

Appendix

These supplementary materials include the prompt templates (§A), the details of the static AST patterns (§B), and the implementation details (§C), and additional results (§D). The source code is available at the following anonymous link: <https://anonymous.4open.science/r/efficientcode-8CED/README.md> and will be released upon publication.

A Prompt Templates

A.1 Generation LLM

You are a software engineer with Python expertise, and your task is to complete the code with the given prefix. Your generated code should be the optimal in time efficiency and memory usage.

During each generation step, you need to rethink step by step whether your generation is optimal in time efficiency. You need to self-evaluate whether the generated code is the optimal in time efficiency, if it is not optimal, you need to reflection and regenerate it.

There are test examples included in the prompt and you need to analyze it. The completed code needs to be included in a code block. {task}

A.2 Critique LLM (default)

{code snippet: “code“ }

Please rate the above code snippet based on the following performance-related criteria:

1. Time Complexity (Big-O Notation): Assess the time complexity of the code and assign a score reflecting its efficiency in terms of how the time complexity scales with input size. A score closer to 1 indicates highly efficient code (e.g., $O(\log n)$, $O(n)$), while a score closer to 0 indicates inefficient code (e.g., $O(n^2)$, $O(2^n)$).
2. Space Complexity (Big-O Notation): Evaluate the space complexity of the code, considering memory usage for variables and data structures. A score closer to 1 represents minimal space usage (e.g., $O(1)$, $O(n)$), while a score closer to 0 reflects high memory consumption (e.g., $O(n)$).
3. Running Time Performance: Provide an estimate of the expected running time for typical input sizes (small, medium, large). Assign a score between 0 and 1 based on the speed of execution, with 1 being the fastest and 0 being the slowest.
4. Memory Usage Efficiency: Evaluate how effectively the code uses memory resources, including variable allocations and data structures. A score closer to 1 indicates optimal memory usage, while a score closer to 0 indicates inefficiency or excessive memory consumption.
5. AST Analysis for Performance: Perform an Abstract Syntax Tree (AST) analysis to assess the efficiency of the code’s structure and operations. The analysis should consider factors like loop depth, redundant expressions, and operator usage. Provide a performance score based on how well-optimized the AST is for execution. A score closer to 1 represents an optimized AST structure, while a score closer to 0 indicates a structure with potential inefficiencies.
6. If the code contains syntax error, the final score is 0.
7. The perplexity of the code snippet is as low as better.

Output: A single numerical value between 0 and 1 that represents the overall performance score based on the above criteria. No additional text or analysis should be provided. Just the final performance score.

{code snippet: “code“ }

Please rate the above code snippet based on the following performance-related criteria:

1. Time Complexity (Big-O Notation): Assess the time complexity of the code and assign a score reflecting its efficiency in terms of how the time complexity scales with input size. A score closer to 1 indicates highly efficient code (e.g., $O(\log n)$, $O(n)$), while a score closer to 0 indicates inefficient code (e.g., $O(n^2)$, $O(2^n)$).
2. Space Complexity (Big-O Notation): Evaluate the space complexity of the code, considering memory usage for variables and data structures. A score closer to 1 represents minimal space usage (e.g., $O(1)$, $O(n)$), while a score closer to 0 reflects high memory consumption (e.g., $O(n)$).
3. Running Time Performance: Provide an estimate of the expected running time for typical input sizes (small, medium, large). Assign a score between 0 and 1 based on the speed of execution, with 1 being the fastest and 0 being the slowest.
4. Memory Usage Efficiency: Evaluate how effectively the code uses memory resources, including variable allocations and data structures. A score closer to 1 indicates optimal memory usage, while a score closer to 0 indicates inefficiency or excessive memory consumption.
5. If the code contains syntax error, the final score is 0.
6. The perplexity of the code snippet is as low as better.

Output: A single numerical value between 0 and 1 that represents the overall performance score based on the above criteria. No additional text or analysis should be provided. Just the final performance score.

510

{code snippet: “code“ }

Please rate the above code snippet based on the following performance-related criteria:

1. Time Complexity (Big-O Notation): Assess the time complexity of the code and assign a score reflecting its efficiency in terms of how the time complexity scales with input size. A score closer to 1 indicates highly efficient code (e.g., $O(\log n)$, $O(n)$), while a score closer to 0 indicates inefficient code (e.g., $O(n^2)$, $O(2^n)$).
2. Space Complexity (Big-O Notation): Evaluate the space complexity of the code, considering memory usage for variables and data structures. A score closer to 1 represents minimal space usage (e.g., $O(1)$, $O(n)$), while a score closer to 0 reflects high memory consumption (e.g., $O(n)$).
3. Running Time Performance: Provide an estimate of the expected running time for typical input sizes (small, medium, large). Assign a score between 0 and 1 based on the speed of execution, with 1 being the fastest and 0 being the slowest.
4. Memory Usage Efficiency: Evaluate how effectively the code uses memory resources, including variable allocations and data structures. A score closer to 1 indicates optimal memory usage, while a score closer to 0 indicates inefficiency or excessive memory consumption.
5. AST Analysis for Performance: Perform an Abstract Syntax Tree (AST) analysis to assess the efficiency of the code’s structure and operations. The analysis should consider factors like loop depth, redundant expressions, and operator usage. Provide a performance score based on how well-optimized the AST is for execution. A score closer to 1 represents an optimized AST structure, while a score closer to 0 indicates a structure with potential inefficiencies.
6. If the code contains syntax error, the final score is 0.

Output: A single numerical value between 0 and 1 that represents the overall performance score based on the above criteria. No additional text or analysis should be provided. Just the final performance score.

512

513 A.5 Critique LLM (without AST, without perplexity)

{code snippet: “code“ }

Please rate the above code snippet based on the following performance-related criteria:

1. Time Complexity (Big-O Notation): Assess the time complexity of the code and assign a score reflecting its efficiency in terms of how the time complexity scales with input size. A score closer to 1 indicates highly efficient code (e.g., $O(\log n)$, $O(n)$), while a score closer to 0 indicates inefficient code (e.g., $O(n^2)$, $O(2^n)$).
2. Space Complexity (Big-O Notation): Evaluate the space complexity of the code, considering memory usage for variables and data structures. A score closer to 1 represents minimal space usage (e.g., $O(1)$, $O(n)$), while a score closer to 0 reflects high memory consumption (e.g., $O(n)$).
3. Running Time Performance: Provide an estimate of the expected running time for typical input sizes (small, medium, large). Assign a score between 0 and 1 based on the speed of execution, with 1 being the fastest and 0 being the slowest.
4. Memory Usage Efficiency: Evaluate how effectively the code uses memory resources, including variable allocations and data structures. A score closer to 1 indicates optimal memory usage, while a score closer to 0 indicates inefficiency or excessive memory consumption.
5. If the code contains syntax error, the final score is 0.

Output: A single numerical value between 0 and 1 that represents the overall performance score based on the above criteria. No additional text or analysis should be provided. Just the final performance score.

514

515 B Static AST Patterns

- 516 1. **Nested Loops:** The program detects nested loops, which are a potential source of inefficiency
517 due to increased time complexity.

```
# 1. Detect Nested Loops
nested_loops_penalty = 10 # Larger penalty for nested loops
for node in ast.walk(tree):
    if isinstance(node, ast.For) or isinstance(node, ast.While):
        for other_node in ast.walk(tree):
            if isinstance(other_node, (ast.For, ast.While)) and node != other_node:
                if isinstance(node, ast.For) and isinstance(other_node, ast.For):
                    score -= nested_loops_penalty
```

- 518 2. **Redundant Function Calls (Inside Loops):** Checks for repeated calls to functions like
519 expensive_function within loops and suggests moving them outside the loop.

```
# 2. Detect Redundant Function Calls Inside Loops
redundant_function_calls_penalty = 8 # Moderate penalty for redundant calls
for node in ast.walk(tree):
    if isinstance(node, (ast.For, ast.While)):
        for stmt in node.body:
            if isinstance(stmt, ast.Expr) and isinstance(stmt.value, ast.Call):
                func_name=stmt.value.func.id if isinstance(stmt.value.func, ast.Name) else ""
                if func_name == "expensive_function":
                    score -= redundant_function_calls_penalty
```

- 520 3. **Redundant Function Calls (Memorization):** Flags redundant calls to functions with
521 identical arguments and suggests memorization.

```
# 3. Detect Redundant Function Calls (Memoization Opportunities)
redundant_function_calls_memoization_penalty = 5
function_calls = {}
for node in ast.walk(tree):
    if isinstance(node, ast.Expr) and isinstance(node.value, ast.Call):
        func_name = node.value.func.id if isinstance(node.value.func, ast.Name) else ""
        if func_name == "expensive_function":
            if func_name not in function_calls:
                function_calls[func_name] = set()
            args = tuple(ast.dump(arg) for arg in node.value.args)
            if args in function_calls[func_name]:
```

```

        score -= redundant_function_calls_memoization_penalty
        function_calls[func_name].add(args)

```

- 522 4. **Inefficient Use of Data Structures:** Identifies inefficient data structures, like using a list for
523 membership testing.

```

# 4. Detect Inefficient Use of Data Structures (list for membership test)
inefficient_data_structure_penalty = 6
for node in ast.walk(tree):
    if isinstance(node, ast.Expr) and isinstance(node.value, ast.Compare):
        if isinstance(node.value.left, ast.Name) and isinstance(node.value.comparators[0], ast.List):
            score -= inefficient_data_structure_penalty

```

- 524 5. **Excessive Function Calls in Loops:** Detects function calls in loops that might be expensive
525 and suggests optimization.

```

# 5. Detect Excessive Function Calls in Loops
excessive_function_calls_penalty = 7
for node in ast.walk(tree):
    if isinstance(node, (ast.For, ast.While)):
        for stmt in node.body:
            if isinstance(stmt, ast.Expr) and isinstance(stmt.value, ast.Call):
                if isinstance(stmt.value.func, ast.Name):
                    if stmt.value.func.id in ["expensive_function", "some_expensive_function"]:
                        score -= excessive_function_calls_penalty

```

- 526 6. **Unnecessary Recursion:** Finds unnecessary recursion and suggests a more efficient iterative
527 approach.

```

# 6. Detect Unnecessary Recursion
unnecessary_recursion_penalty = 12
for node in ast.walk(tree):
    if isinstance(node, ast.FunctionDef):
        if any(isinstance(n, ast.Call) and isinstance(n.func, ast.Name) \
                and n.func.id == node.name for n in ast.walk(node)):
            score -= unnecessary_recursion_penalty

```

- 528 7. **Deeply Nested Conditional Statements:** Warns when the conditional logic is too deeply
529 nested, which can affect readability and efficiency.

```

# 7. Detect Deeply Nested Conditional Statements
deeply_nested_conditions_penalty = 4
for node in ast.walk(tree):
    if isinstance(node, ast.If):
        depth = 0
        parent = node
        while isinstance(parent, ast.If):
            depth += 1
            parent = parent.parent if hasattr(parent, 'parent') else None
        if depth > 3:
            score -= deeply_nested_conditions_penalty

```

- 530 8. **Inefficient String Concatenation:** Detects inefficient string concatenation inside loops and
531 suggests using the join() method.

```

# 8. Detect Inefficient String Concatenation
inefficient_string_concatenation_penalty = 6
for node in ast.walk(tree):
    if isinstance(node, ast.For):
        for stmt in node.body:
            if isinstance(stmt, ast.Expr) and isinstance(stmt.value, ast.BinOp) \
                and isinstance(stmt.value.op, ast.Add):
                if isinstance(stmt.value.left, ast.Str) and isinstance(stmt.value.right, ast.Str):
                    score -= inefficient_string_concatenation_penalty

```

- 532 9. **Inefficient File/Database Operations:** Flags file and database operations inside loops,
533 which could be optimized by batching or caching.

```

# 9. Detect Inefficient File/Database Operations
inefficient_io_operations_penalty = 10
for node in ast.walk(tree):
    if isinstance(node, ast.With):
        for stmt in node.body:
            if isinstance(stmt, ast.Expr) and isinstance(stmt.value, ast.Call):
                func_name = stmt.value.func.id if isinstance(stmt.value.func, ast.Name) else ""

```

		<pre>if func_name in ["open", "execute", "query"]: score -= inefficient_io_operations_penalty</pre>
534	10. Large Functions:	Identifies large functions that could benefit from refactoring for clarity and performance
535		<pre># 10. Detect Large Functions (Refactoring Opportunity) large_function_penalty = 8 for node in ast.walk(tree): if isinstance(node, ast.FunctionDef): function_size = len(node.body) if function_size > 20: # Arbitrary threshold for large functions score -= large_function_penalty</pre>
536	11. Inefficient Loop Terminology:	Identifies inefficient loop constructs like <code>range(len(data))</code>
537		<pre># 11. Detect Inefficient Loop Terminology (e.g., range(len(data)) vs for item in data) inefficient_loop_terminology_penalty = 6 for node in ast.walk(tree): if isinstance(node, ast.For): if isinstance(node.iter, ast.Call) and isinstance(node.iter.func, ast.Name) \ and node.iter.func.id == "range": if isinstance(node.iter.args[0], ast.Call) \ and isinstance(node.iter.args[0].func, ast.Name) and node.iter.args[0].func.id == "len": score -= inefficient_loop_terminology_penalty</pre>
538	12. Potential Syntax Errors:	Identifies syntax errors or incomplete code.
		<pre>try: tree = ast.parse(code) except SyntaxError as e: issues.append(f"SyntaxError detected:{e}. Code might be incomplete. Analyzing partial AST.") score -= 20 # Penalize incomplete code</pre>

539 C Implementation Details

540 We employ the official open-source implementations of `EffiLearner`⁸ and `PerfCodeGen`⁹ for
541 evaluation. Since no official implementation is available for `Self-Debug`, we re-implemented it
542 based on the descriptions provided in the original paper. For the standard vanilla Perplexity-based
543 decoding, we evaluate commonly used sampling strategies, including top-p and best-of-n for a fair
544 and stable comparison. We set a maximum limit of new tokens to 256 to enforce a stopping criterion
545 for token generation. The temperature is set to 1 by default. Furthermore, we use regular expressions
546 to extract the code portion from the model’s response. If multiple code implementations are contained,
547 we only extract and test the first one.

548 Our method is implemented using beam search and best-of-n selection, with a default beam width $b =$
549 1 and number of trials $n = 50$. Table 5 summarizes the default hyperparameter settings for the differ-
550 ent configurations of our method used in the ablation study. The default configuration, corresponding
551 to the main paper results, uses the composite scoring function $\alpha \cdot r_{\text{AST}} + \beta \cdot r_{\text{LLM}} + \gamma \cdot \text{PP}$ with $b = 1$ and
552 $n = 50$. The source code will be released upon publication and is currently available at the following
553 anonymous link: <https://anonymous.4open.science/r/efficientcode-8CED/README.md>

	α	β	γ
$\alpha \cdot r_{\text{AST}} + \beta \cdot \text{PP}$ ($b = 1, n = 50$)	1.2	0.4	–
$\alpha \cdot r_{\text{LLM}} + \beta \cdot \text{PP}$ ($b = 1, n = 50$)	1.0	0.4	–
$\alpha \cdot r_{\text{AST}} + \beta \cdot r_{\text{LLM}} + \gamma \cdot \text{PP}$ ($b = 1, n = 50$)	1.3	1	0.4
$\alpha \cdot r_{\text{AST}} + \beta \cdot \text{PP}$ ($b = 50, n = 1$)	1.5	0.5	–
$\alpha \cdot r_{\text{LLM}} + \beta \cdot \text{PP}$ ($b = 50, n = 1$)	0.9	0.5	–
$\alpha \cdot r_{\text{AST}} + \beta \cdot \text{LLM}$ ($b = 50, n = 1$)	1.2	0.8	–

Table 5: Weight Configuration for Different Reward Functions.

⁸ <https://github.com/huangd1999/EffiLearner>

⁹ <https://github.com/SalesforceAIResearch/perfcodegen>

554 D Additional Results

555 D.1 Experiments on HumanEval+ and COFFE dataset

556 We present the detailed quantitative results in Table 6, comparing different methods on HumanEval+
557 and COFFE dataset. This serves as a supplementary analysis to Table 1 in the main paper.

Datasets	Models	Methods	ET(Avg)↓	ET(Median)↓	NET(Avg)↓	NET(Median)↓	NMU↓	Correctness↑
HumanEval+	DeepSeek-6.7b	Perplexity (Best-of-n)	1.39	1.26	0.89	0.85	1.25	49.89%
		Perplexity (Top-p)	1.44	1.26	1.06	0.86	1.15	50.12%
		Self-Debug	0.73	1.20	0.89	0.95	1.17	54.76%
		EffiLearner	0.64	1.05	0.75	0.75	1.01	13.15%
		PerfCodeGen	0.72	0.84	0.73	0.72	0.97	55.81%
		Ours	0.05	0.82	0.63	0.62	0.91	62.20%
	CodeLlama-7b	Perplexity (Best-of-n)	2.51	2.38	2.02	1.96	1.15	48.77%
		Perplexity (Top-p)	2.55	2.41	2.14	1.94	1.13	47.95%
		Self-Debug	2.81	2.25	1.96	1.02	1.12	52.83%
		EffiLearner	1.70	1.10	1.81	1.80	1.06	15.94%
		PerfCodeGen	1.78	1.91	1.79	1.75	1.04	54.29%
		Ours	1.54	1.89	1.71	1.68	0.95	60.37%
	OpenCoder-8b	Perplexity (Best-of-n)	1.33	1.21	0.83	0.81	1.18	51.72%
		Perplexity (Top-p)	1.37	1.23	0.98	0.82	1.11	52.04%
		Self-Debug	0.69	1.12	0.85	0.91	1.09	56.94%
		EffiLearner	0.60	0.98	0.68	0.70	0.95	11.08%
		PerfCodeGen	0.68	0.81	0.70	0.68	0.94	58.02%
		Ours	0.04	0.75	0.59	0.60	0.87	64.89%
	CodeLlama-13b	Perplexity (Best-of-n)	1.65	1.33	0.96	0.90	1.31	48.12%
		Perplexity (Top-p)	1.50	1.34	1.10	0.91	1.20	48.39%
		Self-Debug	0.78	1.22	0.93	0.99	1.19	53.41%
		EffiLearner	0.68	1.08	0.79	0.78	1.03	14.87%
		PerfCodeGen	0.75	0.89	0.77	0.73	1.01	54.92%
		Ours	0.17	0.53	0.44	0.51	0.93	60.08%
	StarCoder2-15b	Perplexity (Best-of-n)	1.30	1.18	0.82	0.80	1.15	52.46%
		Perplexity (Top-p)	1.35	1.20	0.95	0.83	1.08	52.71%
		Self-Debug	0.70	1.10	0.84	0.89	1.06	57.12%
		EffiLearner	0.61	0.96	0.66	0.69	0.93	10.58%
		PerfCodeGen	0.66	0.79	0.69	0.67	0.92	58.94%
		Ours	0.03	0.73	0.58	0.59	0.86	65.31%
	DeepseekCoder-v2-16b	Perplexity (Best-of-n)	1.21	1.09	0.76	0.74	1.04	54.31%
		Perplexity (Top-p)	1.26	1.11	0.87	0.78	0.99	54.89%
		Self-Debug	0.65	1.02	0.78	0.84	1.00	59.36%
		EffiLearner	0.55	0.91	0.62	0.64	0.89	9.62%
		PerfCodeGen	0.61	0.75	0.65	0.63	0.88	61.24%
		Ours	0.02	0.67	0.51	0.56	0.82	68.47%
	Qwen2.5-Coder-32b	Perplexity (Best-of-n)	1.10	0.98	0.68	0.66	0.96	56.78%
		Perplexity (Top-p)	1.14	1.01	0.79	0.70	0.91	57.25%
		Self-Debug	0.59	0.96	0.71	0.79	0.94	61.58%
		EffiLearner	0.49	0.85	0.58	0.60	0.84	8.21%
		PerfCodeGen	0.56	0.71	0.60	0.58	0.83	63.45%
		Ours	0.01	0.61	0.47	0.52	0.78	69.83%
COFFE	DeepSeek-6.7b	Perplexity (Best-of-n)	2.15	2.31	1.64	1.78	1.28	40.12%
		Perplexity (Top-p)	2.08	2.22	1.55	1.63	1.24	41.75%
		Self-Debug	2.26	2.33	1.72	1.84	1.21	43.19%
		EffiLearner	1.92	2.05	1.49	1.53	1.10	11.34%
		PerfCodeGen	1.75	1.89	1.38	1.45	1.06	44.66%
		Ours	0.18	1.69	1.21	1.30	0.91	47.90%
	OpenCoder-8b	Perplexity (Best-of-n)	1.96	2.08	1.48	1.62	1.18	41.88%
		Perplexity (Top-p)	1.91	2.03	1.39	1.55	1.13	43.50%
		Self-Debug	2.14	2.26	1.63	1.75	1.16	44.95%
		EffiLearner	1.83	1.96	1.35	1.40	1.08	10.97%
		PerfCodeGen	1.69	1.81	1.32	1.39	1.03	46.22%
		Ours	0.17	1.60	1.15	1.22	0.88	48.77%
	StarCoder2-15b	Perplexity (Best-of-n)	2.01	2.18	1.51	1.67	1.22	39.21%
		Perplexity (Top-p)	1.98	2.14	1.44	1.58	1.17	40.56%
		Self-Debug	2.20	2.30	1.69	1.80	1.19	42.37%
		EffiLearner	1.78	1.93	1.36	1.41	1.05	9.95%
		PerfCodeGen	1.66	1.77	1.29	1.34	1.00	43.66%
		Ours	0.16	1.57	1.10	1.18	0.86	46.93%
	DeepseekCoder-v2-16b	Perplexity (Best-of-n)	2.29	2.42	1.70	1.88	1.29	37.73%
		Perplexity (Top-p)	2.20	2.35	1.62	1.76	1.25	39.00%
		Self-Debug	2.34	2.49	1.79	1.92	1.22	40.84%
		EffiLearner	1.94	2.09	1.45	1.52	1.12	10.11%
		PerfCodeGen	1.82	1.94	1.39	1.46	1.06	42.11%
		Ours	0.19	1.65	1.23	1.29	0.90	45.88%
	Qwen2.5-Coder-32b	Perplexity (Best-of-n)	2.42	2.55	1.83	1.94	1.31	36.27%
		Perplexity (Top-p)	2.37	2.49	1.76	1.86	1.27	38.03%
		Self-Debug	2.46	2.59	1.87	1.95	1.24	39.96%
		EffiLearner	2.03	2.15	1.53	1.60	1.14	10.42%
		PerfCodeGen	1.91	2.01	1.43	1.50	1.08	41.02%
		Ours	0.20	1.70	1.28	1.35	0.91	44.66%

Table 6: Comparison of the generated code efficiency on **HumanEval+** and **COFFE**. Methods that explicitly optimize efficiency are shaded; best results are in **bold**.

D.2 Additional Results on Method Processing Time

To further support the findings presented in the main paper, we report the median processing time during the decoding phase for each method on the Mercury, HumanEval+, and COFFE datasets in Table 7–9. These results serve as a supplement to Table 2 in the main paper.

Consistent with our earlier findings, our method demonstrates a significant advantage in decoding efficiency compared to baseline methods specifically designed to optimize the generated code beyond the perplexity, e.g., Self-Debug, EffiLearner and PerfCodeGen. These baselines achieve improved runtime performance or correctness in the generated outputs, but do so at the expense of substantially higher decoding latency—in some cases, one to two orders of magnitude slower than our method. In contrast, our approach achieves superior code efficiency, competitive or superior functional correctness, while maintaining fast generation speeds. It is worth noting that the vanilla baselines not explicitly optimized for code efficiency (i.e., Perplexity) remain faster, but they do not offer the same runtime benefits in the generated code as our approach.

Models	Perplexity (Best-of-n)	Perplexity (Top-p)	Self-Debug	EffiLearner	PerfCodeGen	Ours
DeepSeek-6.7b	2.89	2.22	3998.65	3555.88	3979.42	25.17
OpenCoder-8b	2.91	2.05	4117.52	4242.41	4884.11	23.36
StarCoder2-15b	6.06	5.13	7111.11	5989.32	7373.07	153.32
DeepseekCoder-v2-16b	6.27	5.84	7215.69	6004.49	7517.46	65.59
Qwen2.5-Coder-32b	13.33	11.03	25543.87	18787.55	20089.51	115.58

Table 7: The median processing time (in seconds) of each method on the **Mercury** dataset.

Models	Perplexity (Best-of-n)	Perplexity (Top-p)	Self-Debug	EffiLearner	PerfCodeGen	Ours
DeepSeek-6.7b	3.18	2.45	4212.84	3763.19	4189.66	27.14
OpenCoder-8b	3.21	2.28	4328.75	4463.52	5078.94	25.36
StarCoder2-15b	6.61	5.52	7358.99	6224.08	7613.42	159.47
DeepseekCoder-v2-16b	6.68	6.21	7432.77	6230.55	7735.28	69.18
Qwen2.5-Coder-32b	14.02	11.59	26438.73	19420.86	20833.11	121.46

Table 8: The median processing time (in seconds) of each method on the **HumanEval+** dataset.

Models	Perplexity (Best-of-n)	Perplexity (Top-p)	Self-Debug	EffiLearner	PerfCodeGen	Ours
DeepSeek-6.7b	2.74	2.08	3895.42	3450.31	3870.15	24.38
OpenCoder-8b	2.78	1.95	4012.67	4129.84	4775.33	22.41
StarCoder2-15b	5.83	4.90	6982.76	5855.24	7232.58	149.11
DeepseekCoder-v2-16b	6.01	5.60	7099.28	5883.20	7400.38	63.45
Qwen2.5-Coder-32b	12.95	10.67	25032.15	18345.77	19675.28	112.76

Table 9: The median processing time (in seconds) of each method on the **COFFE** dataset.

D.3 Impact of Different Critique LLMs

We present the detailed quantitative results of varying critique LLMs in Table 10. This serves as a supplementary analysis to Table 3 in the main paper.

Critique LLMs	ET (Avg)	ET (second, Median)	NET (Avg)	NET (Median)	NMU	Correctness
DeepSeek-6.7b	0.04	0.92	0.66	0.63	1.42	63.89%
CodeLlama-7b	0.04	0.97	0.68	0.66	1.22	61.13%
OpenCoder-8b	0.04	0.86	0.62	0.59	0.89	64.17%
DeepseekCoder-v2-16b	0.04	0.85	0.61	0.59	0.92	64.79%
Qwen2.5-Coder-32b	0.04	0.83	0.60	0.58	0.88	65.02%

Table 10: More metrics on the impact of different critique LLMs of our method with OpenCoder-8b as the target generation LLM on **Effibench**.

574 D.4 Hyperparameter Tuning

575 As part of our preliminary study, we performed hyperparameter tuning to determine effective configurations for sampling strategies in code generation. This tuning was conducted on the Codellama-7b-instruct-hf and Codellama-13b-instruct-hf models on the HumanEval+ datasets. The outcomes of these tuning experiments are summarized below.

579 This tuning was performed using the CodeLlama-7B-Instruct-HF and CodeLlama-13B-Instruct-HF models on the HumanEval+ dataset. The results of these experiments, detailing the average execution time (ET) of generated code under various hyperparameter settings, are presented in Table 11.

α	β	γ	Codellama-7b-instruct-hf	Codellama-13b-instruct-hf
1	1	0.5	1.61	0.21
1.3	1	0.5	1.55	0.19
1.3	1	0.4	1.54	0.17
1.5	1	0.5	1.49	0.19
1.5	0.7	0.3	1.56	0.18

Table 11: The ET (Avg) of the generated code on **HumanEval+** dataset across different settings.

582 D.5 Ablation Study

583 We further investigate how different combinations of configuration components influence the efficiency of generated code across the COFFE and HumanEval+ benchmarks. Figure 6 and 7 present the results. Notably, while individual components vary in their contributions, combining all different rewards generally yields the most effective outcomes in terms of code efficiency. Among the components, the AST reward appears to contribute the least when used in isolation.

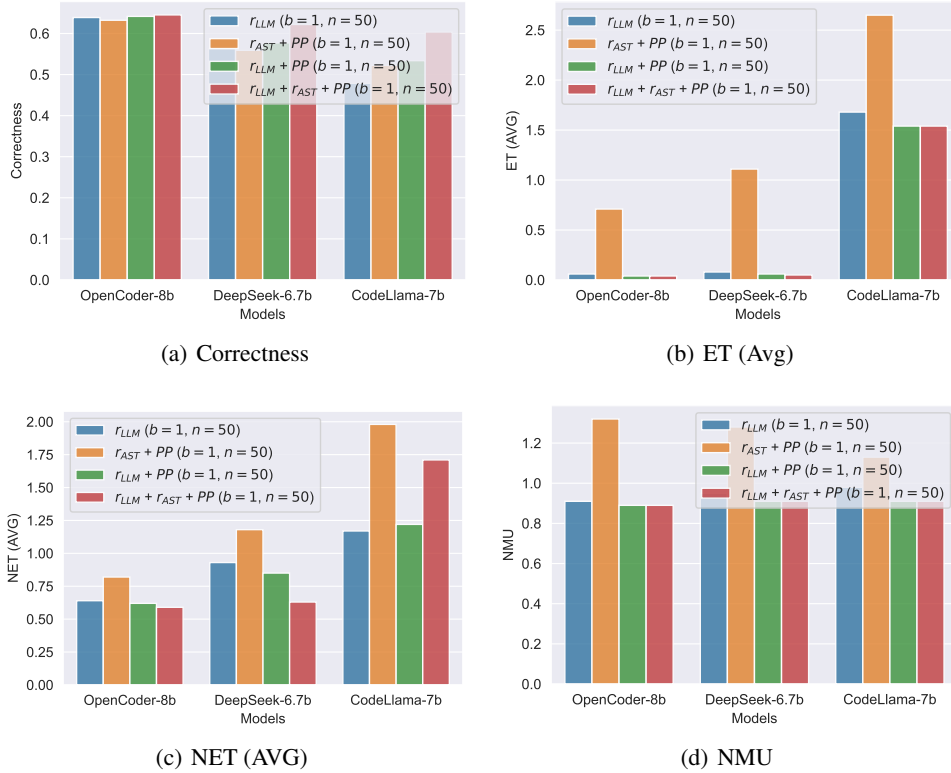
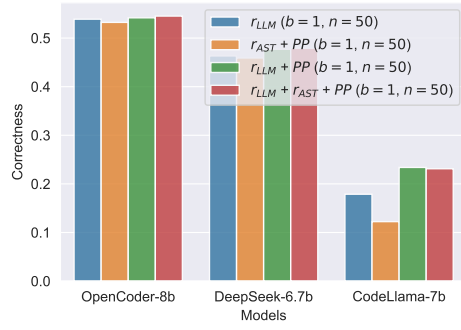
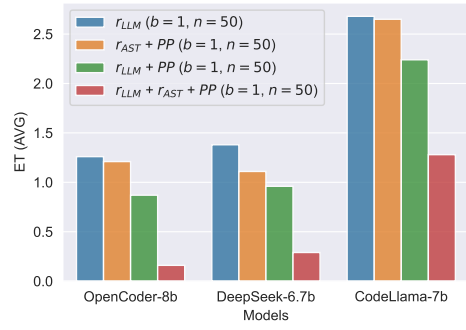


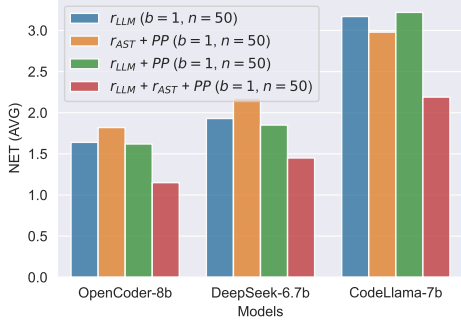
Figure 6: Comparison across different configurations on **HumanEval+** with OpenCoder-8B as f_{θ} .



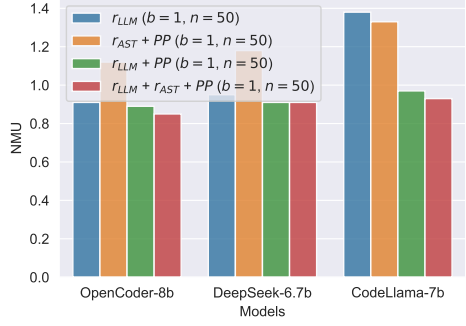
(a) Correctness



(b) ET (Avg)



(c) NET (Avg)



(d) NMU

Figure 7: Comparison across different configurations on **COFFE** with OpenCoder-8B as f_θ .