FALCON: Feedback-driven Adaptive Long/short-term memory reinforced Coding Optimization system

Zeyuan Li[*1], **Yangfan He**[*2,3], **Lewei He**[1], **Jianhui Wang**[4], **Tianyu Shi**[5], **Bin Lei**[6], **Yuchen Li**[7], **Qiuwu Chen**[7]

[1] School of Software, South China Normal University
[2] University of Minnesota - Twin Cities
[3] Henan RunTai Digital Technology Group Co., Ltd., China
[4] University of Electronic Science and Technology of China
[5] University of Toronto [6] University of Connecticut [7] AI center, AIGCode Inc.
2023024326@m.scnu.edu.cn he000577@umn.edu
helewei@m.scnu.edu.cn 2022091605023@std.uestc.edu.cn
*

Abstract

Recently, large language models (LLMs) have achieved significant progress in automated code generation. Despite their strong instruction-following capabilities, these models frequently struggle to align with user intent. Specifically, they are hampered by datasets that lack diversity and fail to address specialized tasks or edge cases. Furthermore, challenges in supervised fine-tuning (SFT) and reinforcement learning from human feedback (RLHF) lead to failures in generating precise, human-intent-aligned code. To tackle these challenges and improve the code generation performance for automated programming systems, we propose a novel **F**eedback-driven **A**daptive **L**ong/short-term memory reinforced **C**oding **O**ptimizatio**N** framework (i.e., FALCON) to enhance automated programming system performance. From the global level, Long-term memory improves code quality by retaining and applying learned knowledge, while from the local level, Short-term memory allows for the incorporation of immediate feedback from compilers and AI systems. Additionally, we introduce meta-reinforcement learning with feedback rewards to solve the global-local bi-level optimization problem, enhancing the model's adaptability across diverse code generation tasks. Evaluations using benchmarks such as APPs and CodeUltraFeedback demonstrate that our approach not only increases the functional correctness of the generated code, but also improves its overall quality. Our open-sourced code can be found at `https://github.com/liturte/FALCON`.

1 Introduction

The development of Large Language Models (LLMs) has significantly advanced automated code generation [1]. Models like CodeLLaMA [2] and DeepSeek-Coder [3], tailored for code-centric tasks, have demonstrated outstanding performance across programming challenges. While LLMs excel in instruction-following through tuning [4], they often misalign with user intent, making feedback-based adjustments critical. For example, InstructGPT [5] leverages reinforcement learning with human feedback (RLHF), and CodeRL [6] uses compilation feedback to refine model performance. Similarly, CompCoder [7] enhances code compilability with compiler feedback, and RLTF [8] offers

[*]Equal Contribution.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

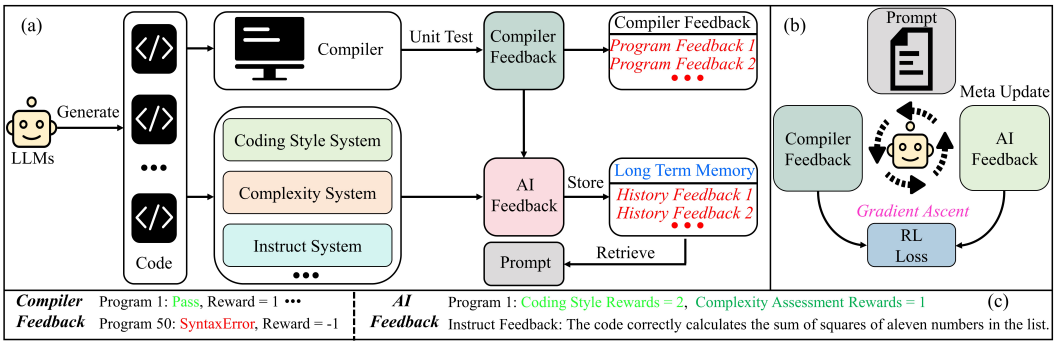

Figure 1: (a) Code generation and evaluation process using compiler and AI feedback systems. (b) Prompt retrieval from long-term memory for improved generation. (c) Reinforcement learning integrates feedback from compiler and AI feedback for model optimization.

fine-grained feedback on compiler errors. However, current RL frameworks generate compilation errors and overlook non-differentiable features like coding style [4], affecting performance. To address these challenges, we propose a reinforcement learning system combining long-term and short-term memory feedback. Long-term memory tracks trends over time for higher-quality code retrieval, while short-term memory captures recent errors and immediate feedback. The main contributions of this paper are as follows:

- We combine **Short/Long-Term Memories for Reinforcement Learning** with short-term memory for real-time corrections and long-term memory for experiences from accumulated historical runs.
- We incorporate **Non-Differentiable Code Features** such as coding style and readability into the feedback loops, enabling a more comprehensive evaluation of generation quality.
- We involve **Meta-Reinforcement Learning** for better generalization across various tasks.

2 Methodology

During code generation, the system stores task descriptions, generated code, and feedbacks (e.g., compilation results, code style, complexity) in long-term memory. By retrieving this information, the model references high-quality code, avoids mistakes, and meets the required standards. After code generation, a judge model evaluates the code and calculates rewards based on feedback, which is then used to update the model's parameters. All generated code and feedback are stored for future reference and optimization. Our framework is shown in Figure 1.

2.1 Long-Term Memory Feedback

Retrieving information from long-term memory significantly improves code quality. We use the FAISS framework [9] to retrieve relevant historical code, feedback, and evaluation scores. Task descriptions and feedback are transformed into embedding vectors and then indexed. During code generation, a query vector from the current task retrieves the top-k most similar historical data, guiding the process and avoiding past errors. The prompt template is provided in the appendix.

Consider a set of historical data $\mathcal{D} = (t_i, f_i, e_i)_{i=1}^{n}$, where t_i represents the task description, f_i is the corresponding feedback, and e_i is the evaluation score. We use an embedding function $\phi(\cdot)$ to transform these tasks and feedback into embedding vectors $\boldsymbol{v}_i = \phi(t_i, f_i)$ and index them using FAISS.

During the code generation phase, the current task description $\boldsymbol{t}_{\text{current}}$ and feedback $\boldsymbol{f}$ are transformed into a query vector $\boldsymbol{q} = \phi(\boldsymbol{t}_{\text{current}})$. We compute the similarity between the query vector $\boldsymbol{q}$ and the historical vectors $\boldsymbol{v}_i$ using cosine similarity $\cos(\boldsymbol{q}, \boldsymbol{v}_i)$, and retrieve the top-$k$ most similar historical tasks. The retrieval process can be represented as:

$$\{(t_{i_1}, f_{i_1}, e_{i_1}), \ldots, (t_{i_k}, f_{i_k}, e_{i_k})\} = \text{Top-}k\,(\boldsymbol{v}_i \mid i = 1, 2, \ldots, n) \tag{1}$$

By referencing these most relevant historical tasks and feedbacks, the system can guide the current code generation process with past mistakes avoided and ultimate code quality improved.

2.2 Short-Term Memory Feedback

During the reinforcement learning phase, we use the generated code $\hat{w}$ to construct the reinforcement learning loss function, as shown below:

$$L_{r1} = -\sum_{t=S_{\text{fine}}}^{E_{\text{fine}}} R_{\text{fine}}(\hat{w}_t) \log p(\hat{w}_t | D, \theta, \hat{w}_{1:t-1}) \tag{2}$$

where $R_{\text{fine}}(*)$ represents the reward coefficient, and S_{fine} and E_{fine} denote the start and end positions of the code snippet, respectively. These values are determined based on different types of feedback.

2.2.1 Compiler Feedback

For compiler feedback, we adopt the same settings as CodeRL:

$$R_{\text{coarse}}(\hat{W}) = \begin{cases} 1.0, & \text{if } FB(\hat{W}) \text{ is pass} \\ -0.3, & \text{if } FB(\hat{W}) \text{ is failure} \\ -0.6, & \text{if } FB(\hat{W}) \text{ is runtime error} \\ -1.0, & \text{if } FB(\hat{W}) \text{ is syntax error} \end{cases} \tag{3}$$

where the value of R_{coarse} is based on compiler feedback with $S_{\text{coarse}} = 0$ and $E_{\text{coarse}} = T$ for the start and end positions.

For fine-grained feedback, we follow RLTF, using adaptive rewards proportional to test case success:

$$R_{\text{error}}(\hat{W}) = -0.3 + 1.3 \times \frac{N_{\text{pass}}}{N_{\text{pass}} + N_{\text{fail}}} \tag{4}$$

We introduce a reward mechanism that adjusts based on short-term memory recall of error rates and test error counts. This integrates past performance to improve decision-making by considering historical error patterns:

$$R_{\text{negative}} = -\sum_{\text{error}} N_{\text{error}} \times P_{\text{error}} \tag{5}$$

where N_{error} is the short-term recall of error frequency, and P_{error} represents the long-term recall of error proportions.

2.2.2 AI Feedback

Furthermore, we design the LLM-as-a-Judge approach to assess code quality. By leveraging the advanced reasoning capabilities of large language models (LLMs), we can identify the complexity and subtle nuances of the code, enabling an evaluation based on code preferences. We focus on the following coding preferences: (1) **Coding Style** refers to the consistency in formatting and structure, ensuring clarity and maintainability, including naming conventions, indentation, and document comments; (2) **Code Complexity** measures the logical complexity of the code, emphasizing resource optimization and efficiency; and (3) **Instruction-Following** assesses how well the code adheres to specific requirements, focusing on strict adherence to predefined specifications or tasks. We use an evaluation model with scores ranging from -1 to 2 as reward signals in the reinforcement learning process. Templates for prompting are provided in the appendix.

2.3 Meta-Reinforcement Learning for Bi-level Code Generation Optimization

As shown in Figure 2, we utilize a meta-reinforcement learning framework to optimize code generation, integrating both long-term and short-term memories for better adaptability. From the global level, Long-term memory D_{long} stores historical tasks, generated codes, and feedbacks to provide valuable context, while from the local level, short-term memory D_{short} focuses on recent feedback to enable real-time adjustments. This approach leverages the MAML framework [10] for efficient task adaptation with minimal updates.

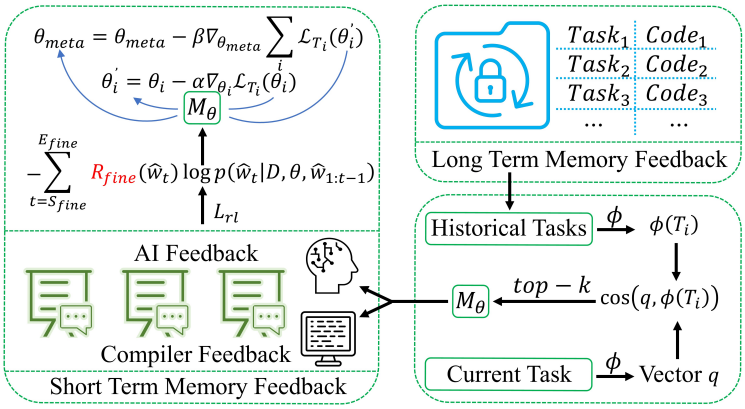

Figure 2: Our proposed training framework integrates long-term and short-term memory feedback for code generation. Long-term memory retrieves relevant historical tasks to guide the current task, while short-term memory feedback—including AI feedback and compiler feedback—optimizes generated code via reinforcement learning. Meta-reinforcement learning further enhances adaptability through inner loop (local task optimization) and outer loop (global optimization across tasks).

Long-term memory enhances code generation by retrieving similar past tasks. Task descriptions are embedded into a query vector $\boldsymbol{q}$ via an embedding function ϕ. The top-k most relevant historical tasks are retrieved using cosine similarity, and the model $\mathcal{M}_\theta$ generates the code $\hat{W}$ based on the retrieved tasks:

$$\hat{W} = \mathcal{M}_\theta\left(\text{top-}k\left(\cos(\boldsymbol{q}, \phi(\mathcal{T}_i)) \mid i = 1, 2, \ldots, n\right)\right)$$

Short-term memory is used to adapt the model locally by adjusting its parameters based on recent feedback. For each task $\mathcal{T}_i$, the inner loop optimization updates the parameters:

$$\theta_i' = \theta_i - \alpha\nabla_{\theta_i}\mathcal{L}_{\mathcal{T}_i}(\theta_i)$$

where α is the learning rate and $\mathcal{L}_{\mathcal{T}_i}$ is the task-specific loss function.

The outer loop performs global optimization of the meta-learning parameters θ_{meta} by aggregating feedback across multiple tasks:

$$\theta_{\text{meta}} = \theta_{\text{meta}} - \beta\nabla_{\theta_{\text{meta}}} \sum_i \mathcal{L}_{\mathcal{T}_i}(\theta_i')$$

where β is the meta-learning rate. This ensures better generalization across tasks.

The overall framework combines short-term and long-term memory feedbacks with meta-reinforcement learning to achieve coordinated optimization for both global generalization and local task adaptation:

$$\theta_{\text{final}} = \text{Optimize}\left(\theta_{\text{meta}}, \theta, \{\theta_i'\}\right)$$

3 Evaluation

Due to page limitations, we have provided a detailed evaluation in the appendix. Our evaluation confirms that our framework effectively enhances the model's coding capabilities, not only in the functional correctness of code but also in other metrics such as coding style.

4 Conclusions and Future Work

In this work, we introduce FALCON by integrating long-term and short-term memory feedbacks to optimize code generation for automated programming systems through meta-RL strategies. Long-term memory retains past interactions to improve code generation, while short-term memory enables dynamic adjustments based on recent feedback. Our proposed framework can reduce manual effort and make automated programming systems more effective. In the future, we aim to improve the diversity of programming languages and consider more complex code for different types of hardware (e.g., CPU, GPU, FPGA) with analysis of their related costs.

References

[1] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. Large language models meet nl2code: A survey, 2023. URL https://arxiv.org/abs/2212.09420.

[2] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[3] 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] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[5] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf.

[6] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/8636419dea1aa9fbd25fc4248e702da4-Paper-Conference.pdf.

[7] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback, 2022. URL https://arxiv.org/abs/2203.05132.

[8] Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=hjYmsV6nXZ.

[9] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library, 2024. URL https://arxiv.org/abs/2401.08281.

[10] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017. URL https://arxiv.org/abs/1703.03400.

A Detailed Evaluation Result

A.1 Quantitative Evaluation on APPS

To ensure a fair comparison, we use the CodeT5 770M model as our baseline. Our benchmarks include the latest advancements that integrate reinforcement learning (RL) with large language models (LLMs), particularly CodeRL, PPOCoder, and RLTF. For evaluation, we applied the same benchmarks and settings used in these previous works. The results demonstrate that our approach delivers additional performance improvements, surpassing other RL-based methods. This indicates that, with appropriate feedback, RL can effectively optimize the output space of models, thereby enhancing the quality of code generation. Specifically, our method achieved pass@1 rates of 8.60%, 2.56%, and 1.25% in the Introductory, Interview, and Competition categories, respectively. The experimental results are illustrated in Table 2.

To validate the scalability and robustness of our framework, we conducted experiments with the larger model, DeepSeek-Coder-Instruct-6.7B, to further evaluate its performance. Notably, the improvement on introductory-level tasks was significant, which can be attributed to the use of long-term memory that enhanced the quality of generated data and further unlocked the model's potential. The results are illustrated in Table 3.

A.2 Quantitative Evaluation on HumanEval and MBPP

To further validate the effectiveness of our method, we evaluated the zero-shot performance of the DeepSeek-Coder-Instruct model, trained with our method on our custom dataset, using the well-established MBPP and HumanEval benchmarks. We also compared these results against other reinforcement learning methods, such as PPOCoder and RLTF. The experimental results are illustrated in Table 1.

Compared to other reinforcement learning methods, our method consistently outperforms in both the HumanEval and MBPP benchmarks. The significant advantage of our method can be attributed to its diversified feedback mechanism. Unlike other methods that may focus on a single metric, our method continuously optimizes the model's generation capability through multi-dimensional feedback. This approach proves particularly effective in complex tasks and demonstrates a strong ability to enhance the generation of correct code, especially in a zero-shot learning setting.

Table 1: The results of pass@1 on the MBPP and HumanEval benchmarks. We evaluate the LLMs' performance in code generation under a zero-shot learning setting.

Model	Humaneval	MBPP
DeepSeek-Coder-Instruct	73.8	74.9
PPOCoder	76.8	76.2
RLTF	76.8	75.9
Ours	**82.9**	**80.7**

A.3 Quantitative Evaluation on CODAL-Bench

In addition to evaluating the functional correctness of the code, to validate the effectiveness of short-term memory feedback, we used CODAL-Bench, a rigorous and comprehensive benchmark designed to assess and compare LLMs for consistency with coding preferences. We used the DeepSeek-Coder-Instruct-6.7B model to conduct tests on the CODAL-Bench benchmark. After applying the framework, there was noticeable improvement in various coding preferences, particularly in Code Complexity and Coding Style. This can be attributed to the inclusion of feedback on these aspects in the short-term memory. However, the improvement in Instruction Following was not as significant. The results are illustrated in Figure 3.

Table 2: Quantitative evaluation on APPS benchmark."Intro": introductory, "Inter": interview, "Comp": competition-level tasks. To ensure a fair comparison, We apply these RL-based methods, including PPOCoder, CodeRL, and RLTF, using the same base model, CodeT5, as a backbone. We also compared with models that have a larger number of parameters.

Method	Size	pass@1				pass@5			
		Intro	Inter	Comp	all	Intro	Inter	Comp	all
Codex	12B	4.14	0.14	0.02	0.92	9.65	0.51	0.09	2.25
GPT-Neo	2.7B	3.90	0.57	0	1.12	5.50	0.80	0	1.58
CodeT5 base	770M	3.85	0.58	0.02	1.05	8.52	1.53	0.25	2.82
PPOCoder	770M	4.06	0.79	0.15	1.32	9.97	2.06	0.70	3.37
CodeRL	770M	7.08	1.86	0.75	2.69	16.37	4.95	2.84	6.81
RLTF	770M	8.40	2.28	1.10	3.27	18.60	5.57	**3.70**	7.87
Ours	770M	**8.60**	**2.56**	**1.24**	**3.50**	**19.75**	**5.85**	3.57	**8.17**

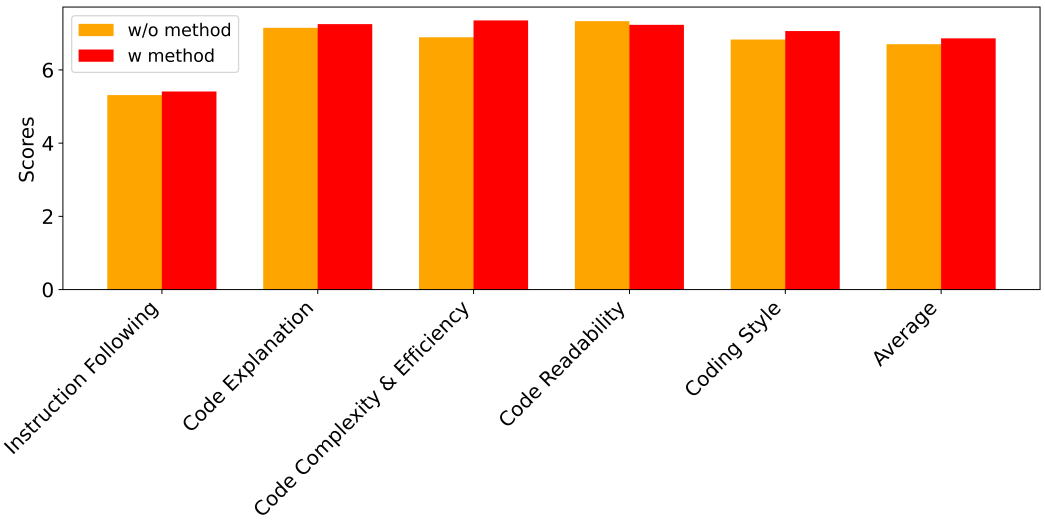

Figure 3: Quantitative Evaluation on CODAL-Bench

Table 3: Different Large Language Model as the backbone

Model	Size	Method	Intro	Inter	Comp	All
DeepSeek-Coder	6.7B	w/o	16.70	7.20	2.30	8.12
DeepSeek-Coder	6.7B	w	22.40	8.52	3.70	10.33

B Example

We provided an example of code generation with long-term memory. When generating code without long-term memory, it often results in repetitive ValueError issues. By incorporating long-term memory to retrieve the most relevant code blocks and embedding them as context during generation, the quality of the generated code can be significantly improved.

Generate directly

```
def solve():
    p = stdin.readline().strip()
    n = len(p)
    ans = (n - 1) * 9 + int(p[0])
    for i in range(1, n):
        ans = max(ans, int(p[i]) * 10 ** (n - i - 1) + int(p[i - 1]) * 10 ** (n - i) + int(p[i + 1:]) + 1)
    return ans
```

ValueError: invalid literal for int() with base 10: ''

Generate with long-Term Memory

```
def solve():
    p = stdin.readline().strip()
    n = len(p)
    ans = (n - 1) * 9 + int(p[0])
    for i in range(1, n):
        suffix = int(p[i + 1:]) if i < n - 1 else 0
        ans = max(ans, int(p[i]) * 10 ** (n - i - 1) + int(p[i - 1]) * 10 ** (n - i) + suffix + 1)
    return res
```

Long-Term Memory

```
Task:You are given
three integers k...
Code: def solve(k, pa,
pb):
    dp = [0] * 1001
    dp[0] = 1
    for i in range(1,
1001):
        dp[i] = dp[i - 1] *
pa + dp[i - 1] * pb
    ans = 0
    for i in range(k,
1001):
        ans += dp[i - k] *
(pb / (pa + pb))
    return ans
Compiler Feedback:-
0.6
Coding Style: 2
Complexity : 0
Instruct Feedback: ...
```

Figure 4: An example of code generation with long-term memory incorporated.

C Prompts

We have compiled relevant prompt templates for code generation based on long-term memory retrieval and AI feedback.

Table 4: Code Generation Template

Please write a Python function based on the task description, referencing the historical context for inspiration. Ensure that the generated code follows the provided requirements and avoids the listed errors. **Instruction:** TASK: [instruction] **Context of relevant code:** - Historical Task: [Brief description of a similar task] - Code: [Code snippet] - Style Score: [Style score] - Efficiency Score: [Efficiency score] - Additional Feedback: [Additional comments or issues] **Requirements:** 1. Ensure the generated code adheres to best practices for Python, including proper naming conventions, consistent formatting, and coding standards. 2. Optimize the code for performance, avoiding unnecessary recursion or nested loops. 3. Use built-in or efficient library functions whenever applicable to improve both readability and performance. **Avoid the following errors:** - [Historical error] – Avoid structural or logical issues found in previous code snippets. **Output:**

Table 5: Coding Style Assessment Template.

Evaluate the coding style of provided code segments. Assess how well the code adheres to the best practices of the language, focusing on readability, maintainability, and efficiency in line with the language's idiomatic style. **Evaluation Criteria** *Readability*: Is the code easy to read and understand? *Maintainability*: Can the code be easily modified or extended? *Efficiency*: Does the code execute tasks in an efficient manner? *Adherence to Idiomatic Style*: Does the code follow the stylistic norms and conventions specific to the programming **Reward:** Rate outputs on a scale of -1 to 2: **-1. Poor Adherence:** The code significantly deviates from standard practices, showing poor readability, maintainability, and efficiency. **0. Basic Adherence:** The code makes some effort to follow language conventions but lacks consistency in readability, maintainability, or efficiency. **1. Good Adherence:** The code generally follows standards, demonstrating adequate readability, maintainability, and efficiency, though with room for improvement. **2. Excellent Adherence:** The code exemplifies best practices, with high readability, maintainability, and efficiency, fully adhering to idiomatic conventions.

D Error Category

Due to the differences in languages accepted by Compiler Feedback during unit tests for various language tasks, we have standardized the definition of sub-errors in Compiler Feedback. The table 8 9 10 below outlines our specifications for Python, C, and Java.

Table 6: Complexity Assessment Template.

Evaluate the solutions and code provided by the assistant based on their complexity. Assess how well the code manages complexity while achieving the desired outcomes. **Evaluation Criteria** *Time Efficiency*: Does the code minimize computational time? *Resource Efficiency*: Does the code use resources (e.g., memory, CPU) judiciously? *Algorithm Effectiveness*: Are the chosen algorithms accurate and efficient in achieving the desired outcomes? *Optimization*: Has the code been optimized for quick processing without compromising the solution's correctness or efficiency? **Reward:** Rate outputs on a scale of -1 to 2: **-1. Overly Complex:** The code is unnecessarily complicated, with a high level of complexity that makes it hard to understand or maintain. **0. Acceptable Complexity:** The code has a reasonable level of complexity, but there may be opportunities to simplify further. **1. Moderately Simple:** The code is simple and well-organized, with minimal complexity and clear logic. **2. Optimal Simplicity:** The code exemplifies the best practices in minimizing complexity while ensuring functionality.

Table 7: Instruction following Assessment Template.

Evaluate the assistant's fidelity to provided instructions. Assess how accurately the assistant's responses align with user directives, noting any deviations and their justification. **Evaluation Criteria** *Precision in Following Instructions*: Does the assistant adhere to the specifics of the provided instructions? *Justification for Deviations*: If deviations occur, are they justified by critical necessity or explicit user request? *Alignment with User Directives*: How well do the assistant's responses match the user's specified needs and expectations? *Necessity of Deviations*: Are any deviations from instructions made only in situations deemed absolutely necessary or upon direct user request? **Reward:** Rate outputs on a scale of -1 to 2: **-1. Non-Compliant:** The assistant frequently deviates from instructions without necessity or user consent. **0. Acceptable Complexity:** The assistant shows some adherence to instructions but deviates without strong justification. **1. Moderately Simple:** The assistant generally follows instructions, with deviations occurring but justified by necessity or user request. **2. Optimal Simplicity:** The assistant follows instructions closely, with minimal deviations, all of which are well justified.

Table 8: Common Python Errors with Categories.

Sub-error	Description	Category
Syntax Error	Code contains syntax errors that cause the compilation to fail.	Syntax Error
Indentation Error	Wrong indentation format.	Syntax Error
Index Error	Index operation is out of bounds.	Runtime Error
Type Error	An operation or function was applied to an object of an inappropriate type.	Runtime Error
Value Error	An operation or function received an argument with the correct type but with an inappropriate value.	Runtime Error
EOF Error	The input() function encountered an end-of-file condition (EOF) without reading any data.	Runtime Error
Timeout Error	Code execution time exceeds time limit.	Runtime Error
Name Error	A local or global name is not defined.	Runtime Error
Key Error	A mapping (dictionary) key is not found in the set of existing keys.	Runtime Error
Import Error	The imported package is not found.	Runtime Error
ZeroDivision Error	The second argument of a division or modulo operation is zero.	Runtime Error
Recursion Error	Code execution recursive operation exceeds the maximum limit.	Runtime Error

Table 9: Common C Language Errors with Categories.

Sub-error	Description	Category
Segmentation Fault	Accessing memory that the program doesn't have permission to access.	Runtime Error
Null Pointer Dereference	Attempting to dereference a pointer that is NULL.	Runtime Error
Buffer Overflow	Writing data outside the allocated buffer memory.	Runtime Error
Memory Leak	Dynamically allocated memory not being freed.	Runtime Error
Syntax Error	A syntax mistake in the code, such as a missing semicolon.	Syntax Error
Type Mismatch	Assigning a value of one type to a variable of another type.	Syntax Error
Uninitialized Variable	Using a variable before it has been initialized.	Runtime Error
Undefined Behavior	Code that can exhibit unpredictable behavior depending on compiler or runtime environment.	Runtime Error
Division by Zero	Attempting to divide a number by zero.	Runtime Error
Infinite Loop	A loop that never terminates due to incorrect logic.	Runtime Error

Table 10: Common Java Language Errors with Categories.

Sub-error	Description	Category
NullPointerException	Attempting to access an object with a 'null' reference.	Runtime Error
ArrayIndexOutOf BoundsException	Accessing an array index that is out of bounds.	Runtime Error
ClassCastException	Casting an object to a subclass it is not an instance of.	Runtime Error
NumberFormat Exception	Attempting to convert a string to a number, but the string doesn't have the appropriate format.	Runtime Error
StackOverflowError	Recursive method calls that exceed the stack size.	Runtime Error
Syntax Error	Any mistake in the code structure such as missing braces or semicolons.	Syntax Error
ClassNotFound Exception	The Java class is not found at runtime.	Runtime Error
IllegalArgument Exception	A method has been passed an illegal or inappropriate argument.	Runtime Error
ArithmeticException	Division by zero or other illegal arithmetic operations.	Runtime Error
UnsupportedOperation Exception	When a requested operation is not supported.	Runtime Error