tallest tree that is decorated. This will give you the correct maximum height of the decorated trees.

Here's a suggestion to help improve your approach:

1. Instead of finding the maximum height of the trees that are not decorated, you can also

(d) After providing what seems to be good advice, the model insists on providing another suggestion (potentially a repeat of the first) before hitting the token limit.

Figure 8: (Continued from previous page) The third and fourth examples where multiple pieces of feedback are produced per prompt, in spite of being asked to produce a single suggestion.

Finally, the decision was made to set maximum new token thresholds at 256, 128, and 320 respectively for plans, feedback, and rewrites. This was done to limit excessively long context lengths to achieve reasonable generation times, and based on observations during cross-validation that generations exceeding these lengths were often redundant or completely degenerate (e.g., repeating the same phrase line after line). However, this unavoidably resulted in instances where the model was interrupted before completing its thoughts, some of which may have been key to the task at hand. Researchers who can afford longer context lengths and/or are armed with more robust stop-generation mechanisms[8] are encouraged to take up this framework.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4.

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

[8]Some of my later experiments attempted to control response length by including, "Respond in five sentences or fewer" in the prompt, following Shuyang Xiang's blog post here: https://towardsdatascience.com/challenges-in-stop-generation-within-llama-2-25f5fea8dea2

Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Alpacafarm: A simulation framework for methods that learn from human feedback.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2023. Gptscore: Evaluate as you desire.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large language models are zero-shot reasoners.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step.

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023a. Learning performance-improving code edits.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023b. Self-refine: Iterative refinement with self-feedback.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis.

OpenAI. 2023. Gpt-4 technical report.

Ethan Perez, Saffron Huang, Francis Song, Trevor Cai, Roman Ring, John Aslanides, Amelia Glaese, Nat McAleese, and Geoffrey Irving. 2022. Red teaming language models with language models.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Yixuan Su, David Vandyke, Sihui Wang, Yimai Fang, and Nigel Collier. 2021. Plan-then-generate: Controlled data-to-text generation via planning.

Yufei Tian and Nanyun Peng. 2022. Zero-shot sonnet generation with discourse-level planning and aesthetics features.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

A Prompt Templates

Figures 9, 10, 11, 12, 13, 14, and 15 show the templates that were used for experiments, with detailed descriptions of design choices and references in the captions. In all graphics, whitespace and newlines were tweaked for clarity.

Code Prompt (No Whiteboard Phase)

```
[INST] Write a python code to solve the
following coding problem that obeys the
constraints and passes the example test
cases. The output code needs to
{QUESTION_GUIDE}. Please wrap your
codeanswer using ```:

{question}

{starter_code}[/INST]
```

Figure 9: This code prompt is only used for the baseline, Experiment 1. It replicates the template in Rozière et al. (2023, Figure 13).

Code Prompt (After Whiteboard Phase)

```
[INST] Write a python code to solve the
following coding problem that obeys the
constraints and passes the example test
cases. The output code needs to
{QUESTION_GUIDE}. Please wrap your
codeanswer using ```:

{question}

-----Plan-----
{plan}

{starter_code}[/INST]
```

Figure 10: The prompt for code that supplies the plan from the Whiteboard phase as additional context for the generator. It is identical to the one in Figure 9, but appends the plan. The plan is slotted in and formatted in a manner that fits in with how APPS formatted many of its question sub-section headings, and this choice seemed to work best in cross-validation. All experiments after Experiment 1 use this coding prompt template.

B Full-Circle Example of Framework

The series of figures 16–27 demonstrate how the prompt templates are instantiated, side-by-side with the generations they beget. The example APPS question is sourced from the test set and is rated as introductory. Generations shown are from Experiment 4A, where feedback was given by a “practical software engineer” persona; this configuration ultimately leads to two correct coded-up solutions (specifically, the first and fourth attempts out of the five pass all test cases) and so passes this question in pass@5 terms. Bolding highlights what has changed from prompt to prompt, and demonstrates how the prompts in a team-refine cycle incrementally build. The series of prompts and generations also emphasize the large context sizes the generator faces: APPS questions are long to begin with, and this framework piles on additional context load.

```
Plan Prompt

[INST] Explain how to solve the
following problem. Don't code, just
describe a solution in words:

{question}

[/INST] Let's think step by step.
```

Figure 11: The same prompt is used in all experiments to prompt for the original plan C_0 (excepting Experiment 2, which tests two versions, the better performing of which is displayed here and used for all ensuing experiments). The design works off the template in Rozière et al. (2023, Figure 13). In addition, both the simplicity, and the specific wording of the negative instruction of the form "Don't do something, just do something else" for output control purposes, are inspired by Madaan et al. (2023b, Figure 22: FEEDBACK prompt for Code Readability). Finally, the phrase *"Let's think step by step"* is appended to the prompt to elicit chain-of-thought reasoning (Kojima et al., 2023).

```
Feedback Prompt

[INST] Review my approach to solving
the following problem:

{question}

My approach:

{old_plan}

Give one suggestion to help improve my
approach. Respond in five sentences or
fewer.

[/INST]
```

Figure 12: The basic feedback prompt, for the generator to self-critique its own content plans. The design of the template is inspired by choices in Madaan et al. (2023b, especially Figure 22: FEEDBACK prompt for Code Readability).

C Sandbox for Executing Generated Code

I mimicked the testing sandbox setup used in Chen et al. (2021, Section 2.3. *Sandbox for Executing Generated Code*). My procedure is summarized below:

1. Set up gVisor[9][10] on the VM dedicated for testing, and learn how to run containers[11] with it.
2. Disable external networking for Docker containers[12].

Please see the public GitHub repository for more details on testing/evaluating using this infrastructure: https://github.com/N-G-Asker/whiteboard-then-code.

D Statistical Confidence Intervals

Table 2 gives Wilson confidence intervals with $\alpha = 0.95$ for the pass@k APPS results reported in Table 1. The idea to use the Wilson Confidence Interval comes from Madaan et al. (2023b, Appendix J: Statistical Confidence Intervals), where this calculation is used to determine statistical significance when comparing performance across multiple models on the same benchmark. (The results reported in Madaan et al. (2023b, Compare Table 1 and Table 13) appear to assume the Wilson Confidence Interval is centered about the raw experimental success rate $\hat{p}$ – e.g., the solve-rate estimate – when in fact it is centered about an adjusted value of $\hat{p}$. Notice in particular how, in that work, the interval midpoints in Table 13 are the same values as the raw scores in Table 1.)

[9] https://gvisor.dev/
[10] https://gvisor.dev/docs/user_guide/install/
[11] https://gvisor.dev/docs/user_guide/quick_start/docker/
[12] https://gvisor.dev/docs/user_guide/networking/

Feedback Prompt – Persona

```
<s>[INST]

<<SYS>> {persona_directive} <</SYS>>

Review my approach to solving the
following problem:

{question}

My approach:

{old_plan}

Give one suggestion to help improve my
approach. Respond in five sentences or
fewer.

[/INST]
```

Figure 13: The feedback prompt for personas, intended to simulate a peer with expertise in a particular area giving feedback to the generator. The design of the template is inspired by choices in Madaan et al. (2023b, especially Figure 22: FEEDBACK prompt for Code Readability).

Feedback Prompt – Persona Example

```
<s>[INST]

<<SYS>> You are an algorithms expert
focused on correctness and efficiency,
including time complexity and space
complexity.<</SYS>>

Review my approach to solving the
following problem:

{question}

My approach:

{old_plan}

Give one suggestion to help improve my
approach. Respond in five sentences or
fewer.

[/INST]
```

Figure 14: The feedback prompt for personas, intended to simulate a peer with expertise in a particular area giving feedback to the generator. The design of the template is inspired by choices in Madaan et al. (2023b, especially Figure 22: FEEDBACK prompt for Code Readability).

Rewrite Prompt

```
<s>[INST] Review my approach to solving
the following problem:

{question}

My approach:

{old_plan}

Give one suggestion to help improve my
approach. Respond in five sentences or
fewer. [/INST]

{feedback} </s>

<s>[INST]

Now update my approach. [/INST]

Okay! Here is the new version of your
approach:
```

Figure 15: The prompt requesting the generator provide a reworked plan, in light of the feedback and old plan. The design of the template is inspired by choices in Madaan et al. (2023b, especially Figure 23: REFINE prompt for Code Readability, and Figure 32: REFINE prompt for Math Reasoning.).

Plan Prompt

```
[INST] Explain how to solve the following problem. Don't code, just describe a solution in words:
Find the sum of the integers between 1 and N (inclusive), whose sum of digits written in base 10 is between
A and B (inclusive).

-----Constraints-----
 - 1 \leq N \leq 10^4
 - 1 \leq A \leq B \leq 36
 - All input values are integers.

-----Input-----
Input is given from Standard Input in the following format:
N A B

-----Output-----
Print the sum of the integers between 1 and N (inclusive), whose sum of digits written in base 10 is between
A and B (inclusive).

-----Sample Input-----
20 2 5

-----Sample Output-----
84

Among the integers not greater than 20, the ones whose sums of digits are between 2 and 5, are:
2,3,4,5,11,12,13,14 and 20. We should print the sum of these, 84.

[/INST] Let's think step by step.
```

Figure 16

Generated Plan

1. We need to find the sum of the integers between 1 and N (inclusive) whose sum of digits written in base 10 is between A and B (inclusive).
2. We can find the sum of digits of a number in base 10 by adding the digits of the number.
3. We can find the sum of the integers between 1 and N (inclusive) by adding the numbers from 1 to N.
4. We can find the sum of the integers between 1 and N (inclusive) whose sum of digits written in base 10 is between A and B (inclusive) by finding the numbers whose sum of digits written in base 10 is between A and B (inclusive) and adding them.

Let's break down the problem into smaller parts.

1. Find the numbers whose sum of digits written in base 10 is between A and B (inclusive).
2. Find the sum of the numbers whose sum of digits written in base 10 is between A and B (inclusive).

Now, let's solve the first part.

1. We can

Figure 17: Note that the preset cutoff threshold (max_new_tokens) causes this generation to be incomplete.

Feedback Prompt

```
[INST]
<<SYS>>You are a practical software engineer focused on implementation and testing.<</SYS>>

Review my approach to solving the following problem:

Find the sum of the integers between 1 and N (inclusive), whose sum of digits written in base 10 is between
A and B (inclusive).
        ⋮
Among the integers not greater than 20, the ones whose sums of digits are between 2 and 5, are:
2,3,4,5,11,12,13,14 and 20. We should print the sum of these, 84.

My approach:
1. We need to find the sum of the integers between 1 and N (inclusive) whose sum of digits written in base
10 is between A and B (inclusive).
        ⋮
Now, let's solve the first part.

1. We can

Give one suggestion to help improve my approach. Respond in five sentences or fewer.
[/INST]
```

Figure 18: Note the incomplete/cutoff generations from prior steps are inserted "as is" without modification to later prompt stages.

Generated Feedback

One suggestion to help improve your approach is to use a more efficient algorithm to find the numbers whose sum of digits written in base 10 is between A and B (inclusive). Instead of iterating through all the numbers between 1 and N (inclusive) and checking their sum of digits, you can use a mathematical formula to directly find the numbers that satisfy the condition. For example, you can use the formula for the sum of the digits of a number in base 10, which is given by:

sum of digits = (number in base 10) mod 9 + ((number

Figure 19: Notice again the incomplete/cutoff generation.

Rewrite Prompt

```
[INST]
Review my approach to solving the following problem:

Find the sum of the integers between 1 and N (inclusive), whose sum of digits written in base 10 is between
A and B (inclusive).
         ⋮
My approach:
1. We need to find the sum of the integers between 1 and N (inclusive) whose sum of digits written in base
10 is between A and B (inclusive).
         ⋮
Give one suggestion to help improve my approach. Respond in five sentences or fewer.
[/INST]
  One suggestion to help improve your approach is to use a more efficient algorithm to find the numbers
whose sum of digits written in base 10 is between A and B (inclusive). Instead of iterating through all the
numbers between 1 and N (inclusive) and checking their sum of digits, you can use a mathematical formula to
directly find the numbers that satisfy the condition. For example, you can use the formula for the sum of
the digits of a number in base 10, which is given by:

sum of digits = (number in base 10) mod 9 + ((number </s>

<s>[INST]
Now update my approach.
[/INST]

Okay! Here is the new version of your approach:
```

Figure 20: Note the SYSTEM message is absent here, even though it was present in the prior step's prompt (prompt for feedback).

Generated Rewrite

1. We need to find the sum of the integers between 1 and N (inclusive) whose sum of digits written in base 10 is between A and B (inclusive).
2. We can find the sum of digits of a number in base 10 by adding the digits of the number.
3. We can find the sum of the integers between 1 and N (inclusive) by adding the numbers from 1 to N.
4. We can find the sum of the integers between 1 and N (inclusive) whose sum of digits written in base 10 is between A and B (inclusive) by finding the numbers whose sum of digits written in base 10 is between A and B (inclusive) and adding them.

Let's break down the problem into smaller parts.

1. Find the numbers whose sum of digits written in base 10 is between A and B (inclusive).
2. Find the sum of the numbers whose sum of digits written in base 10 is between A and B (inclusive).

Now, let's solve the first part.

1. We can use a mathematical formula to directly find the numbers that satisfy the condition. The formula is:

sum of digits = (number in base 10) mod 9 + ((number in base 10) mod 9)

This formula gives the sum of the digits of a number in base

Figure 21: .

Code Prompt

```
[INST] Write a python code to solve the following coding problem that obeys the constraints and passes the
example test cases. The output code needs to read from and write to standard IO. Please wrap your code
answer using ```:

Find the sum of the integers between 1 and N (inclusive), whose sum of digits written in base 10 is between
A and B (inclusive).
        ⋮
Among the integers not greater than 20, the ones whose sums of digits are between 2 and 5, are:
2,3,4,5,11,12,13,14 and 20. We should print the sum of these, 84.

-----Plan-----
1. We need to find the sum of the integers between 1 and N (inclusive) whose sum of digits written in base
10 is between A and B (inclusive).
2. We can find the sum of digits of a number in base 10 by adding the digits of the number.
        ⋮
sum of digits = (number in base 10) mod 9 + ((number in base 10) mod 9)

This formula gives the sum of the digits of a number in base
[/INST]
```

Figure 22

Code Generation [1 / 5]

```
import sys

def get_numbers_with_digit_sum(n, a, b):
    numbers = []
    for i in range(1, n+1):
    num = i
    sum_of_digits = 0
    while num > 0:
        sum_of_digits += num % 10
        num //= 10
    if a <= sum_of_digits <= b:
        numbers.append(i)
    return numbers

def main():
    n, a, b = map(int, sys.stdin.readline().split())
    numbers = get_numbers_with_digit_sum(n, a, b)
    print(sum(numbers))

if __name__ == "__main__":
    main()
```

Figure 23: Passes all test cases.

Code Generation [2 / 5]

```
def find_sum_of_integers_between_1_and_n(n, a, b):
    # Find the numbers whose sum of digits written in base 10 is between A and B
(inclusive)
    numbers = [i for i in range(1, n+1) if (i % 9 + (i % 9) == a and i % 9 + (i %
9) <= b)]

    # Find the sum of the numbers
    sum = 0
    for number in numbers:
    sum += number

    return sum

# Test the function with the given example
print(find_sum_of_integers_between_1_and_n(20, 2, 5)) # Output: 84
```

Figure 24: Fails at least one test case.

Code Generation [3 / 5]

```
def solve(n, a, b):
	# Find the numbers whose sum of digits written in base 10 is between A and B
(inclusive)
	numbers = []
	for i in range(1, n+1):
	if sum(map(int, str(i))) in range(a, b+1):
		numbers.append(i)
	return sum(numbers)
```

Figure 25: Fails at least one test case.

Code Generation [4 / 5]

```
import sys

def solve(N, A, B):
	# Find the numbers whose sum of digits written in base 10 is between A and B
(inclusive)
	numbers = []
	for i in range(1, N+1):
	num = i
	sum_of_digits = 0
	while num > 0:
		sum_of_digits += num % 10
		num //= 10
	if A <= sum_of_digits <= B:
		numbers.append(i)

	# Find the sum of the numbers whose sum of digits written in base 10 is between A and
B (inclusive)
	sum_of_numbers = 0
	for num in numbers:
	sum_of_numbers += num

	return sum_of_numbers

if __name__ == "__main__":
	N, A, B = map(int, sys.stdin.readline().split())
	result = solve(N, A, B)
	print(result)
```

Figure 26: Passes all test cases.

Code Generation [5 / 5]

```
def get_numbers_with_digit_sum_in_range(n, a, b):
      # Initialize a set to store the numbers
      numbers = set()

      # Iterate from 1 to n
      for i in range(1, n+1):
      # Convert the current number to base 10
      num_in_base_10 = int(str(i), 10)

      # Calculate the sum of digits of the current number
      sum_of_digits = (num_in_base_10 % 9) + ((num_in_base_10 % 9) * 2)

      # If the sum of digits is between A and B, add the number to the set
      if a <= sum_of_digits <= b:
            numbers.add(i)

      # Return the set of numbers
      return numbers
```

Figure 27: Fails all test cases.

Experiment	Pass@1	Pass@2	Pass@3	Pass@4	Pass@5
1	17.03 ± 2.88*	21.35 ± 3.14*	23.78 ± 3.26*	25.49 ± 3.34*	26.87 ± 3.39*
2.A.	11.29 ± 2.41	14.16 ± 2.66	15.82 ± 2.79	17.00 ± 2.87	17.86 ± 2.93
2.B.	11.26 ± 2.41	14.76 ± 2.71	16.67 ± 2.85	18.07 ± 2.94	19.23 ± 3.02
3	10.26 ± 2.31	13.90 ± 2.64	15.98 ± 2.80	17.39 ± 2.90	18.50 ± 2.97
4.A.	8.53 ± 2.13	11.25 ± 2.41	12.88 ± 2.56	14.05 ± 2.66	14.98 ± 2.73
4.B.	8.73 ± 2.15	11.58 ± 2.44	13.30 ± 2.59	14.55 ± 2.69	15.58 ± 2.77
5	9.19 ± 2.20	12.36 ± 2.51	14.47 ± 2.69	16.14 ± 2.81	17.57 ± 2.91
6	8.37 ± 2.11	11.37 ± 2.42	13.24 ± 2.59	14.52 ± 2.69	15.43 ± 2.76
7	8.82 ± 2.16	11.99 ± 2.48	13.92 ± 2.65	15.31 ± 2.75	16.35 ± 2.83

Table 2: Wilson confidence intervals with $\alpha = 0.95$ for the pass@k APPS results reported in Table 1. This follows Madaan et al. (2023b, Appendix J: Statistical Confidence Intervals), where this calculation is used to determine statistical significance when comparing performance across multiple models on the same benchmark. The asterisk (*) indicates that the difference between the baseline's performance and that of all other experiments is statistically significant under a 95% confidence interval.