Translation of Low-Resource COBOL to Logically Correct and Readable Java leveraging High-Resource Java Refinement

Shubham Gandhi
TCS Research
gandhi.shubham@tcs.com

Manasi Patwardhan
TCS Research
manasi.patwardhan@tcs.com

Jyotsana Khatri
TCS Research
jyotsana.khatri@tcs.com

Lovekesh Vig
TCS Research
lovekesh.vig@tcs.com

Raveendra Kumar Medicherla
TCS Research
raveendra.kumar@tcs.com

ABSTRACT

Automated translation of legacy code to modern programming languages is the need of the hour for modernizing enterprise systems. This work specifically addresses automated COBOL to Java translation. Traditional rule-based tools for this perform statement-wise translation, overlooking possible modularization and refactoring of the source COBOL code to translate to human-readable target Java code. Our investigation reveals that state-of-the-art Large Language Models (LLMs) in the domain of code encounter difficulties with regard to logical correctness and readability when directly translating low-resource COBOL code to Java. To address these challenges, we propose an LLM-based workflow, leveraging temperature sampling and refinement-based strategies, to not only ensure logical correctness of the translation but also maximize the readability of the target Java code. We exploit the fact that, due to their extensive exposure to human-written Java codes during pre-training, the LLMs are more equipped with profound comprehension and capability for refining translated Java codes than COBOL to Java translation. With a dataset sourced from CodeNet, we perform sequential refinement of the translated high-resource Java code with execution-guided logic feedback followed by LLM-based readability feedback. We demonstrate that this yields better performance in terms of logical correctness (81.99% execution accuracy) and readability (0.610 score), than LLM based translation with test cases and readability guidance (60.25% and 0.539) or refinement of the translation task itself (77.95% and 0.572).

CCS CONCEPTS

• **Computing methodologies** → *Machine translation*; *Neural networks*; • **Software and its engineering** → *Software evolution*; *Maintaining software*.

KEYWORDS

Code Translation, Low Resource Programming Languages, Large Language Models, Code Readability, Self-Refinement

LLM4Code '24, April 20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0579-3/24/04
https://doi.org/10.1145/3643795.3648388

ACM Reference Format:
Shubham Gandhi, Manasi Patwardhan, Jyotsana Khatri, Lovekesh Vig, and Raveendra Kumar Medicherla. 2024. Translation of Low-Resource COBOL to Logically Correct and Readable Java leveraging High-Resource Java Refinement. In *2024 International Workshop on Large Language Models for Code (LLM4Code '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3643795.3648388

1 INTRODUCTION

As digital landscapes evolve, efficiently modernizing legacy codes becomes necessary. Manual efforts demand expertise in both legacy and modern languages. Hence, there is a need for automated legacy to modern language translation solutions for correct, efficient and scalable navigation of legacy systems. In this work, we focus on translating COBOL code to Java. There has been extensive work done on code translation with LLMs for high-resource languages such as Java, Python, etc [4, 7, 9, 14, 24, 27, 28]. Low resource languages such as COBOL do not get much exposure during pre-training, as we subsequently observe that direct COBOL to Java translation (no additional guidance) results, in terms of execution accuracy (*EA*) by state-of-the-art code LLMs such as ChatGPT[1] and WizardCoder[2] [19], are quite poor (19.57% and 8.70%, respectively), yielding logically incorrect Java codes as showcased in Table 1. In addition to the COBOL's low-resource setting, the lower performance is due to differences in paradigms and syntax. COBOL's verbose English-like syntax, contrasting with Java's concise C++-like syntax, presents a challenge in mapping COBOL statements to abstract high-level Java constructs. Adapting COBOL's procedural and imperative paradigm to Java's object-oriented one requires intricate restructuring for more modular and reusable Java code. A COBOL code snippet can be mapped to a Java library or function, requiring refactoring. As illustrated in Table 1, the direct LLM-based translated Java code has a flatter (non-modularized) structure with no function calls (*readinput, calculateAnswer, calculateSum*) and refactored (abstracted) code (*input.split()*). More examples are available in Github page[3] Section C.

Although there exist syntax-directed tools for this task, these are rule-based translations. A program is represented as an AST (Abstract Syntax Tree) and translated to target language using hand-written rules [21]. One such tool for COBOL to Java translation is

[1] gpt-3.5-turbo-0613 API endpoint https://platform.openai.com/docs/models/gpt-3-5
[2] Quantized model https://huggingface.co/TheBloke/WizardCoder-15B-1.0-GPTQ
[3] https://anonymous.4open.science/r/CobolToJavaCodeTranslation-66A3/CodeTranslation_LLM4Code_Appendix.pdf

Table 1: Example Java code generated by syntax-directed approach (Complete code in Github Page); LLM based direct translation; our workflow, Incorrect vs. Correct Logic; Non-Modularized vs. Modularized vs. Over-modularized; Non-Abstracted vs. Abstracted code. Problem: There are 999 towers of height 1, (1+2), (1+2+..999), at intervals of 1m. *a* meters for tower 1 and *b* meters for tower 999 are not covered with snow. Find total snow, assuming atleast 1m deep snow between two adjacent towers.

Source COBOL code	Syntax-directed approach	Direct LLM-based Translation	Our Workflow

.....
ACCEPT INP.
PERFORM ablen TIMES
 	
PERFORM VARYING j FROM cur BY 1
UNTIL INP(j:1) = SPACE
END-PERFORM
 	
COMPUTE len = j - cur
MOVE INP(cur:len) TO ab11(i)
COMPUTE cur = j + 1
ADD 1 TO i
 	
END-PERFORM.
 	
COMPUTE div = ab11(2) - ab11(1).
MOVE 0 TO s.
MOVE 1 TO i.
 	
PERFORM div TIMES
ADD i TO s
ADD 1 TO i
END-PERFORM.
 	
COMPUTE t = s - ab11(2).
MOVE t TO ans.
DISPLAY ans.
STOP RUN.
	.....
	public class Abc_099_b extends Program {
	public int getAblen() {
	return super.getDisplayInt
	(0,2,false,false,false);
	}
	public String getAblenAsString() {
	return super.toString(0,2);
	}
	public void setAblen(int val) {
	super.setDisplayInt
	(0,2,val,false,false,false);
	}
	.....
	public static void main(String[] args) {
	processCmdLineArgs(args);
	Abc_099_b instance_ = new Abc_099_b();
	instance_.execute(null);
	System.exit(instance.getReturnCode());
	}
	public void execute(ProgramEnv __env) {
	initialize(this);
	doCobolGotoStart();
	}
	.....
	Paragraph main=new Paragraph(this) {
	.....}
	public void initialize(Program p) {
	.....}
	}
		.....
		public class ABC_099_B {
		public static void main
		(String[] args) {
		.....
		int cur = 1;
		String INP = scanner.nextLine();
		int i = 1;
		...
		for (int k = 0; k < ablen; k++) {
		j = cur;
		while (INP.charAt(j-1) != ' ') {
		j++;
		}
		len = j - cur;
		ab11[i] = Integer.parseInt(INP.
		substring(cur-1, cur-1+len));
		cur = j + 1;
		i++;
		}
		div = ab11[1] - ab11[0];
		...
		for (int k = 0; k < div; k++) {
		s += i;
		i++;
		}
		t = s - ab11[1];
		.....
		}
		}
			.....
			public class ABC_099_B {
			public static void main
			(String[] args) {
			.....
			int[] ab = readInput(scanner);
			long ans = calculateAnswer(ab);
			.....
			}
			
			private static int[] readInput
			(Scanner scanner) {
			.....
			String[] parts = input.split(" ");
			ab[0] = Integer.parseInt(parts[0]);
			ab[1] = Integer.parseInt(parts[1]);
			return ab;
			}
			
			private static long calculateAnswer
			(int[] ab) {
			.....
			long div = b - a;
			long s = calculateSum(div);
			long t = s - b;
			return t;
			}
			
			private static long calculateSum
			(long n) { ..... }
			}

opencobol2java[4]. Unlike LLMs, though, these tools produce logically correct Java programs (no compilation or execution errors), they tend to produce statement-wise translations as illustrated in Table 1 (complete code is on Github page). This does not produce optimally modularized Java code with semantically meaningful code segments, which are required for *maintainability* and *comprehension*. Moreover, such tools require tedious maintenance of translation rules and translators for each of the several dialects of COBOL leading to project delays. To address these issues, for COBOL to Java translation, we focus on both aspects logical correctness and human readability of the target Java code. We propose an LLM-based workflow (Figure 1) that leverages the following: (i) temperature sampling [11] and code refinement [20], which have proven to be beneficial for other code related tasks, viz. Code Generation [6, 7, 36, 42], Repair [38–40], Optimization [20], Software Testing [35], etc, (ii) the high resource nature of the target language, Java, for refinement and (iii) availability of I/O Test Cases.

Empirical analysis with two best-performing code LLMs (ChatGPT and WizardCoder) shows that our best-performing workflow achieves 86.34% execution accuracy and 0.646 readability score on CodeNet, validating the following claims: (i) Temperature sampling aids translation to get better Java code in terms of logical correctness and readability. (ii) Aspect (logical correctness and readability) specific refinements of translated Java code yield better code than trying to ensure logical correctness and readability as a part of COBOL to Java translation task. (iii) Existing LLM-based translation approach by Pan et al. [24], targeting only high-resource programming languages, refines the target code in the context of the source code. In contrast, Java being high-resource, we claim that refinement of translated Java code without having COBOL code in the context (refinement of translated Java code) yields better performance, as compared to refinement of target Java code with the COBOL code in the context (refinement of translation task), where low-resource COBOL acts as a distractor for refinement. (iv) Aspect (logical correctness and readability) specific sequential refinements lead to better Java code than combined refinement for both aspects.

[4] https://opencobol2java.sourceforge.net/

2 APPROACH

Our approach consists of a workflow using a base LLM, consisting of (i) Translation (*TM*) (ii) Logic Refinement (*LRM*), and (iii) Readability Refinement (*RRM*) Modules (*aspect*-specific refinements).

2.1 Translation Module (*TM*)

There are four components which create distinct configurations of prompts for *TM* - (i) *BASE* (ii) *I/O* (iii) *INS* and (iv) INS_{def}. The primary objective of *BASE* is to evaluate the LLM's inherent ability to perform direct translation. For this, we keep the prompt minimal without additional aids. With *I/O*, we enhance *TM* by incorporating sample I/O pairs of Test Cases in the prompt, which can aid *TM*, resulting in Java code that may accurately replicate the I/O behavior of the COBOL code (giving expected output for given input as well as adhering to I/O format). *INS* provides basic instructions to produce logically correct and readable Java code , whereas, INS_{def} consists of carefully crafted guidelines to preserve the logical correctness and readability of the translated code. Since readability can often be ambiguous, certain definitions related to readability criteria applicable for code translation are included in

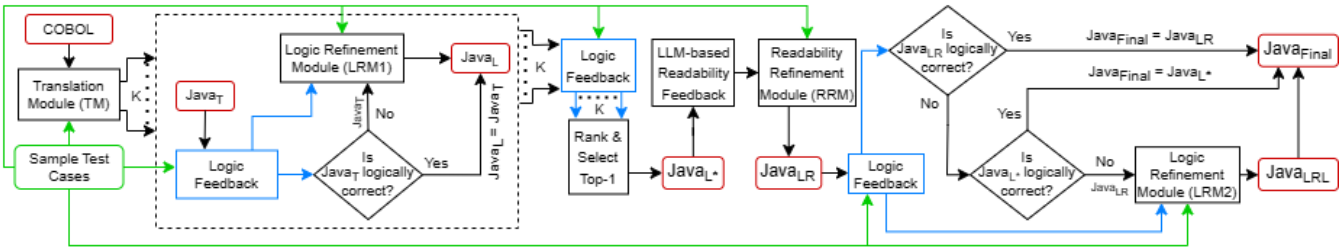

Figure 1: Our Approach: LLM-based aspect-specific sequential refinement workflow defined in Section 2.4

this component. These criteria are: Overall Readability [34], Abstraction, Modularization, Dead (Unnecessary) code, Logical segmentation, Complexity [3], Comments [34], Naming convention [34], Testability and Alignment [10]. Some of these criteria, such as abstraction, modularization, logical segmentation and unnecessary code, are specifically designed to address readability concerns of COBOL-to-Java translation, whereas others are sourced from prior literature. Attaching distinct combinations of the above-explained components to *BASE* leads us to the following six prompt configurations for *TM*: (i) TM_{BASE} (Github page prompt 1) (ii) $TM_{BASE+I/O}$ (Github page prompt 2) (iii) $TM_{BASE+INS}$ (Github page prompt 3) (iv) $TM_{BASE+INS_{def}}$ (Github page prompt 4) (v) $TM_{BASE+I/O+INS}$ (Github page prompt 5) and (vi) $TM_{BASE+I/O+INS_{def}}$. The input to *TM* is COBOL code, and output is the translated Java code with a selected prompt configuration. As an example we illustrate the prompt for $TM_{BASE+I/O+INS_{def}}$ configuration in Prompt 1. Prompts for rest of the configurations are illustrated on the Github Page.

2.2 Logic Refinement Module (*LRM*)

The *LRM* aims to ensure logical correctness of the resultant translated Java code. On the similar lines of Huang et al. [12], we do not rely on the intrinsic capabilities of LLMs for the refinement, but instead use external feedback for the same. We compile and execute the Java code generated by the prior module. If the resultant Java code compiles and generates expected output for all the test cases, we bypass *LRM* for that sample. In case of failure, we gather the error messages (from compiler or runtime system) and categorize the errors into four types: (i) Compilation Errors (*CE*), (ii) Runtime Errors (*RE*), (iii) Functional errors (*FE*) (code executes but does not provide expected output) and (iv) Non-Termination Errors (*NTE*), to generate logic feedback. We consider *CE* to be more severe than *RE* and *NTE*, which in turn would be more severe than *FE*. If the logic feedback indicates any of the above errors, we proceed to *LRM* using the feedback along with the test cases as part of the refinement prompt (Prompt 2). Note that this prompt does not have the source COBOL code in the context. The input to *LRM* is a logically incorrect Java code with logic feedback, and output is the Java code repaired by the LLM, currently in a non-iterative setting.

2.3 Readability Refinement Module (*RRM*)

This module generates LLM-based readability feedback for the translated Java code and rectifies the same in case of negative feedback. To provide feedback for the translated Java code, we do not rely on the LLM's inherent understanding of readability and use the aforementioned task-specific readability criteria. We include the definitions of the criteria in the feedback prompt (Github page prompt 8) and ensure that the feedback is both detailed (mentions specific portions of the code along with the understanding of the corresponding problematic aspects of readability, if any) and comprehensive in nature. The LLM leverages the insights gained from the readability feedback, along with sample I/O, to enhance the readability of the generated Java code by making it more refactored, modularized, and aligned with Java coding standards (Prompt 3). Note that source COBOL code is not in the context.

Prompt 1: $TM_{BASE+I/O+INS_{def}}$ Translation Prompt

```
1 Translate the following COBOL Code to Java to satisfy
  the following sample test cases without hardcoding
  inputs while taking the following guidelines into
  consideration.{OUTPUT_FORMAT}
2 SAMPLE TEST CASES: {test_cases}
3 GUIDELINES:
4 The logic of the code should be preserved when
  translating from COBOL to Java.
5 The resulting Java code should be readable in nature.
6 {READABILITY_DEFINITIONS}
7 COBOL CODE: {cobol}
```

Prompt 2: Logic Refinement Prompt

```
1 Refine and rewrite the following Java Code based on the
  following guidelines and satisfy the following sample
  test cases without hardcoding inputs. {OUTPUT_FORMAT}
2 SAMPLE TEST CASES: {test_cases}
3 GUIDELINES: {logic_feedback}
4 JAVA CODE: {java}
```

Prompt 3: Readability Refinement Prompt

```
1 {READABILITY_DEFINITIONS}. Can you give suggestions to
  improve readability of the following java code based on
  abstraction, modularization, unnecessary code, logical
  segmentation, complexity, comments, naming convention,
  testability and alignment of the following code. Give
  suggestions for each feature separately first. Do not
  fix the code, just provide suggestions.
2 JAVA CODE: {java}
3 SAMPLE TEST CASES: {test_cases}
4 GUIDELINES: {readability_feedback}
5 Now fix the code based on the above guidelines.{
  OUTPUT_FORMAT}
```

2.4 LLM-based Workflow

Figure 1 illustrates our workflow using the above defined modules in the order: *TM*, *LRM*1, *RRM*, *LRM*2, where *LRMx* is the x^{th} LRM instance. Thus, it performs translation followed by sequential *aspect*-specific refinements. For our workflow, we use the best-performing *TM* configuration. We feed a COBOL code to *TM* and generate K samples of Java code ($Java_T$) using temperature sampling. Out of these K samples, the erroneous ones are passed to *LRM*1 to rectify the detected errors discussed above. The logically refined K samples ($Java_L$) are again compiled, executed and ranked based on their logical correctness. The logically correct ones get the highest rank, followed by the ones with *FE*, followed by *RE* or *NTE* followed by *CE*. The readability scores (*RS*) (Explained in section 3.2) are used as tiebreakers. For example, consider that there are N<=K samples with no errors (logically correct), and if $N > 1$, the sample from N with greatest *RS* gets the highest rank. The Java code with the highest rank ($Java_L*$) is forwarded to *RRM*. As logical correctness gets priority over readability, the Java Code generated as an output of *RRM* ($Java_{LR}$) is again checked for logical correctness. If it is logically incorrect and $Java_L*$ is logically correct, we stick to $Java_L*$ as the final output. Otherwise, if both $Java_L*$ and $Java_{LR}$ are logically incorrect, we feed $Java_{LR}$ to *LRM*2 to ensure logical correctness of the resulting Java code ($Java_{LRL}$).

3 EXPERIMENTATION AND RESULTS

3.1 Dataset

We source our dataset from CodeNet [26], which is derived from code submission platforms AIZU[5] and AtCoder[6]. It has 4053 problem statements with multiple submissions in 55 programming languages. To the best of our knowledge, it is the only public dataset having COBOL samples with test cases. We investigate ChatGPT's exposure to CodeNet as a part of pre-training data. We find that although ChatGPT can generate some statistics for CodeNet, it cannot complete partially provided problem statements or COBOL/Java codes in the dataset. This suggests that it might have seen some web description of CodeNet, but not the actual data in terms of problem statements and codes. We select the problem statements that have at the least one accepted COBOL submission and randomly select a submission for each problem, along with provided I/O test cases, resulting in 322 COBOL code samples. This set is consistent across all settings in Table 2. We DO NOT treat the Java submissions for that problem as ground truth, as our metric (discussed in Section 3.2) is independent of the same. As we work in a zero-shot setting, we use all the COBOL codes as test samples. The number of COBOL code lines (ranges from 11-358) serve as our criterion of code difficulty, with distribution: (i) Very Easy: 73 (ii) Easy: 58 (iii) Medium: 69 (iv) Hard: 57 (v) Extra Hard: 65.

3.2 Metrics

As we aim to produce logically correct and readable Java programs, we use the following metrics for evaluation.

3.2.1 Execution accuracy (EA). It represents the percentage of translated Java code samples that execute correctly, i.e. without any errors and yield expected output for all the test cases provided in the dataset. We use *OpenJDK v11.0.20.1* to execute the Java codes.

3.2.2 Traditional Readability Score (RS). We use the approach in Scalabrino et al. [29] for *RS* computation (Range 0-1). They train a classifier using logistic regression on code features defined by Buse and Weimer [5], Dorn [10], Posnett et al. [25] including spatial and textual features (Section 4.2).

3.2.3 LLM-based readability Score. As *RS* does not consider some of the criteria important for COBOL-to-Java translation viz. abstraction, modularization, logical segmentation, and redundancy; we use ChatGPT (prompt illustrated on the Github page) to evaluate the resultant Java code for these criteria.

3.3 Large Language Models (LLMs)

We use two base LLMs pre-trained on natural language as well as code, viz. WizardCoder [19] and OpenAI's ChatGPT(*gpt-3.5-turbo*). Due to computational constraints, we use a 4-bit quantized version of WizardCoder-15B. For temperature sampling, we use temperature 0.7 for more diverse output; for refinement modules we use temperature 0.2 for more deterministic output. We use a *MIG A100* GPU with 20 GiB memory for WizardCoder$_Q$ inference.

3.4 Baselines

All Prompts for the baselines are illustrated on the Github page.

3.4.1 Refinement of Translation. This baseline is designed following Pan et al. [24], to demonstrate that our workflow (refinement of translated Java code) results in better Java code than performing refinement of translation task, as mentioned in Section 1. Same as our proposed workflow, for this baseline, we compute the results with the best performing configuration $TM_{BASE+I/O+INS_{def}}$ and K = 5 temperature samples. We perform two variants of refinement modules, viz. one-step refinement for logical correctness and readability ($TM_{BASE+I/O+INS_{def}} \rightarrow (LRM + RRM)_{trans}$) and aspect-specific sequential refinement ($TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{trans} \rightarrow RRM_{trans} \rightarrow LRM2_{trans}$). Here $_{trans}$ denotes refinement of the translation task with having COBOL code in the context.

3.4.2 One-Step Refinement of translated Java code. Our workflow ($TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{java} \rightarrow RRM_{java} \rightarrow LRM2_{java}$) uses sequential aspect-specific refinement. To validate that it produces better Java code (details in Section 3.5.1), we use this baseline with only single-step refinement of translated Java code for both logical correctness and readability ($TM_{BASE+I/O+INS_{def}} \rightarrow (LRM + RRM)_{java}$). Here $_{java}$ denotes refinement of the resultant java with no COBOL code in the context.

3.4.3 Variations in Temperature Sampling. To demonstrate effectiveness of temperature sampling we execute our workflow with distinct values of K (1, 5, 13). We also study a variation of the workflow where the highest ranking sample is selected to be forwarded to *LRM*1 after *TM* itself (K = 5 → 1) as opposed to passing all 5 samples (output of *TM*) to *LRM*1 and then ranking and selecting the best sample (Section 2.4).

[5] https://onlinejudge.u-aizu.ac.jp.
[6] https://atcoder.jp/

Table 2: W - WizardCoder$_Q$ and C - ChatGPT; K - no. of temperature samples; Bold and Underlined: Overall best for K = 5; Bold: Best in the block; †- Best sample selected after TM; * - Same number (13) of total inference calls.

	Workflow	K	Exec. Acc. (%)		Read. Score (0-1)	
			W	C	W	C
Baselines	$TM_{BASE+I/O+INS_{def}} \rightarrow (LRM + RRM)_{trans}$	5	50.31	76.71	0.643	0.643
	$TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{trans} \rightarrow RRM_{trans} \rightarrow LRM2_{trans}$*	5	**51.86**	**77.95**	0.601	0.572
	$TM_{BASE+I/O+INS_{def}} \rightarrow (LRM + RRM)_{java}$	5	**51.86**	75.78	**0.673**	**0.663**
	$TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{java} \rightarrow RRM_{java} \rightarrow LRM2_{java}$	1	28.88	70.50	0.569	0.582
		5 → 1 †	**51.86**	77.02	0.606	0.603
Translation Module Ablation	TM_{BASE}	5	19.88	36.96	0.605	0.587
	$TM_{BASE+INS}$	5	21.43	45.34	0.408	0.412
	$TM_{BASE+INS_{def}}$	5	18.94	36.65	**0.649**	0.262
	$TM_{BASE+I/O}$	1	22.05	47.83	0.303	0.459
		5	41.93	66.77	0.430	0.519
	$TM_{BASE+I/O}$*	13	**64.29**	**72.67**	0.585	**0.624**
	$TM_{BASE+I/O+INS}$	5	45.03	63.66	0.600	0.536
	$TM_{BASE+I/O+INS_{def}}$	1	25.47	44.72	0.525	0.471
		5	49.07	60.25	0.614	0.539
	$TM_{BASE+I/O+INS_{def}}$*	13	59.32	68.94	0.624	0.577
	$TM_{BASE+I/O+CoT}$	1	9.63	48.45	0.525	0.467
	$TM_{BASE+I/O+INS_{def}+CoT}$	1	17.08	40.06	0.444	0.486
Refinement Modules Ablation	$TM_{BASE+I/O} \rightarrow LRM1_{java}$	5	50.93	75.47	0.559	0.495
	$TM_{BASE+I/O} \rightarrow LRM1_{java} \rightarrow RRM_{java}$	5	51.24	77.95	0.603	0.547
	$TM_{BASE+I/O} \rightarrow LRM1_{java} \rightarrow RRM_{java} \rightarrow LRM2_{java}$*	5	51.24	**79.81**	0.571	0.547
	$TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{java}$	5	52.80	76.40	0.602	0.569
	$TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{java} \rightarrow RRM_{java}$	5	**53.42**	78.88	0.645	**0.609**
	$TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{java} \rightarrow RRM_{java} \rightarrow LRM2_{java}$ $(w/o\ I/O)$*	5	51.24	79.50	**0.652**	0.594
Proposed	$TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{java} \rightarrow RRM_{java} \rightarrow LRM2_{java}$*	5	**53.42**	**81.99**	**0.617**	**0.610**

3.4.4 Direct translation of COBOL to logically correct and readable Java. We treat one of our ablations, $TM_{BASE+I/O+INS_{def}}$ as a baseline to show our workflow's superiority over direct translation.

3.5 Results and Discussion

We address key research questions (RQ) based on results in Table 2.

3.5.1 RQ1: Does Aspect-wise Sequential Refinement of translated Java code better performs than baselines? We observe improvement in *EA* with our proposed workflow over the following baselines: (i) both the variants of Refinement of Translation (ii) One-step refinement of translated Java code (iii) Direct translation of COBOL to logically correct and readable Java without refinement. For baselines (i) and (ii), we observe a lower *EA* for samples with higher code lengths. For some samples, with addition of COBOL code in the context, the refinement prompt exceeds the token limit of LLMs (ChatGPT=4096; WizardCoder$_Q$=2048). This demonstrates another advantage of our workflow needing no contextual COBOL code for refinement. For ChatGPT, we observe substantial improvements in *EA* over these baselines (>5%), whereas for WizardCoder$_Q$ we observe marginal improvements (~2%). As opposed to *EA*, variations with one-step refinement baselines have higher *RS* than our workflow. This is because, as discussed in Section 2.4, our workflow prioritize logical correctness over readability.

3.5.2 RQ2: Do readability enhancements impact code logic? We observe that addressing readability with basic ($TM_{BASE+I/O+INS}$) and elaborate instructions ($TM_{BASE+I/O+INS_{def}}$) shows improvement in *EA* over $BASE + I/O$ for WizardCoder$_Q$ but not for ChatGPT. Hence, we execute our workflow with both best-performing configurations of *TM* ($TM_{BASE+I/O+INS_{def}}$ for WizardCoder$_Q$ and $TM_{BASE+I/O}$ for ChatGPT). Moreover, *RRM* not only improves *RS*, but also *EA* for both LLMs ($TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{java} \rightarrow RRM_{java}$ over $TM_{BASE+I/O+INS_{def}} \rightarrow LRM1_{java}$) showcasing positive effect of readability enhancements over code logic.

3.5.3 RQ3: How does incorporation of Chain of Thought (CoT) Prompting impact performance? For *TM*, we include CoT prompting [16, 37], which has been explored for code generation [11, 13, 18, 36]. We ask the LLMs to generate a high-level plan (CoT) of the COBOL code prior to generating the corresponding Java translation in the same prompt (Prompts and example CoT on the Github page). We also include few-shot examples from a left-out set with manually crafted CoT. We compare CoT-based scores with the best-performing configurations of *TM* to check the effect of inclusion of CoT prompting to these configurations ($TM_{BASE+I/O+INS_{def}+CoT}$ and $TM_{BASE+I/O+CoT}$) with no temperature sampling (K=1). With the inclusion of CoT, we see an average decrease in *EA*, whereas only a marginal increase in *RS*, for both the LLMs. This can be due to the few shot samples causing the prompt to exceed context

limit. Given the higher significance placed on *EA*, we exclude CoT variations of *TM* from our workflow.

3.5.4 RQ4: Does temperature sampling improve EA and RS?. For both LLMs, higher *EA* and *RS* of our workflow as compared to the baseline with K=1 , indicate the positive effect of temperature sampling. Same is indicated by increase in *EA* and *RS*, with increase in temperature samples (K = 13 from 5) for both the LLMs (Github page Figure 2). Temperature sampling helps WizardCoder$_Q$ more than ChatGPT. With the proposed workflow, for a sample, we perform maximum 13 inference calls to the LLMs (5 for temperature sampling for *TM* + 5 for *LRM1* + 2 for *RRM* using LLM-based feedback and 1 for *LRM2*). We perform temperature sampling with K=13, i.e. same number of inference calls to that of the workflow (Indicated by $*$ in Table 2). For ChatGPT, our workflow with $TM_{BASE+I/O+INS_{def}}$ configuration yields better *EA* and *RS* than temperature sampling with the same number of inference calls. However, for WizardCoder$_Q$, the results are reversed.

3.5.5 RQ5: Does a combination of LLMs perform better? With higher performance of temperature sampling for WizardCoder$_Q$ and refinement for ChatGPT, we execute our workflow with Wizard-Coder$_Q$ for *TM* and ChatGPT for all three refinements. This leads to cost savings in terms of lesser number of API calls to ChatGPT. We find the value of K to be 25, where WizardCoder$_Q$ reaches stagnancy as far as *EA* is concerned. With this workflow, we yield the best performance of *EA* = 86.34% and *RS* = 0.646.

3.5.6 RQ6: Does inclusion of I/O improve EA and RS?. Inclusion of I/O of test cases for *TM*, *LRM* and *RRM* consistently yields better *EA* for both LLMs, as intended in Section 2.1. However, it does not consistently result in an increase in *RS*.

3.6 Qualitative Analysis

3.6.1 Contribution of Modules for Logical Error Correction. Figure 2 shows the contribution of each module towards logical error correction (Section 2.2) for ChatGPT (Similar illustration for WizardCoder$_Q$ on Github page Figure 1). With ChatGPT, after *TM* with the best configuration, we observe a higher proportion of *FE* and *RE* than *CE* and *NTE*. For WizardCoder$_Q$, the number of *CE* (22.05%) after *TM* is greater than ChatGPT. *LRM*1 is highly effective for ChatGPT and corrects significant erroneous codes (28.26%), even eliminating all *CE*s. We associate this fixing capability with the detailed error messages being passed iin the logic feedback and sample I/O pairs. For WizardCoder$_Q$, some errors from all categories are fixed, but the increase in logically correct samples (5.60%) is significantly lesser than ChatGPT. For ChatGPT, *RRM* fixes some instances of *RE* and *FE* while also converting some of them to *CE*. For WizardCoder$_Q$ also, although a few errors are fixed, this module results in a net transformation of other errors to *CE*. The attempt of the LLMs to make the code more readable can lead to an overall increase in code size owing to the addition of comments and modularization. For some samples, this also leads to the context length limit being exceeded which ultimately can result in compilation errors for the resultant Java programs. In the future, we plan to take the severity of error types into consideration when making the decision of selection of Java code between the output of *RRM* ($Java_{LR}$) or the output of *LRM*1 ($Java_{L}*$) (section 2.4). Thus, if $Java_{LR}$ results into higher severity errors than $Java_{L}*$, we plan to retain $Java_{L}*$ as the output. This would prevent the propagation of errors from low to higher severity. *LRM*2 fixes some instances of *FE* and *RE* to boost the *EA* for ChatGPT further. However, it is not as effective for WizardCoder$_Q$, resulting in further transformation of other errors (R - 1.86%, N - 1.24%, F - 7.14%) to *CE*. At the output of each module, for both the LLMs, we observe that there is generally a lesser number of *NTE* as compared to other error types. ChatGPT is better than WizardCoder$_Q$ at refinement, given that it fixes a significant amount and variety of errors through sequential refinements. The example demonstrated in Table 1, shows a sample where *TM* gives Java code that causes *RE*. *LRM*1 further improves this to yield Java code that causes only *FE* (Github page Table 2). Further, *RRM* completely fixes this code, which provides the expected outputs for given I/Os.

3.6.2 Bug Analysis. To analyze failure cases, we choose 25 samples, uniformly distributed across difficulty (Section 3.1). We categorize bugs in failed Java codes as: (i) S - Syntactic and Semantic differences between COBOL and Java, (ii) L - Dependency and Logic (missing imports or incorrect logic replication), (iii) M - Model generation capability and (iv) D - Data (incorrect data types, input parsing or output formatting issues). As a single program can have multiple bugs, there is no direct one-to-one mapping between the type of bugs and the error types discussed above. Thus, each bug type can lead to different error types. Table 3a shows the distribution of the 25 samples over the bug types (including no bugs) analyzed at the output of each module for ChatGPT (Github page Table 5 for WizardCoder$_Q$). One sample may belong to more than one bug type. ChatGPT is able to fix D bugs after the refinements. but not WizardCoder$_Q$. One major cause of S bugs is that indices in COBOL start at 1 as opposed to 0 in Java, leading to buggy Java where indices are shifted by 1. We observe that *LRM*1 is effective in repairing such bugs for both LLMs. Major L bugs are missing import statements for libraries like Scanner. For both LLMs, the refinement modules fix considerable bugs of this type. The specific error location that is provided as part of logic feedback aids in pinpointing and resolving bugs of types S and L. We observe M bugs for both LLMs for the cases where the response exceeds the context limit or does not adhere to the required format (code enclosed within <JAVA_CODE_BEGIN> and <JAVA_CODE_END> tags).

Table 3: (a) Distribution of 25 samples over bug types, analyzed module-wise for ChatGPT. D - Data; S - Syntax and Semantics; L - Logic and Dependency; M - Model; C - Logically Correct; (b) Effect of difficulty: EA (%) and RS for ChatGPT

Bug	TM	LRM1	RRM	LRM2
D	7	4	4	2
S	6	1	1	1
L	8	5	2	1
M	0	0	1	2
C	11	17	19	20

(a) Bugs

Bin	EA	RS
1	95.89	0.696
2	93.10	0.624
3	84.06	0.649
4	78.95	0.574
5	56.92	0.490

(b) Difficulty

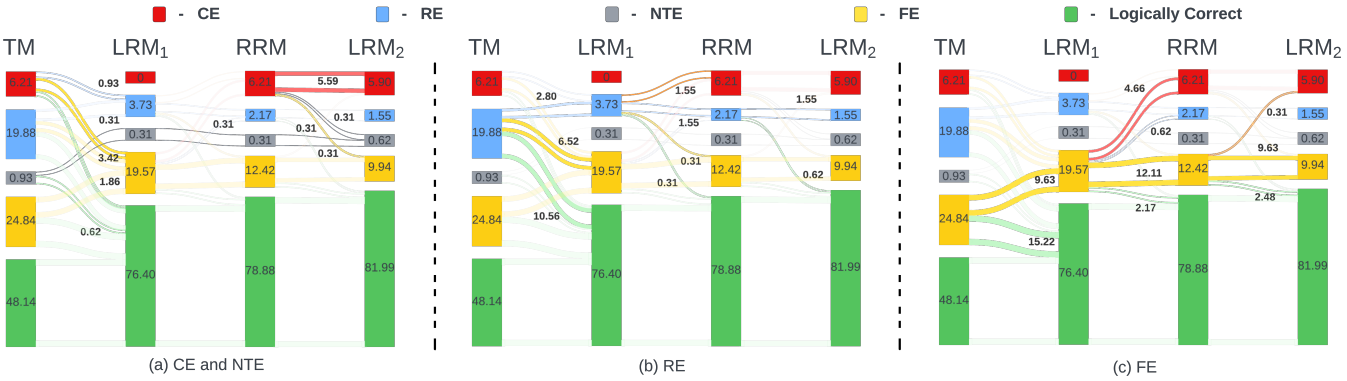

Figure 2: Contribution of each module towards correction of logical errors. Illustration for ChatGPT LLM.

Table 4: C_M, W_M C_C, W_C - Average Manual and ChatGPT evaluation scores for ChatGPT and WizardCoder$_Q$ translations. Co_C, Co_W - Correlation between ChatGPT scores and manual annotation for ChatGPT and WizardCoder$_Q$ translations

	C_C	C_M	W_C	W_M	Co_C	Co_W
Abstraction	3.68	4.8	3.96	4.3	0.36	0.21
Modularization	3.04	4.64	3.18	4.4	0.39	0.43
Logical segmentation	3.32	4.96	3.36	4.76	0.21	0.47
Redundant code	4.06	4.2	4.32	4.12	0.45	0.5

This is observed more for WizardCoder$_Q$, where *LRM2* causes a drastic increase in M bugs.

3.6.3 Code Difficulty. Table 3b and Github page Table 6 show that, for both LLMs, *EA* and *RS* decreases with increasing levels of difficulty (defined in Section 3.1).

3.6.4 Readability. In addition to the *RS* criteria (Section 3.2), we manually score (scale: 0-5) the resultant Java codes of our proposed workflow for the above-discussed 25 samples for the missing criteria, viz. abstraction, modularization, unnecessary code and logical segmentation. The average abstraction scores indicate good performance for capturing abstract codes with both ChatGPT and WizardCoder$_Q$ (Table 4). With ChatGPT, for 24 out of 25 samples, the resultant Java codes use libraries like *Scanner* for taking input, *Arrays* for sorting (functions like *Arrays.sort*()) wherever necessary or import *Math* functions. However, for one sample, it fails to import *Scanner*, uses hard-coded input and does not convert the array input and its processing (which has a longer, complex COBOL code) to abstract Java (example on Github page Table 8). For WizardCoder$_Q$, the Java codes do not import *Scanner* for 14 out of 25 samples and directly use *Scanner* or use hard-coded input. For 15 resultant Java codes ChatGPT and WizardCoder$_Q$ create functions optimally, yielding high scores for modularization, but for remaining samples, there is some scope for modularization. 12 and 17 samples for ChatGPT and WizardCoder$_Q$ do not have any unnecessary code (after eliminating the extra text). For both LLMs, we observe that the resultant Java codes are well indented, follow good logical segmentation, however, do not follow meaningful naming conventions but use source code variable names.

We further use a ChatGPT based evaluator to get scores for the above criteria (Github page prompt 16) for the same set of samples. The correlation of these scores with manual annotations (Table 4) for abstraction for WizadCoder$_Q$ is lower than ChatGPT. This is because, for few samples, WizadCoder$_Q$ generates extra text appended to the code (Github page Table 7) confusing the ChatGPT evaluator. The correlation for modularization is high for both the models. Lower correlation for logical segmentation is the result of ChatGPT evaluator scores not reflecting good logical segmentation of resultant Java codes and suggesting room for improvement without mentioning explicit reasons. Similarly, ChatGPT evaluator tends to give a low score for unnecessary code even with an explanation mentioning that no redundant code exists. It also considers *System.exit*(0) as unnecessary code. Overall, the low correlation between manual annotation and ChatGPT evaluator scores indicates a scope for improvement in LLM-based evaluation.

4 RELATED WORK

4.1 Code Translation

Converting codebases from legacy to modern languages is time-consuming and costly [27]. Most of the traditional rule-based approaches require knowledge of both source and target languages to design hand-crafted rules [21].To overcome this, ML models were proposed to learn the statistical alignments between programming languages [1, 15]. Approaches of Neural Machine Translation (NMT) are limited by the availability of parallel corpora [2, 8]. Transcoder [27] apply unsupervised NMT for languages with a small parallel corpora. Lachaux et al. [14] proposed a better pre-training mechanism specific to programming languages. Roziere et al. [28] use unit test cases to improve code translation performance. Recently, LLMs pre-trained with code such as Codex [7], and PaLM [9] have been used for code-translation. Athiwaratkun et al. [4] (MBXP) train their own LLMs and test their zero-shot and few-shot, monolingual and multilingual translation capabilities. Pan et al. [24] compare recent LLMs and code-LLMs for high-resource languages: starcoder [17], llama2 [32], codegen [22], codegeex [41], and GPT-4 [23] to show that GPT-4 performs the best. We emulate their taxonomy for our error and bug analysis in Section 3.6. Though there exist some rule-based works on COBOL to Java translation [31, 33] most of the latest neural and LLM-based approaches

are not tested for translation of low-resource legacy to modern languages, which is challenging as explained in Section 1.

4.2 Code Readability

Code readability is defined as the ease of understanding and maintaining [34], they evaluate readability based on meaningfulness of variable and function names, quality of comments, and the readability of algorithmic implementation. Initial works on readability extract features from code blocks and train a classifier with different types of features like structural (average number of parenthesis, identifiers etc.), entropy, and Halstread's volume [5, 25]. Dorn [10] defines a more extensive set of features based on four aspects: visual, spatial, alignment and linguistic. Scalabrino et al. [29] introduce additional textual features based on lexicon analysis. Above defined features might not be sufficient to estimate the ease of understanding. In addition, it needs high-level abstractions [30]. In our approach, we include abstraction and modularization aspects, which are very specific to and essential for translation from legacy to modern languages. Madaan et al. [20] propose a self-refinement framework with readability-specific feedback generated using the same LLM to improve code readability. As opposed to this, in our approach we explicitly specify translation specific readability criteria for more comprehensive feedback and refinement.

5 CONCLUSION

We propose an LLM-based workflow for automation of COBOL to Java translation. The workflow not only ensures the logical correctness of translated Java programs but also ensures readability. It exploits Java's high-resource nature, and performs refinement over translated Java code with compiler, execution and LLM-based feedback for logic and readability, without having the low-resource COBOL code in the context. Using a dataset derived from CodeNet, we empirically demonstrate the positive effect of introducing I/O test pairs, temperature sampling, and sequential logic and readability refinements of the translated Java, consistently across two distinct LLMs, viz. ChatGPT and WizardCoder$_Q$. We observe complementary benefits of these LLMs in terms of reducing errors and fixing bugs with temperature sampling and refinement, leading us to define a workflow using both the LLMs, which yields the best execution accuracy of 86.34% for COBOL to Java translation with readability score of 0.646. In future we plan to develop a hybrid solution combining the positive effects of syntax-based tools (logical correctness) and LLMs (better readability) for better performance.

REFERENCES

[1] Karan Aggarwal et al. 2015. *Using machine translation for converting python 2 to python 3 code.* Technical Report. PeerJ PrePrints.
[2] Wasi Uddin Ahmad et al. 2023. Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages. arXiv:2205.11116 [cs.CL]
[3] Naser Al Madi. 2022. How readable is model-generated code? examining readability and visual inspection of github copilot. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.
[4] Ben Athiwaratkun et al. 2022. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations*.
[5] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on software engineering* 36, 4 (2009), 546–558.
[6] Angelica Chen et al. 2023. Improving Code Generation by Training with Natural Language Feedback. arXiv:2303.16749 [cs.SE]
[7] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv e-prints* (2021), arXiv–2107.
[8] Xinyun Chen et al. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).
[9] Aakanksha Chowdhery et al. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL]
[10] Jonathan Dorn. 2012. A general software readability model. (2012).
[11] Jessica Ficler and Yoav Goldberg. 2017. Controlling Linguistic Style Aspects in Neural Language Generation. arXiv:1707.02633 [cs.CL]
[12] Jie Huang et al. 2023. Large Language Models Cannot Self-Correct Reasoning Yet. (2023). arXiv:2310.01798 [cs.CL]
[13] Xue Jiang et al. 2023. Self-planning Code Generation with Large Language Model. *arXiv e-prints* (2023), arXiv–2303.
[14] Marie-Anne Lachaux et al. 2021. DOBF: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems* 34 (2021), 14967–14979.
[15] Guillaume Lample et al. 2018. Phrase-Based & Neural Unsupervised Machine Translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 5039–5049.
[16] Jia Li et al. 2023. Structured Chain-of-Thought Prompting for Code Generation. arXiv:2305.06599 [cs.SE]
[17] Raymond Li et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]
[18] Xiping Liu and Zhao Tan. 2023. Divide and Prompt: Chain of Thought Prompting for Text-to-SQL. arXiv:2304.11556 [cs.CL]
[19] Ziyang Luo et al. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs.CL]
[20] Aman Madaan et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651* (2023).
[21] Aniketh Malyala et al. 2023. On ML-Based Program Translation: Perils and Promises. *arXiv preprint arXiv:2302.10812* (2023).
[22] Erik Nijkamp et al. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG]
[23] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[24] Rangeet Pan et al. 2023. Understanding the Effectiveness of Large Language Models in Code Translation. *arXiv e-prints* (2023), arXiv–2308.
[25] Daryl Posnett et al. 2011. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*. 73–82.
[26] Ruchir Puri et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.
[27] Baptiste Roziere et al. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020), 20601–20611.
[28] Baptiste Roziere et al. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773* (2021).
[29] Simone Scalabrino et al. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 6 (2018), e1958.
[30] Simone Scalabrino et al. 2019. Automatically assessing code understandability. *IEEE Transactions on Software Engineering* 47, 3 (2019), 595–613.
[31] Harry M Sneed and Katalin Erdoes. 2013. Migrating AS400-COBOL to Java: a report from the field. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 231–240.
[32] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]
[33] Aditya Trivedi and Ugrasen Suman. 2013. Design of a Reverse Engineering Model (A Case Study of COBOL to Java Migration). *International Journal of Computer Applications* 79, 5 (2013).
[34] Sergey Troshin and Nadezhda Chirkova. 2022. Probing Pretrained Models of Source Codes. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*. 371–383.
[35] Junjie Wang et al. 2023. Software Testing with Large Language Model: Survey, Landscape, and Vision. arXiv:2307.07221 [cs.SE]
[36] Xuezhi Wang et al. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. (2023). arXiv:2203.11171 [cs.CL]
[37] Jason Wei et al. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]
[38] Chunqiu Steven Xia et al. 2022. Practical Program Repair in the Era of Large Pre-trained Language Models. arXiv:2210.14179 [cs.SE]
[39] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. arXiv:2301.13246 [cs.SE]
[40] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. arXiv:2005.10636 [cs.SE]
[41] Qinkai Zheng et al. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. arXiv:2303.17568 [cs.LG]
[42] Yuqi Zhu et al. 2023. Improving Code Generation by Dynamic Temperature Sampling. *arXiv preprint arXiv:2309.02772* (2023).

Received 8 December 2023; accepted 15 January 2024