

STORYCODER: BRIDGING NARRATIVES AND FORMALITY FOR CODE GENERATION IN LLMs

Anonymous authors

Paper under double-blind review

ABSTRACT

The code generation capabilities of large language models stem largely from pre-training on structured code patterns and their strong context-based reasoning abilities. However, previous research on code generation has primarily focused on using short, fragmented, instruction-like prompts, which often fail to encourage contextual understanding. Inspired by the way humans organize fragmented information into coherent explanations, we propose a new method that reformulates coding problems as natural language narratives to promote integrative thinking. To this end, we introduce STORYCODER, a framework that reformulates code generation prompts into narrative text. Our results show that rich contextual expressions in natural language can enhance code generation performance and lead models to adopt consistent and structured problem-solving strategies. We quantitatively demonstrate that our method provides integrative information not captured by simple rephrasings and guides models to adopt correct algorithms and implementation strategies, thereby improving code generation performance. Experiments on three benchmarks, HumanEval, CodeForces, and LiveCodeBench, show an average improvement of 28.6% in the precision of zero-shot pass@10.

1 INTRODUCTION

Problem-solving ultimately depends on how clearly the information is conveyed and understood. Effective problem solving may require both the solver’s capability to interpret information and the way the problem is framed (Vessey, 1991; Kelton et al., 2010). However, in practice, task descriptions are incomplete or ambiguous, forcing solvers to infer missing details from context. These gaps are especially challenging in complex tasks that demand contextual understanding or multi-step reasoning. Large language models (LLMs) face the same difficulty: their performance depends not only on internal reasoning but also on how effectively the task is specified and interpreted (Laban et al., 2025). In this work, we investigate how to improve the delivery and interpretation of information in LLMs, focusing on code generation tasks. Programming tasks are particularly suitable for this study: they are built on logically distinct structures, and their solutions can be explicitly validated using test cases (Wang et al., 2025; Light et al., 2025).

We introduce STORYCODER, a narrative-based prompting method that transforms short, instruction-like problem statements into coherent natural language. This design is grounded in cognitive science findings that humans comprehend and reason more effectively by organizing fragmented conditions into coherent mental models and using analogical structures to facilitate deeper reasoning (Johnson-Laird, 1983; Gentner, 1983; Holyoak & Lu, 2021). In this framework, models identify the appropriate algorithm that will form the logic of the code, align it with a suitable narrative genre, and reformulate it into a story with three sections: task overview, constraints, and example input/output (Figure 1.) By connecting fragmented conditions into coherent descriptions, narratives help LLMs capture context and follow more structured reasoning. We evaluate STORYCODER across closed-source and open-source models on three benchmarks—HumanEval, CodeForces, and LiveCodeBench. Extensive experiments demonstrate consistent performance gains, with an average improvement of 28.3%p in zero-shot pass@10 accuracy.

Beyond overall accuracy, we find that narrative prompts substantially increase the likelihood of selecting the correct algorithms and reduce implementation errors. In contrast, when narratives are expressed in mismatched genres, performance drops significantly. This suggests that LLMs

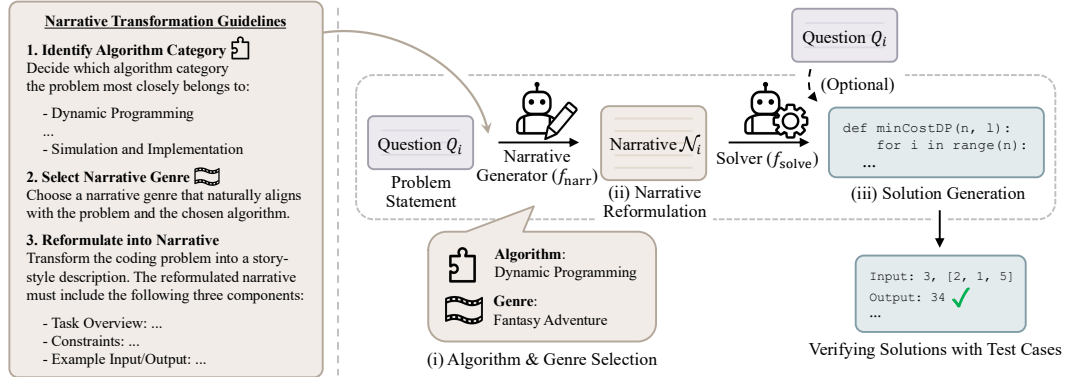


Figure 1: **Overview of STORYCODER framework.** Given a question Q_i , (i) model first identifies an algorithmic category and selects a narrative genre, then (ii) reformulates the problem into a structured narrative \mathcal{N}_i consisting of task overview, constraints, and example input/output, and (iii) passes the narrative (optionally with Q_i) to a solver model to generate code solutions, which are then verified with test cases.

have an inherent ability to understand narrative structures that align well with tasks such as code generation. These observations support our hypothesis that narratives are a natural and effective tool to encourage integrative reasoning.

Our key contributions are as follows:

- We propose STORYCODER, a narrative-based prompting method for code generation that reformulates fragmented prompts into coherent descriptions.
- We demonstrate consistent empirical improvements across diverse models and benchmarks, achieving a 28.3%p average gain in zero-shot pass@10 accuracy.
- We provide quantitative analyses showing that narrative reformulation guides LLMs in choosing the correct algorithms and reducing implementation errors.

2 RELATED WORK

Structured prompt engineering for code generation. Recent work in code generation has increasingly focused on structuring prompts to improve the reasoning process of LLMs. Some approaches introduce explicit intermediate steps, such as control structures or modular subcomponents, to guide program synthesis more reliably (Li et al., 2025a; Le et al., 2024; Huang et al., 2023). Recent work demonstrates that language-based input-output patterns can enable structured and verifiable reasoning in code generation (Li et al., 2025b). These strategies typically aim to align the internal reasoning process of the model with syntactic correctness and functional accuracy. In addition, continued pre-training on mathematical code helps models better handle symbolic expressions and abstract logic in programming tasks (Lu et al., 2025). While these approaches rely on explicitly structured reasoning or fine-tuned feedback signals, our method STORYCODER adopts a narrative-based prompt design that encourages integrative understanding without requiring explicit modular decomposition.

Prompt reformulation and test-time reasoning in LLMs. Another line of research examines how rephrasing prompts at test time affects the way LLMs reason on a task. Chain-of-thought prompting, especially when combined with self-consistency, helps models explore diverse reasoning paths and improves robustness through output aggregation (Wei et al., 2022; Wang et al., 2023). Beyond explicit reasoning steps, rephrasing task instructions influences how the model interprets the problem, regardless of whether the phrasing is optimized for its preferences or chosen more arbitrarily (Fu et al., 2024; Zhou et al., 2024). These findings indicate that subtle changes in phrasing can influence how models internally interpret a given task. Building on this insight, our method reformulates prompts into coherent narratives that embed conditions, intent, and examples within a single, unified structure.

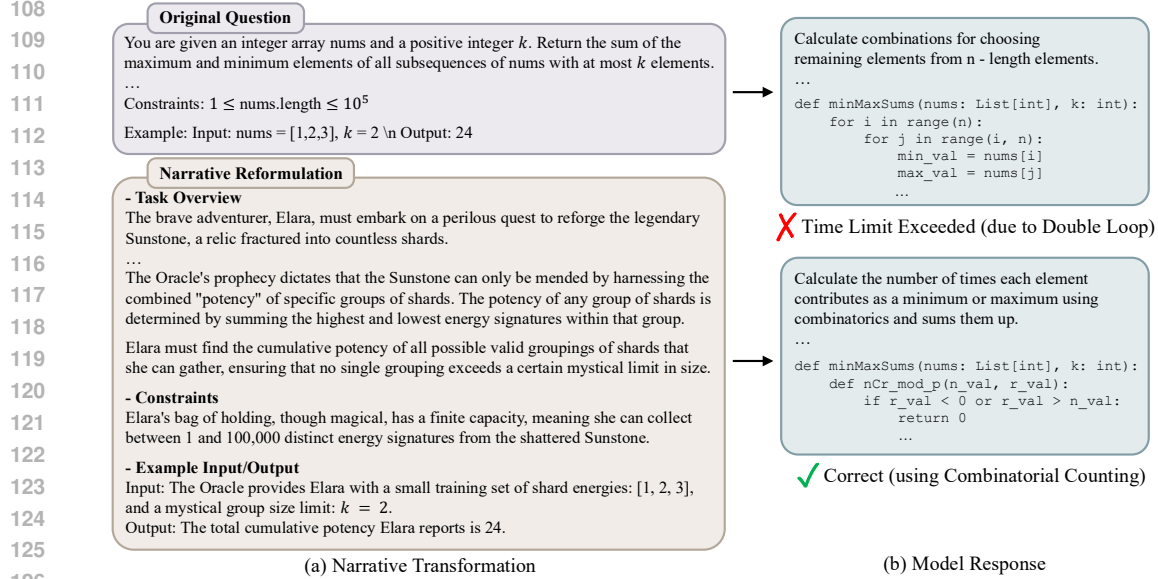


Figure 2: **Example of narrative reformulation.** The narrative representation bridges problem description and model reasoning, guiding the model from inefficient non-optimal solutions toward algorithmic strategies.

3 STORYCODER: REFORMULATING QUESTION INTO NARRATIVES

3.1 CONVENTIONAL BASELINES FOR CODE GENERATION

Early evaluations of code generation often used similarity-based metrics (e.g., CodeBLEU), but these failed to capture functional correctness. Recent benchmarks adopt execution-based evaluation, leading to the $\text{pass}@k$ metric (Chen et al., 2021). Within this setting, following baselines are considered: Repeated Sampling, Paraphrasing and Chain-of-Thought (CoT) prompting.

Repeated Sampling is the simplest way to improve $\text{pass}@k$ performance. The model is asked to generate multiple outputs for the same input, typically by stochastic sampling with high temperature. With more candidates in the pool, the chance of including at least one correct solution increases.

Paraphrasing is another simple baseline that reformulates the original problem statement by altering surface expressions without affecting its meaning (Zhou et al., 2024). This provides a variant of the prompts while leaving the fundamental representation of the task unchanged.

Chain-of-Thought (CoT) is another approach that encourages the model to structure its problem-solving process (Wei et al., 2022; Huang et al., 2023). Instead of directly producing code, the model is prompted to generate intermediate reasoning steps before the final output. This technique introduces explicit reasoning into the generation process, helping the model reflect on its problem-solving steps.

3.2 NARRATIVE REFORMULATION FOR DEEPER COMPREHENSION

The approaches discussed in Section 3.1 are useful, but they are limited in expanding the solution space, involve surface-level changes, or fail to reflect actual model reasoning (Turpin et al., 2023). To overcome these limitations, we propose a narrative reformulation framework that enables the model to deeply understand input representations. This framework helps the model understand the task through structured analogies and metaphors, and it is grounded in findings from cognitive science.

We construct a framework that reformulates code generation questions into a narrative format in three stages as shown in Figure 1: (i) for the i -th question Q_i , choosing an appropriate algorithmic category a_i and a narrative genre g_i ; (ii) rewriting Q_i as a structured narrative \mathcal{N}_i with three parts: task overview, constraints, and example input/output; and (iii) solving the task using \mathcal{N}_i . For each

problem Q_i , multiple narrative reformulations are generated by repeated sampling. Note that repeatedly sampled narratives differ from simply drawing multiple solutions, as each narrative provides a distinct perspective and plot that broadens the model’s representational space for interpreting and reasoning about the task. We denote the index of each reformulation variant by $j \in \{1, \dots, N\}$. Formally, the overall pipeline can be described as follows:

$$\{a_i^j, g_i^j, \mathcal{N}_i^j\}_{j=1}^N = f_{\text{narr}}(Q_i), \quad \text{Ans}(\mathcal{N}_i^j) = f_{\text{solve}}(\mathcal{N}_i^j), \quad (1)$$

where f_{narr} is the generator model that formulates narratives and f_{solve} is the solver model that generates $\text{Ans}(\cdot)$. Note that $\mathcal{N}_i^j \sim P(\cdot \mid a_i^j, g_i^j)$, since the narrative \mathcal{N}_i^j is conditioned by a_i^j and g_i^j chosen by f_{narr} .

In stage (ii), unlike the prior work (Sadiri Javadi et al., 2025), we carefully design the narrative components for programming tasks, where precise format and formal constraints are required. To ensure that reformulated problem preserve both narrative coherence and computational strictness, we divide a narrative \mathcal{N}_i^j into three parts:

- **Task Overview** (TO_i^j): presents the coding objective within a narrative frame, integrating scattered conditions into a coherent system that guides comprehension and reasoning.
- **Constraints** (C_i^j): reframes input ranges, time limits, and operational rules as natural restrictions in the story, allowing the model to internalize constraints within the narrative space.
- **Example Input/Output** (E_i^j): integrates sample test cases into contextual scenarios, aligning input/output examples with the story structure, so that bridging between concrete problem solving region and the narrative space.

This three-part structure is grounded in well-established findings from cognitive science. First, situating the problem in a **Task Overview** aligns with the theory of mental models, which emphasizes that humans comprehend and reason by constructing coherent representations of situations (Johnson-Laird, 1983). Second, articulating **Constraints** within the narrative makes the rules clear, so that readers do not have to think hard to integrate fragmented information. This reduces unnecessary mental effort and helps readers focus on solving the problem (Chandler & Sweller, 1991). Finally, integrating **Example Input/Output** into narrative situations takes advantages of the role of analogy in facilitating understanding and transfer, mapping abstract problem structures onto given scenarios (Gentner, 1983; Holyoak & Lu, 2021).

The transformation example is shown in Figure 2. Together, these three components of the narrative $\mathcal{N}_i^j = \{\text{TO}_i^j, \text{C}_i^j, \text{E}_i^j\}$ standardize the reformulation process, ensuring that all essential details of the original question of code generation are preserved while also allowing cognitive principles to be naturally integrated into the narrative structure.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETTINGS

Models. We evaluate a total of 11 models of varying sizes. Among open-source models, we use the instruction-tuned versions of Deepseek-Coder 6.7B (Guo et al., 2024), Deepseek-Coder-V2-Lite (Zhu et al., 2024), Llama-3.1 8B (Grattafiori et al., 2024), Gemma-2 9B and 27B (Team et al., 2024), Qwen-2.5-Coder 7B and 32B (Hui et al., 2024), and Mistral-Small 24B (AI, 2025). For closed-source models, we include Claude-3.5-Haiku (Anthropic, 2024), Gemini-2.5-Flash (Co-manici et al., 2025), and GPT-4.1-mini (OpenAI, 2025). All code generation is performed with a temperature setting of 0.2.

Dataset. We evaluate on three benchmarks: HumanEval (Chen et al., 2021), a hand-crafted dataset with function signatures and docstrings; LiveCodeBench (Jain et al., 2024), a large-scale dataset covering various programming problems from multiple platforms; and CodeForces (Mirzayanov, 2010), a collection of real-world algorithmic problems from competitive programming.

For HumanEval, we exclude questions that are invalid or containing incorrect sample input/output, resulting in a filtered set of 105 questions. For LiveCodeBench, we use the 175 questions from

Table 1: **Pass@10 performance** on three benchmarks. The upper part of the table reports closed-source models, while the lower part reports open-source models. Narrative prompting consistently improves performance over repeated sampling (RS) across all benchmarks and models. Pass@ k performance curves for all models and benchmarks are provided in Appendix A.1.

Model	HumanEval		LiveCodeBench		CodeForces	
	RS	Narrative	RS	Narrative	RS	Narrative
Gemini-2.5-Flash	96.19	96.19	49.71	57.14	45.48	65.97
GPT-4.1-mini	94.29	94.29	47.43	56.57	31.23	49.61
Claude-3.5-Haiku	85.71	94.29	33.71	38.29	46.45	50.51
Average	92.06	94.92	43.62	50.67	41.05	55.36
DSCoder 6.7B	82.86	90.48	22.29	27.43	13.12	18.01
DSCoder-V2-Lite	78.10	93.33	28.57	34.29	25.16	33.14
Llama-3.1 8B	79.05	81.90	21.14	27.43	8.11	19.08
Gemma-2 9B	63.81	82.86	20.00	26.29	11.69	21.39
Gemma-2 27B	76.19	87.62	27.43	34.29	22.38	30.97
Qwen-2.5-Coder 7B	89.52	93.33	26.86	33.14	19.31	26.74
Qwen-2.5-Coder 32B	92.38	94.29	30.86	40.00	15.08	27.10
Mistral-Small 24B	88.57	94.29	33.71	34.86	29.23	42.87
Average	81.31	89.76	26.36	32.22	18.01	27.41

release-v6 and confirm that all are sourced from AtCoder (Ueda & Inc., 2012) or LeetCode (Tang, 2015). For CodeForces, we apply filtering based on question length and difficulty, yielding a final set of 265 questions. For detailed filtering criteria, see the Appendix B.2.

Metric and Evaluation. We report the results using the pass@ k metric, which measures the probability that at least one out of k generated solutions is correct. Following HumanEval, we consider a generated solution to be correct only when it passes all test cases. We present results under a combined setting, referred to as Narrative, which merges the outcomes from two settings: the narrative-only version and the concatenated version where the narrative is paired with the original question, as illustrated in Figure 1. For a fair comparison, we ensure that the number of samples per question in Narrative matches that of baselines. Specifically, for the baselines, we sample twice as many outputs, ensuring that all methods use the same total number of samples n in Eq. 4.

4.2 EXPERIMENTAL RESULTS

Table 1 presents pass@10 results on three coding benchmarks. For closed-source models, we adopt a self-solving setting where $f_{\text{narr}} = f_{\text{solve}}$, while for open-source models, the narratives are generated by Gemini-2.5-Flash (f_{narr}) and solved by each open-source model (f_{solve}). Narrative prompting consistently outperforms the repeated sampling, demonstrating its general effectiveness for code generation. Improvements appear not only on HumanEval but also on more challenging benchmarks such as CodeForces and LiveCodeBench. Additionally, the pass@ k curves in Appendix A.1 show that narrative prompting outperforms the baseline consistently as k increases.

For open-source models, we initially considered evaluating them in a self-solving setting (i.e., $f_{\text{narr}} = f_{\text{solve}}$). However, as shown in Table 2, many models failed to reliably follow the required format, leading to a substantially smaller pool of valid narratives. To ensure consistent and valid narrative inputs, we use Qwen2.5-32B-Instruction as a narrative generator (f_{narr}) to evaluate the self-solving ability of open-source models, as it produces the highest proportion of valid narratives. Other open-source models are used as solvers (f_{solve}). Although this setup is not strict self-solving, this relaxed configuration offers a practical way to see whether narrative reformulation still improves performance with open-source narratives.

The results of pass@10 performance are shown in Table 3. On HumanEval, Qwen2.5-32B narratives (N-Q) consistently yield the best improvements across all solvers, whereas on LiveCodeBench and CodeForces, Gemini-2.5-Flash narratives (N-G) achieve the highest performance. These results suggest that open-source narratives can provide meaningful benefits, particularly on relatively easier benchmarks such as HumanEval, while closed-source narratives show clearer advantages on

Table 2: **Proportion of valid narratives** (i.e., without abnormal repetition or empty content) where each model follows the transformation guidelines properly. Note that for the DeepSeek and Qwen families, we used their base instruction-tuned versions (DeepSeek-V2-Lite-Chat (Zhu et al., 2024), Qwen2.5-7B/32B-Instruct (Yang et al., 2025)) rather than the variants further fine-tuned for coding tasks because the transformation tasks are closer to natural language tasks.

	DeepSeek	Gemma		Llama	Mistral	Qwen2.5	
Model	V2-Lite	27B	9B	3.1 8B	Small-24B	32B	7B
Valid (%)	68.11	76.59	51.74	36.66	86.90	95.96	37.80

Table 3: **Pass@10 performance** of open-source models on HumanEval, LiveCodeBench, and CodeForces. Results are reported for the repeated sampling (RS), narratives generated by Qwen2.5-32B (N-Q), and narratives generated by Gemini-2.5-Flash (N-G). Narrative reformulation improves performance, with open-source (Qwen) narratives showing stronger gains on easier benchmarks and closed-source (Gemini) narratives leading on the more challenging benchmarks.

Model	HumanEval			LiveCodeBench			CodeForces		
	RS	N-Q	N-G	RS	N-Q	N-G	RS	N-Q	N-G
DSCoder 6.7B	82.86	93.75	90.48	22.29	14.04	27.43	13.12	10.82	18.01
DSCoder V2 Lite	78.10	93.75	93.33	28.57	28.07	34.29	25.16	30.39	33.14
Llama 3.1 8B	79.05	90.62	81.90	21.14	19.30	27.43	8.11	14.55	19.08
Gemma 2 9B	63.81	90.62	82.86	20.00	14.04	26.29	11.69	18.40	21.39
Gemma 2 27B	76.19	90.62	87.62	27.43	17.54	34.29	22.38	27.82	30.97
Qwen 2.5 Coder 7B	89.52	96.88	93.33	26.86	22.81	33.14	19.31	24.33	26.74
Qwen 2.5 Coder 32B	92.38	96.88	94.29	30.86	35.09	40.00	15.08	23.17	27.10
Mistral-Small 24B	88.57	93.75	94.29	33.71	31.58	34.86	29.23	35.15	42.87
Average	81.31	93.36	89.76	26.36	22.81	32.22	18.01	23.08	27.41

more challenging benchmarks such as CodeForces and LiveCodeBench. This provides practical insight: open-source narratives such as Qwen2.5-32B may be sufficient for simpler questions, whereas closed-source narratives such as Gemini-2.5-Flash provide stronger support for more complex tasks. Additional experimental results on paraphrase and CoT are provided in Appendix A.2.

5 DISCUSSION

In experiments with code generation benchmarks, systematic performance analysis requires extracting and examining algorithmic sketches, which are critical components for achieving successful code generation. To this end, we design an experimental setup to extract and evaluate algorithms from model outputs. Given a code solution generated by the solver model f_{solve} , we query another model instance f_{alg} to identify the algorithm underlying the solution. This procedure is similar to back-translation in machine translation, where target–source language pairs are generated for self-training (Sennrich et al., 2016; Wang et al., 2025). We apply this process to both the original and narrative versions of each problem, enabling two key comparisons: between the back-translated algorithms from model outputs and the algorithms incorporated into the narratives a_i^j defined in the main experiments (Section 5.1); and between the back-translated algorithms of wrong solutions and the golden algorithms a_i^* derived from correct solutions (Section 5.2.) If not otherwise specified, experiments are conducted with Gemini-2.5-Flash.

5.1 NARRATIVES IMPROVE PROBLEM COVERAGE AND ALGORITHM AGREEMENT

We first compare the back-translated algorithms extracted from all model outputs with the algorithms incorporated into the narratives a_i^j produced in Section 4. We define coverage as the proportion of problems for which at least one correct solution is generated and agreement ratio as the fraction of correct solutions whose underlying algorithms match the algorithm that the model initially predicted when observing the original problem.

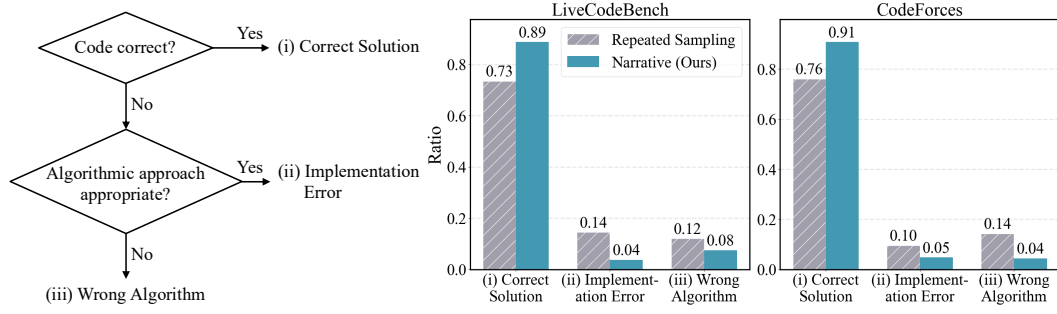


Figure 3: **Error decomposition of model outputs** into (i) correct algorithm choice, (ii) implementation errors, and (iii) incorrect algorithm choice under narrative vs. original prompts. Narrative prompts increase correct algorithm selection while reducing both implementation errors and misaligned choices.

Figure 4 shows that narrative reformulation improves both properties. Along the x-axis, the problem coverage increases on challenging benchmarks such as LiveCodeBench and CodeForces. This indicates that narratives expand the feasible solution space, especially on the harder tasks. Along the y-axis, the agreement ratio increases consistently across all benchmarks, demonstrating that the algorithm selected initially is faithfully reflected in the generated solutions. Together, these show that *narratives enable broader coverage and strengthen algorithmic consistency*, ensuring that the algorithm selection and incorporation step in Section 3 is reflected in the reformulated prompts.

5.2 QUANTIFYING THE CONTRIBUTION OF NARRATIVE PROMPTING

Solutions to code generation tasks can be decomposed into algorithms or sketch ideas, where selecting the right algorithm is necessary to arrive at a correct solution (Wang et al., 2025). An ideal code generation process can be summarized as a three-step pipeline: selecting the appropriate algorithm, implementing the algorithm in detail, and deriving the final correct solution. To analyze how models follow this process, we categorize their outputs into three outcomes as shown in Figure 3: (i) correct solutions, (ii) incorrect responses where the chosen algorithm is appropriate but the implementation is incorrect and leads to an error, and (iii) incorrect responses by selecting the wrong algorithm.

For step (ii) of the categorization, we automatically extract a golden algorithm using the model itself. Specifically, we take all generated code solutions confirmed to be correct, from either the original or narrative problems, and query f_{alg} to identify which algorithm the solutions use. We then determine the golden algorithm a_i^* by majority voting among the candidates:

$$a_i^* = \text{MajorityVote}\left(\left\{f_{\text{alg}}(\text{Ans}(X_i)) \mid X_i \in Q_i \cup \{\mathcal{N}_i^j\}_{j=1}^N, \text{Ans}(X_i) \text{ is correct}\right\}\right). \quad (2)$$

The a_i^* is used in step (ii) to evaluate whether incorrect code solutions nevertheless adopt the correct algorithm, with such cases classified as implementation errors (e.g., inefficient search loops, misused data structures). To better isolate the effect of representation change, we exclude trivial cases where generations for both original and narrative are either all correct or all incorrect. Such problems are too easy or too difficult to provide meaningful patterns.

Figure 3 presents the results using Gemini-2.5-Flash as the narrative generator f_{narr} , the solver f_{solve} , and the back-translator f_{alg} . In both benchmarks, narratives increase the proportion of correct solutions while reducing both implementation errors and incorrect algorithm choices. Notably, the increase in (i) correct solutions and the drop in (ii) implementation errors demonstrate our key claim:

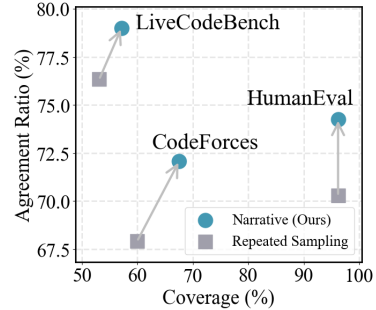


Figure 4: **Effect of narrative reformulation.** The x-axis denotes coverage, the fraction of problems with at least one correct solution; and the y-axis shows the agreement ratio, the proportion of correct solutions consistent with the initial chosen algorithm, a_i .

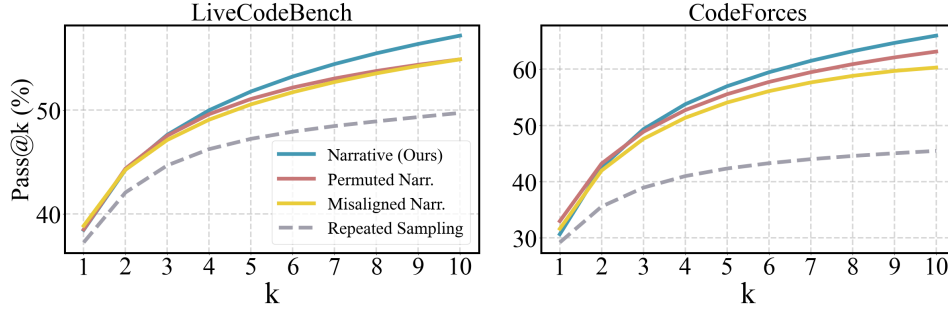


Figure 5: **Comparison of pass@k curves across different prompt settings.** Permuted narratives (components mixed across variants) outperform original prompts but remain below complete narratives, indicating the importance of coherence (Section 5.3.) Misaligned narratives (genres forced from incongruent sets) degrade performance compared to complete narratives, showing that proper representation contributes to effective problem solving (Section 5.4.)

narratives encourage integrative reasoning, helping models not only move toward correct solutions but also generate more reliable implementations through their structured and stepwise design.

5.3 COHERENCE MATTERS IN NARRATIVE REFORMULATION

It is important to quantitatively evaluate the contribution of analogical expressions and compositions because they represent fundamental elements of narrative design (Sadiri Javadi et al., 2025). However, the three components introduced in Section 3—Task Overview (TO_i^j), Constraints (C_i^j), and Example Input/Output (E_i^j)—are all essential for valid code generation. Therefore, instead of conducting ablation studies that remove individual components, we design experiments that permute different versions of these components. This setup allows us to distinguish between the informational benefits inherent to narratives and the additional gains achieved when their elements are coherently integrated.

Recall that each narrative reformulation \mathcal{N}_i^j for question Q_i —the j -th variant of the i -th problem, obtained by repeated sampling from f_{narr} —consists of three parts: $\mathcal{N}_i^j = \{TO_i^j, C_i^j, E_i^j\}$. In the permuted setting, we construct a new narrative $\tilde{\mathcal{N}}_i^{j_1, j_2, j_3}$ by sampling these components from different variants:

$$\tilde{\mathcal{N}}_i^{j_1, j_2, j_3} = \{TO_i^{j_1}, C_i^{j_2}, E_i^{j_3}\}, \quad \text{where } j_1 \neq j_2, j_1 \neq j_3, j_2 \neq j_3. \quad (3)$$

This setup allows us to compare three conditions: **Original** (Q_i), without narrative reformulation; **Complete Narrative** (\mathcal{N}_i^j), all components come from the same variant j ; and **Permuted Narrative** ($\tilde{\mathcal{N}}_i^{j_1, j_2, j_3}$), each component is randomly drawn from a distinct variant. The results in Figure 5 show differences in the three conditions. Surprisingly, the permuted narratives still outperform the original prompts in all k , showing that narrative reformulation itself provides informational benefits even when task overviews, constraints, and examples are drawn from different variants. However, we should note that their performance still falls short of the complete narratives, suggesting that *the biggest improvements can be achieved when all narrative components fit together and the task description works as a single coherent structure.*

5.4 LLMs’ INHERENT RECOGNITION OF OPTIMAL NARRATIVE SPACE

Information is understood differently depending on its form, style, or framing (Thibodeau & Boroditsky, 2011; Gentner, 1983). In our setting, we identify the narrative genre as a primary factor that shapes the overall style and structure of the problem descriptions. To analyze how narrative expression affects interpretation, we deliberately replace well-aligned genres with incongruent ones and observe how models respond to these changes. Figure 6 shows the distribution of optimal genres selected by the three closed-source models in Section 4 across all coding benchmarks. Gemini-2.5-

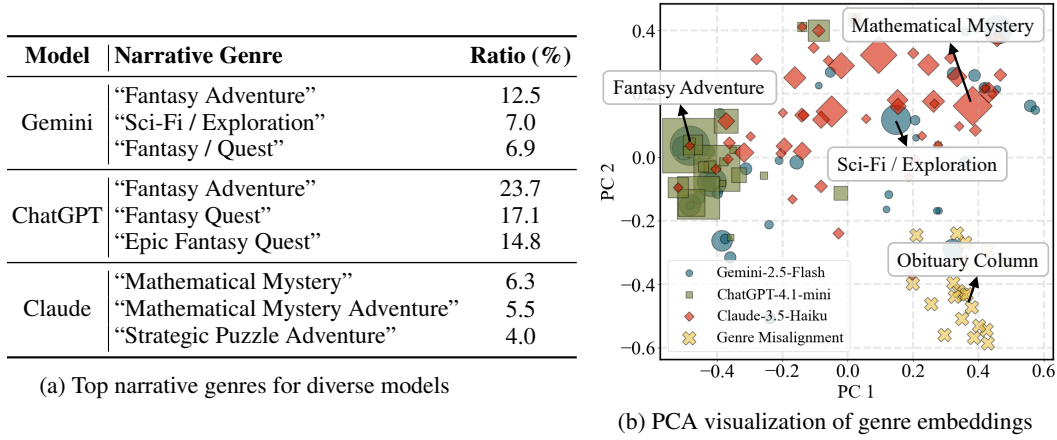


Figure 6: **Top chosen narrative genres that appear at least five times.** (a) The detailed genres with their ratio for each model; (b) PCA visualization of text embeddings of genre names selected by each model, extracted using all-MiniLM-L6-v2 (Wang et al., 2020). Point sizes in (b) are proportional to frequency. The plot shows that the models show varying preferences over genres when reformulating coding benchmark problems into narratives.

Flash and ChatGPT-4.1-mini mainly choose genres such as “Fantasy Adventure,” whereas Claude-3.5-Haiku favors “Mathematical Mystery.” This suggests that each model identifies different representations as optimal and may require different strategies for effective problem-solving.

Meanwhile, we manually constructed a set of misaligned genres, \mathcal{G}_{mis} , a list of 20 genres with administrative, legal, or memorial characteristics, which are disjoint from such optimal and descriptive genres; details are provided in Appendix B.4. Formally, we extend Eq. 1 by explicitly specifying the genre variable g_i^j . In the standard setting, g_i^j is selected naturally by f_{narr} based on the question Q_i . However, in the misalignment setting, we enforce the genre to be drawn from a predefined set of misaligned genres, $g_{\text{mis}} \sim \mathcal{G}_{\text{mis}}$. With this procedure, we obtain the **Misaligned Narratives**, $\mathcal{N}_i^{j,\text{mis}} \sim P(\cdot \mid a_i^j, g_{\text{mis}})$.

The Misaligned Narrative line in Figure 5 shows the pass@ k curve results of this experiment. In both benchmarks, $\mathcal{N}_i^{j,\text{mis}}$ show reduced performance compared to \mathcal{N}_i^j . This indicates that not all narratives are equally effective. LLMs implicitly recognize suitable genres within the narrative space and perform best when the prompts align with the optimal representation. These findings suggest that the benefit of narrative reformulation goes beyond surface-level prompting. Genre is a key element of the overall style and structure of narrative; by showing that genre alignment impacts problem-solving performance, *narratives emerge not merely as stylistic variations, but as cognitive tools that structure problems in ways conducive to problem solving.*

6 CONCLUSION

In this work, we proposed STORYCODER, a framework that reformulates coding problems into coherent narratives to promote integrative reasoning in LLMs, showing consistent performance gains across diverse benchmarks. Beyond demonstrating improved coverage and algorithmic alignment, our findings suggest that narrative coherence and representation alignment are key factors that shape problem-solving effectiveness. More broadly, our study shows the role of narratives as guiding frameworks that help organize and contextualize complex tasks. We expect that future work will explore adaptive genre selection, automated narrative refinement, and the extension of narrative-based prompting to domains such as mathematics, multimodal reasoning, and scientific discovery.

Limitations. Despite our performance gains, our approach is limited by its reliance on the quality of generated narratives, which can vary substantially across models. In addition, narrative reformulation naturally increases prompt length, which may require careful consideration of efficiency in large-scale applications.

REFERENCES

- Mistral AI. Mistral-small-24b-instruct-2501. <https://huggingface.co/mistralai/Mistral-Small-24B-Instruct-2501>, 2025. Instruction-tuned 24B parameter language model, available at Hugging Face. 4
- Anthropic. Claude 3.5 haiku. <https://www.anthropic.com/news/claude-3-5-sonnet>, 2024. Instruction-tuned large language model, available at Anthropic and Vertex AI. 4
- Paul Chandler and John Sweller. Cognitive load theory and the format of instruction. *Cognition and instruction*, 8(4):293–332, 1991. 4
- 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. 3, 4, 13, 15
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025. 4
- Junbo Fu, Guoshuai Zhao, Yimin Deng, Yunqi Mi, and Xueming Qian. Learning to paraphrase for alignment with llm preference. In *EMNLP (Findings)*, pp. 2394–2407, 2024. URL <https://aclanthology.org/2024.findings-emnlp.134>. 2
- Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive science*, 7(2): 155–170, 1983. 1, 4, 8
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. 4
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024. 4
- Keith J Holyoak and Hongjing Lu. Emergence of relational reasoning. *Current Opinion in Behavioral Sciences*, 37:118–124, 2021. 1, 4
- Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. Codecot: Tackling code syntax errors in cot reasoning for code generation. *arXiv preprint arXiv:2308.08784*, 2023. 2, 3
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024. 4
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024. 4, 13, 15
- Philip Nicholas Johnson-Laird. *Mental models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983. 1, 4
- Andrea Seaton Kelton, Robin R Pennington, and Brad M Tuttle. The effects of information presentation format on judgment and decision making: A review of the information systems research. *Journal of Information Systems*, 24(2):79–105, 2010. 1
- Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation. *arXiv preprint arXiv:2505.06120*, 2025. 1
- 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. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=vYhg1xSj8j>. 2

- Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23, 2025a. 2
- Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. CodeIO: Condensing reasoning patterns via code input-output prediction. In *Forty-second International Conference on Machine Learning*, 2025b. URL <https://openreview.net/forum?id=feIaF6vYfL>. 2
- Jonathan Light, Yue Wu, Yiyu Sun, Wenchao Yu, Yanchi Liu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, and Wei Cheng. SFS: Smarter code space search improves LLM inference scaling. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=MCHuGOkExF>. 1
- Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Juntao Pan, Mingjie Zhan, and Hongsheng Li. Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=1Iuw1jcIrf>. 2
- Mikhail Mirzayanov. Codeforces. <https://codeforces.com/>, 2010. Online competitive programming platform. 4, 13, 15
- OpenAI. Gpt-4.1 mini. <https://platform.openai.com/docs/models/gpt-4.1-mini>, 2025. Instruction-tuned language model, available at OpenAI API. 4
- Vahid Sadiri Javadi, Johanne Trippas, Yash Kumar Lal, and Lucie Flek. Can stories help LLMs reason? curating information space through narrative. In Giulia Rambelli, Filip Ilievski, Marianna Bolognesi, and Pia Sommerauer (eds.), *Proceedings of the 2nd Workshop on Analogical Abstraction in Cognition, Perception, and Language (Analogy-Angle II)*, pp. 92–107, Vienna, Austria, August 2025. Association for Computational Linguistics. ISBN 979-8-89176-274-9. doi: 10.18653/v1/2025.analogyangle-1.8. URL <https://aclanthology.org/2025.analogyangle-1.8/>. 4, 8
- Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In Katrin Erk and Noah A. Smith (eds.), *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 86–96, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1009. URL <https://aclanthology.org/P16-1009/>. 6
- Winston Tang. Leetcode. <https://leetcode.com/>, 2015. Online coding interview preparation platform. 5, 15
- Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024. 4
- Paul H Thibodeau and Lera Boroditsky. Metaphors we think with: The role of metaphor in reasoning. *PloS one*, 6(2):e16782, 2011. 8
- Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. Language models don’t always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL <https://openreview.net/forum?id=bzs4uPLXvi>. 3
- Takahiro Ueda and AtCoder Inc. Atcoder. <https://atcoder.jp/>, 2012. Online competitive programming platform. 5, 15
- Iris Vessey. Cognitive fit: A theory-based analysis of the graphs versus tables literature. *Decision sciences*, 22(2):219–240, 1991. 1
- Evan Z Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean M. Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves LLM search for code generation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=48WAZhwHHw>. 1, 6, 7

- Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems*, 33:5776–5788, 2020. 9
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL <https://openreview.net/forum?id=1PL1NIMMrw>. 2
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J. 2, 3
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. 6
- Yue Zhou, Yada Zhu, Diego Antognini, Yoon Kim, and Yang Zhang. Paraphrase and solve: Exploring and exploiting the impact of surface form on mathematical reasoning in large language models. *arXiv preprint arXiv:2404.11500*, 2024. 2, 3
- Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024. 4, 6

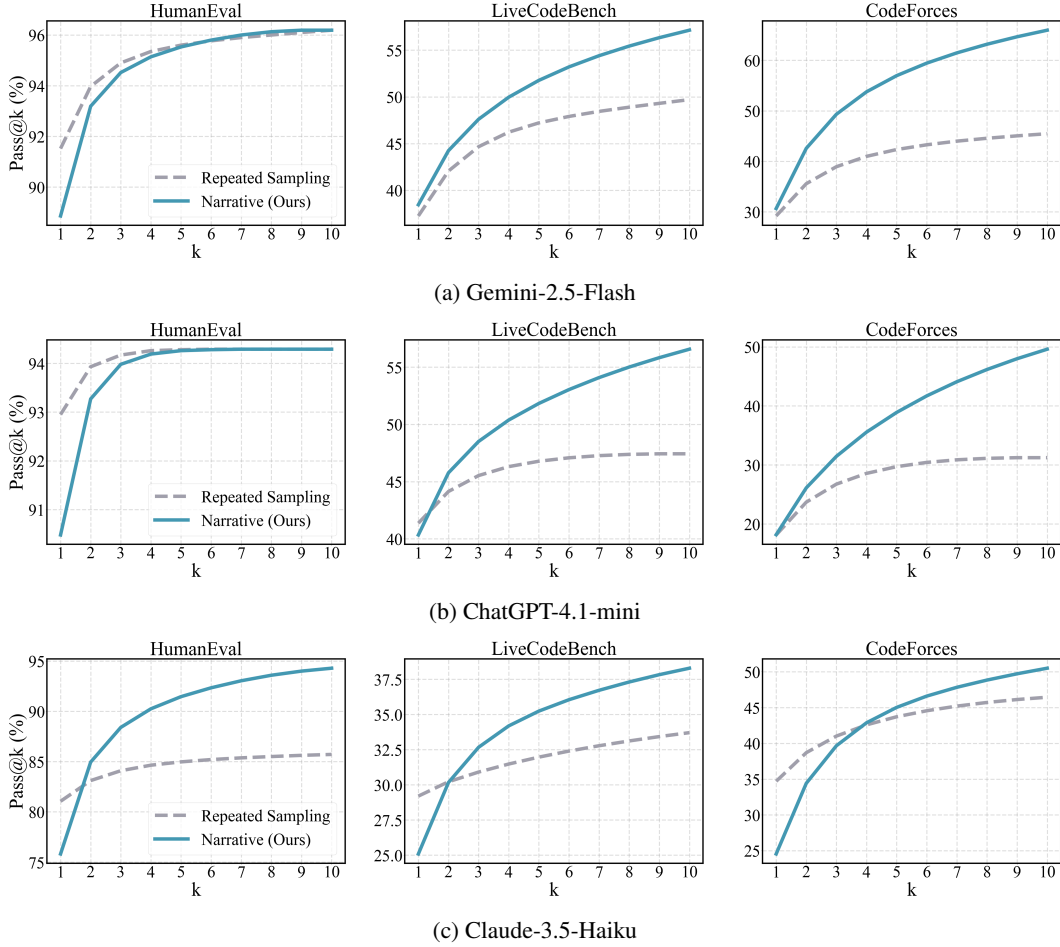


Figure A.1: **Pass@k performance** of closed-source models for $k = 1, \dots, 10$. Across all models and benchmarks, narrative prompting consistently outperforms the baseline as k increases.

Appendix

A ADDITIONAL RESULTS

A.1 PASS@K CURVES ON THREE BENCHMARKS

Figures A.1 and A.2 present the pass@k curves on HumanEval (Chen et al., 2021), LiveCodeBench (Jain et al., 2024), and CodeForces (Mirzayanov, 2010) for closed-source and open-source models. Except for small values of k (around $k = 1$ to 4), narrative prompting outperforms the baseline (Repeated Sampling) in all cases. As k increases, the performance gains become smaller on the easier HumanEval benchmark, while they continue to grow on the more challenging LiveCodeBench and CodeForces benchmarks.

A.2 COMPREHENSIVE RESULTS

Table A.1 reports additional results when GPT-4.1-mini or Claude-3.5-Haiku are used as f_{narr} instead of Gemini-2.5-Flash. Similar to the main results in Table 1, we observe consistent improvements over the Repeated Sampling (RS) prompts across open-source solvers, showing that the generalization of narrative reformulation is not limited to specific model but extends to other closed-source generators as well.

Table A.1: **Pass@10 performance** of open-source solvers (f_{solve}) using repeated sampling (RS), narratives generated by GPT-4.1-mini (G), and Claude-3.5-Haiku (C) as f_{narr} . While the degree of improvement varies across models, performance consistently improves even when f_{narr} is a closed-source model other than Gemini-2.5-Flash, demonstrating the generalization of narrative reformulation across closed- and open-source models.

Model	HumanEval			LiveCodeBench			CodeForces		
	RS	G	C	RS	G	C	RS	G	C
DSCoder 6.7B	82.86	85.71	86.67	22.29	25.71	22.29	13.12	13.40	11.38
DSCoder V2 Lite	78.10	87.62	90.48	28.57	28.57	32.00	25.16	26.03	30.05
Llama 3.1 8B	79.05	81.90	76.19	21.14	25.14	24.00	8.11	13.70	14.33
Gemma 2 9B	63.81	67.62	83.81	20.00	22.86	22.86	11.69	16.06	18.93
Gemma 2 27B	76.19	80.00	87.62	27.43	28.00	29.14	22.38	27.18	27.24
Qwen 2.5 Coder 7B	89.52	88.57	94.29	26.86	29.71	29.14	19.31	20.61	22.35
Qwen 2.5 Coder 32B	92.38	88.57	94.29	30.86	34.29	37.14	15.08	20.61	22.81
Mistral Small 24B	88.57	87.62	89.52	33.71	34.86	34.29	29.23	36.09	33.75
Average	81.31	83.45	87.86	26.36	28.64	28.86	18.01	21.71	22.61

Table A.2: **Pass@10 performance** comparison of Paraphrase (Para.), Chain-of-Thought (CoT), and Narrative (Narr.) prompts with open-source models on three benchmarks. Paraphrase and narrative prompts are generated by Gemini-2.5-Flash. Compared to simple paraphrasing, which only alters surface expressions, or CoT, which does not fundamentally change the representation, our STORYCODER effectively supports the model reasoning and expands the solution space.

Model	HumanEval			LiveCodeBench			CodeForces		
	Para.	CoT	Narr.	Para.	CoT	Narr.	Para.	CoT	Narr.
DSCoder 6.7B	81.90	83.81	90.48	24.00	22.86	27.43	11.67	11.69	18.01
DSCoder V2 Lite	85.71	80.95	93.33	28.57	29.14	34.29	26.40	26.28	33.14
Llama 3.1 8B	83.81	76.19	81.90	22.86	24.57	27.43	7.62	10.92	19.08
Gemma 2 9B	68.57	62.86	82.86	20.57	19.43	26.29	13.40	10.74	21.39
Gemma 2 27B	81.90	80.95	87.62	27.43	25.71	34.29	22.49	22.38	30.97
Qwen 2.5 Coder 7B	88.57	92.38	93.33	26.86	29.71	33.14	23.28	20.58	26.74
Qwen 2.5 Coder 32B	92.38	90.48	94.29	34.29	34.29	40.00	19.63	24.98	27.10
Mistral Small 24B	87.62	90.48	94.29	33.14	34.86	34.86	29.53	27.01	42.87
Average	83.81	82.26	89.76	27.22	27.57	32.22	19.25	19.32	27.41

Table A.2 reports the pass@10 performance of open-source models under three prompting strategies: Paraphrasing, Chain-of-Thought (CoT), and Narrative. While paraphrasing merely changes surface expressions and CoT adds intermediate steps without altering the representation itself, our proposed STORYCODER provides coherent narratives that better support reasoning and broaden the solution space, leading to consistent improvements across benchmarks. The setup of Table A.3 is the same as that of Table A.2, except that the paraphrase prompts are sampled five times. Similar to the results in Table 1, our method achieves higher performance on more challenging benchmarks.

B EXPERIMENTAL DETAILS

B.1 EVALUATION METRIC

The pass@ k metric evaluates the probability that at least one correct solution is obtained among k independently sampled outputs. Formally, given n generated outputs with c correct ones, the expected success rate is

$$\text{pass}@k = \mathbb{E} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]. \quad (4)$$

Table A.3: **Pass@10 performance** comparison of Paraphrase (Para., repeatedly sampled five times), Chain-of-Thought (CoT), and Narrative (Narr.) prompts with open-source models on three benchmarks. Consistent with the main results, STORYCODER performs better on the more challenging benchmarks.

Model	HumanEval			LiveCodeBench			CodeForces		
	Para.	CoT	Narr.	Para.	CoT	Narr.	Para.	CoT	Narr.
DSCoder 6.7B	90.48	83.81	90.48	27.43	22.86	27.43	15.11	11.69	18.01
DSCoder V2 Lite	95.24	80.95	93.33	33.14	29.14	34.29	33.49	26.28	33.14
Llama 3.1 8B	86.67	76.19	81.90	27.43	24.57	27.43	12.34	10.92	19.08
Gemma 2 9B	82.86	62.86	82.86	24.57	19.43	26.29	20.18	10.74	21.39
Gemma 2 27B	88.57	80.95	87.62	30.86	25.71	34.29	30.66	22.38	30.97
Qwen 2.5 Coder 7B	95.24	92.38	93.33	32.00	29.71	33.14	27.67	20.58	26.74
Qwen 2.5 Coder 32B	96.19	90.48	94.29	38.29	34.29	40.00	24.52	24.98	27.10
Mistral Small 24B	92.38	90.48	94.29	34.29	34.86	34.86	38.47	27.01	42.87
Average	90.95	82.26	89.76	31.00	27.57	32.22	25.31	19.32	27.41

In code generation benchmarks, there is no single canonical solution, and multiple programs may be valid for the same problem. As a result, evaluation is typically performed by repeated sampling and checking whether at least one passes all test cases. This characteristic naturally motivates the use of $\text{pass}@k$, which measures the probability that a model produces at least one correct solution within k attempts. Consequently, $\text{pass}@k$ has become an intuitive and widely adopted metric in practice.

B.2 DATASET FILTERING

For reproducibility, we summarize the dataset filtering process applied to each benchmark.

HumanEval (Chen et al., 2021).¹ We exclude samples without input/output examples and standardize the format of all examples. Function names in signatures and examples are unified when they differ. After these adjustments, we obtain a filtered set of 105 samples.

LiveCodeBench (Jain et al., 2024).² To avoid data contamination, we use the release-v6 subset that covers samples from January to April 2025. This subset consists of 112 samples from AtCoder (Ueda & Inc., 2012) and 63 from LeetCode (Tang, 2015), amounting to 175 samples.

CodeForces (Mirzayanov, 2010).³ We exclude problems without input/output examples, select samples with moderate text length ($\text{length} \leq 1000$), and select intermediate- and advanced-level tasks ($\text{rating} \geq 2000$). This results in a final set of 265 samples.

B.3 NARRATIVE TRANSFORMATION EXAMPLES

Figure B.1 shows the prompts for converting original coding questions into narrative format. Figure B.2 presents a complete example of a narrative reformulation and the corresponding response generated by Gemini-2.5-Flash for a sample from LiveCodeBench.

B.4 MISALIGNED GENRES

To construct the misalignment setting, we curated a set of genres that are intentionally incongruent with problem descriptions. These genres were selected to represent contexts that are stylistically or semantically distant from typical programming tasks, ensuring that the resulting narratives do not naturally align with the problem’s intent. Table B.1 presents the complete list of misaligned genres, grouped into four categories.

¹<https://github.com/openai/human-eval>

²<https://github.com/LiveCodeBench/LiveCodeBench>

³<https://huggingface.co/datasets/open-r1/codeforces>

Table B.1: Complete list of misaligned genres grouped into four categories.

Category	Misaligned Genres
Practical / Administrative Documents	Hospital Intake Form; Medical Prescription Form; Personal Information Consent Form; Insurance Claim Form; Visa Application Form; Tax Return Form
Legal / Public Records	Court Transcript of an Extortion Case; Heavy Machinery Operator License; Military Service Exemption Certificate; Divorce Decree; Bank Loan Agreement
Industrial / Media Contexts	Billboard Advertisement for a Toothbrush; Radio Weather Forecast; Model Agency Contract
Funerary / Ritual Records	Funeral Service Program; Memorial Tribute Writing; Obituary Column; Eulogy; Gravestone Inscription; Condolence Letter

C THE USE OF LLMs

We used LLMs only for minor language editing, including adjustments to word choices and clarity. LLMs were not used for the research design, analysis, interpretation, or manuscript preparation.

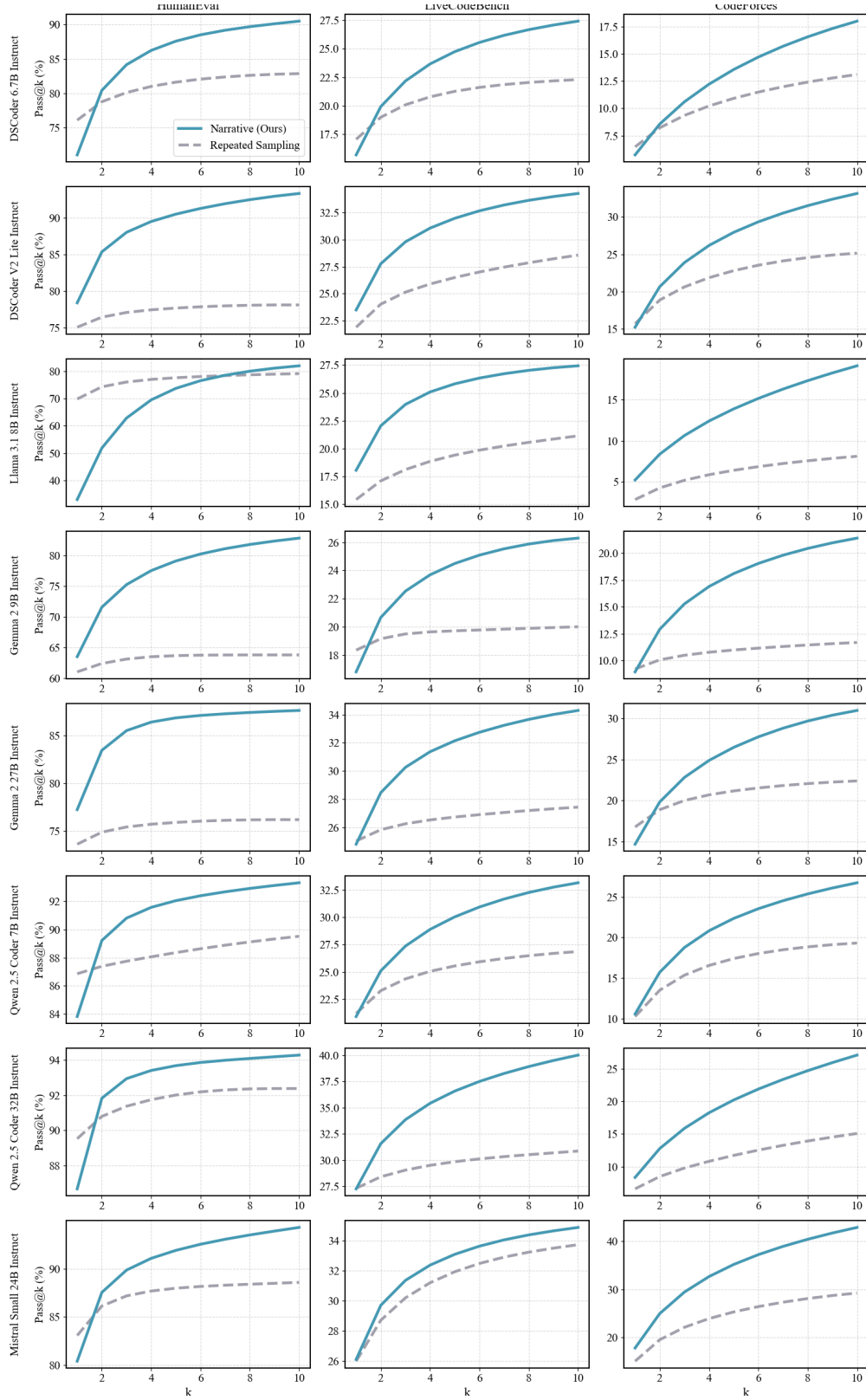


Figure A.2: **Pass@k performance** of open-source models for $k = 1, \dots, 10$. Across all models and benchmarks, narrative prompting consistently outperforms the baseline as k increases.

918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971

Narrative Transformation Guidelines

Please transform the coding problem into a narrative story using the following guidelines.

Guidelines for Narrative Conversion:

Before writing the narrative, you must complete two preliminary steps:

1. Review the major categories of coding test algorithms:
 - Graph Algorithms
 - Dynamic Programming
 - Greedy Algorithms
 - Sorting and Searching
 - String Algorithms
 - Data Structures
 - Mathematics and Number Theory
 - Simulation and Implementation
2. Decide which algorithm category the given problem most closely belongs to.
Then, select a **narrative genre** that naturally aligns with the chosen algorithm.

Output Format:

You must write the output in the **exact following order** with the specified headers:

- Algorithm Category: (one of the categories above)
- Narrative Genre: (a fitting genre of your choice)
- Task Overview: Describe the background and objective of the problem in a clear, narrative-inspired manner. The selected algorithm should be introduced naturally here, with its logic explained as part of the setting or scenario.
- Constraints: State input sizes, value ranges, conditions, and key operational rules. If efficiency or time limits exist, express them as natural constraints. The chosen algorithm should also shape these rules.
- Example Input/Output: Reframe the examples as part of the scenario's flow. Present them as clear, contextual situations.

The narrative must include all essential parts of the original problem, ensuring no constraints, goals, or examples are omitted. Do not include any other text outside these five sections.
****Do not attempt to solve the problem or provide any code. Your task is only to transform the problem statement into the narrative format as specified.****

The coding problem is as follows:

{Coding Problem}

Figure B.1: Instructions for converting a question of code generation benchmarks into narrative format.

Original Question

You are given an $m \times n$ grid. A robot starts at the top-left corner of the grid $(0, 0)$ and wants to reach the bottom-right corner $(m-1, n-1)$. The robot can move either right or down at any point in time.

The grid contains a value $\text{coins}[i][j]$ in each cell:

If $\text{coins}[i][j] > 0$, the robot gains that many coins.

If $\text{coins}[i][j] < 0$, the robot encounters a robber, and the robber steals the absolute value of $\text{coins}[i][j]$ coins.

The robot has a special ability to neutralize robbers in at most 2 cells on its path, preventing them from stealing coins in those cells.

Note: The robot's total coins can be negative.
Return the maximum profit the robot can gain on the route.

Example 1:

Input: $\text{coins} = [[0,1,-1],[1,-2,3],[2,-3,4]]$ \n Output: 8

Explanation: \n An optimal path for maximum coins is:

Start at $(0, 0)$ with 0 coins (total coins = 0).

Move to $(0, 1)$, gaining 1 coin (total coins = $0 + 1 = 1$).

Move to $(1, 1)$, where there's a robber stealing 2 coins. The robot uses one neutralization here, avoiding the robbery (total coins = 1).

Move to $(1, 2)$, gaining 3 coins (total coins = $1 + 3 = 4$).

Move to $(2, 2)$, gaining 4 coins (total coins = $4 + 4 = 8$).

Example 2:

Input: $\text{coins} = [[10,10,10],[10,10,10]]$ \n Output: 40

Explanation: \n An optimal path for maximum coins is:

Start at $(0, 0)$ with 10 coins (total coins = 10).

Move to $(0, 1)$, gaining 10 coins (total coins = $10 + 10 = 20$).

Move to $(0, 2)$, gaining another 10 coins (total coins = $20 + 10 = 30$).

Move to $(1, 2)$, gaining the final 10 coins (total coins = $30 + 10 = 40$).

Constraints:

$m == \text{coins.length}$ \n $n == \text{coins}[i].length$ \n $1 \leq m, n \leq 500$ \n $-1000 \leq \text{coins}[i][j] \leq 1000$

Narrative Reformulation

Task Overview:

In the year 2342, Earth lies fragmented, its surface scarred by the Great Cataclysm. Humanity survives in isolated pockets, relying on ancient, automated mining units to retrieve precious resources from the irradiated ruins. You are tasked with programming the "Pathfinder Unit 7," a sophisticated autonomous robot designed for high-risk recovery missions. Its current assignment: navigate the treacherous Grid Sector Alpha-9. This sector is a vast, decaying urban labyrinth, represented as an ' $m \times n$ ' grid. Each cell within this grid may contain valuable "Aetherium Crystals" (positive coin values) or be infested by "Scavenger Bots" (negative coin values). Aetherium Crystals boost the Pathfinder's energy reserves, while Scavenger Bots drain them by an equivalent amount. The Pathfinder must journey from its deployment point, the top-left corner $(0, 0)$ of the sector, to the designated extraction zone at the bottom-right corner $(m-1, n-1)$. Due to its robust maneuvering system, the Pathfinder can only move directly "South" or "East" from its current position. A critical feature of Pathfinder Unit 7 is its limited "Disruptor Field" capability, which can neutralize the energy-draining effects of up to two Scavenger Bots encountered along its path. Your mission, should you choose to accept it, is to chart a course for Pathfinder Unit 7 that maximizes its net Aetherium gain upon reaching the extraction zone. The core challenge is to strategically utilize the Disruptor Fields to avoid costly encounters, effectively making optimal choices at each step, a classic dynamic programming puzzle.

Constraints:

The Grid Sector Alpha-9 is quite expansive, with its dimensions ' m ' rows and ' n ' columns ranging from '1' to '500' units. The Aetherium Crystal deposits or Scavenger Bot energy drains in any given cell ' $[i][j]$ ' can vary widely, from a drain of '1000' units to a gain of '1000' units. Even after utilizing its Disruptor Fields, Pathfinder Unit 7's total Aetherium balance can dip into the negative, indicating a significant energy debt. The mission's success is measured purely by the final net Aetherium count.

Example Input/Output:

Example 1:

Pathfinder Unit 7's Aetherium Scanner reports the following for a small 3×3 section of Grid Sector Alpha-9:

Input (Aetherium Map): \n $[[0,1,-1],[1,-2,3],[2,-3,4]]$

After careful computation, the optimal trajectory for Pathfinder Unit 7, maximizing its Aetherium gain, is identified.

Output (Max Aetherium Gain): 8

Detailed Trajectory Analysis:

- Deployment at $(0, 0)$: Initial energy reserves are 0 Aetherium.
- Move to $(0, 1)$: Aetherium gain of 1. Total reserves: 1.
- Move to $(1, 1)$: A Scavenger Bot is detected, threatening to drain 2 Aetherium. Pathfinder Unit 7 activates one Disruptor Field, neutralizing the threat. Total reserves remain: 1.
- Move to $(1, 2)$: Aetherium gain of 3. Total reserves: 4.
- Move to $(2, 2)$: Aetherium gain of 4. Total reserves: 8.

Extraction successful with 8 Aetherium.

Example 2:

Another mission involves a smaller, 2×3 Grid Sector Beta-7, known for its rich, unmolested Aetherium veins.

Input (Aetherium Map): \n $[[10,10,10],[10,10,10]]$

The computed optimal path for Pathfinder Unit 7 in this sector yields a high Aetherium return.

Output (Max Aetherium Gain): 40

Detailed Trajectory Analysis:

- Deployment at $(0, 0)$: Initial energy reserves are 10 Aetherium.
- Move to $(0, 1)$: Aetherium gain of 10. Total reserves: 20.
- Move to $(0, 2)$: Aetherium gain of 10. Total reserves: 30.
- Move to $(1, 2)$: Aetherium gain of 10. Total reserves: 40.

Extraction successful with 40 Aetherium. No Scavenger Bots were encountered, so Disruptor Fields remained unused.

(a) Narrative Transformation

import math

class Solution:

def maximumAmount(self, coins: List[List[int]]) -> int:

m = len(coins)

n = len(coins[0])

dp = {}

def solve(r, c, k):

if r == m - 1 and c == n - 1:

return coins[r][c] if coins[r][c] >= 0 or k > 0 else coins[r][c]

if (r, c, k) in dp:

return dp[(r, c, k)]

current_coins = coins[r][c]

val_with_neutralize = -math.inf

if current_coins < 0 and k > 0:

val_with_neutralize = current_coins * -1

if r + 1 < m:

val_with_neutralize = max(

val_with_neutralize,

current_coins * -1 + solve(r + 1, c, k - 1)

)

if c + 1 < n:

val_with_neutralize = max(

val_with_neutralize,

current_coins * -1 + solve(r, c + 1, k - 1)

)

val_without_neutralize = current_coins

if r + 1 < m:

val_without_neutralize = max(

val_without_neutralize,

current_coins + solve(r + 1, c, k)

)

if c + 1 < n:

val_without_neutralize = max(

val_without_neutralize,

current_coins + solve(r, c + 1, k)

)

if current_coins >= 0:

dp[(r, c, k)] = val_without_neutralize

else:

dp[(r, c, k)] = max(val_with_neutralize, val_without_neutralize)

return dp[(r, c, k)]

return solve(0, 0, 2)

✗ (Incorrect)

import collections

class Solution:

def maximumAmount(self, coins: List[List[int]]) -> int:

m = len(coins)

n = len(coins[0])

dp[r][c][k] stores the maximum Aetherium gain to reach (r, c)

using k disruptor fields.

k can be 0, 1, or 2.

dp = {}

Initialize dp table with a very small number

Using a dictionary to store only reachable states

dp[(r, c, k)] = value

Base case: starting point (0, 0)

If coins[0][0] is negative, we might use a disruptor field.

if coins[0][0] < 0:

dp[(0, 0, 1)] = 0 # Used 1 disruptor field

dp[(0, 0, 0)] = coins[0][0] # Did not use disruptor field

else:

dp[(0, 0, 0)] = coins[0][0]

q = collections.deque([(0, 0, 0), (0, 0, 1)]) # (r, c, k)

visited = set()

if (0, 0, 0) in dp:

visited.add((0, 0, 0))

if (0, 0, 1) in dp:

visited.add((0, 0, 1))

for r in range(m):

for c in range(n):

for k in range(3): # k = 0, 1, 2

if (r, c, k) not in dp:

dp[(r, c, k)] = -float('inf')

Iterate through the grid

for r in range(m):

for c in range(n):

current_coin = coins[r][c]

Consider coming from (r-1, c) (South)

if r > 0:

for k_prev in range(3):

if dp[(r - 1, c, k_prev)] != -float('inf'):

Option 1: Don't use disruptor field

new_k = k_prev

new_val = dp[(r - 1, c, k_prev)] + current_coin

dp[(r, c, new_k)] = max(dp[(r, c, new_k)], new_val)

Option 2: Use disruptor field if current_coin

is negative and k_prev < 2

if current_coin < 0 and k_prev < 2:

new_k = k_prev + 1

new_val = dp[(r - 1, c, k_prev)]

dp[(r, c, new_k)] = max(dp[(r, c, new_k)], new_val)

Consider coming from (r, c-1) (East)

if c > 0:

for k_prev in range(3):

if dp[(r, c - 1, k_prev)] != -float('inf'):

Option 1: Don't use disruptor field

new_k = k_prev

new_val = dp[(r, c - 1, k_prev)] + current_coin

dp[(r, c, new_k)] = max(dp[(r, c, new_k)], new_val)

Option 2: Use disruptor field if current_coin

is negative and k_prev < 2

if current_coin < 0 and k_prev < 2:

new_k = k_prev + 1

new_val = dp[(r, c - 1, k_prev)]

dp[(r, c, new_k)] = max(dp[(r, c, new_k)], new_val)

max_aetherium = -float('inf')

for k in range(3):

max_aetherium = max(max_aetherium, dp[(m - 1, n - 1, k)])

return max_aetherium

✓ (Correct)

(b) Model Response

Figure B.2: A complete example of a narrative reformulation ($f_{\text{narr}} = f_{\text{solve}} = \text{Gemini-2.5-Flash}$), where f_{solve} correctly implements the intended dynamic programming specification.