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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LLM4Code ’24, April 20, 2024, Lisbon, Portugal  
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 979-8-4007-0579-3/24/04  
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 and WizardCoder [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\(^1\) Section C.

\(^{1}\)gpt-3.5-turbo-0613 API endpoint https://platform.openai.com/docs/models/gpt-3-5

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 handwritten rules [21]. One such tool for COBOL to Java translation is

Quantized model https://huggingface.co/TheBloke/WizardCoder-15B-1.0-GPTQ

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.

<table>
<thead>
<tr>
<th>Source COBOL code</th>
<th>Syntax-directed approach</th>
<th>Direct LLM-based Translation</th>
<th>Our Workflow</th>
</tr>
</thead>
</table>
| \[
\ldots.
\] | \[
\ldots.
\] | \[
\ldots.
\] | \[
\ldots.
\] |
| ACCEPT INP;
PERFORM ablen TIMES
PERFORM VARYING j FROM cur BY 1
UNTIL INP(1) = SPACE
END-EXEC.
| 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) {
 processCmdLineArg(args);
 Abc_099_b instance_ = new Abc_099_b();
 instance_.execute(mall);
 System.exit(instance.getReturnCode());
}
public void execute(ProgramEnv _env) {
 initialize(this);
 doCobolGoToStart();
}......
| public class ABC_099_B {
 public static void main (String[] args) {
 \ldots.
 int cur = 1;
 String INP = scanner.nextLine();
 int i = 1;
 for (int k = 0; k < ablen; k++) {
 \ldots.
 i = i + 1;
 System.out.println(ab[k] + " ");
 \ldots.
}
 public static int[] readInput (Scanner scanner) {
 \ldots.
 String[] parts = input.split(\"\");
 ab[0] = Integer.parseInt(parts[0]);
 ab[1] = Integer.parseInt(parts[1]);
 return ab;
}
| \[
\ldots.
\] | \[
\ldots.
\] | \[
\ldots.
\] | \[
\ldots.
\] |

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 86.54% 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 Fan 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.

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

\(^4\)https://opencobol2java.sourceforge.net/
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+INSndef (Github page prompt 4) (v) TM_BASE+O/INS (Github page prompt 5) and (vi) TM_BASE+O/INSndef. 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+O/INSndef 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.

---

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

---

Prompt 1: TM_BASE+O/INSndef 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: (READABILITY_DEFINITIONS)
4 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, LRM1, RRM, LRM2, where LRM1 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 (\( J_{java} \)) using temperature sampling. Out of these K samples, the erroneous ones are passed to LRM1 to rectify the detected errors discussed above. The logically refined K samples (\( J_{javaL} \)) 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 (\( J_{javaL^*} \)) is forwarded to RRM. As logical correctness gets priority over readability, the Java Code generated as an output of RRM (\( J_{javaR} \)) is again checked for logical correctness. If it is logically incorrect and \( J_{javaL} \) is logically correct, we stick to \( J_{javaL} \) as the final output. Otherwise, if both \( J_{javaL} \) and \( J_{javaR} \) are logically incorrect, we feed \( J_{javaLR} \) to LRM2 to ensure logical correctness of the resulting Java code (\( J_{javaLR^*} \)).

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 provide 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 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], Posnet 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 task, we use 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 \((TM_{BASE+I/O+INS_{def}} \rightarrow (LM + RRM)_{trans})\) 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 \((TM_{BASE+I/O+INS_{def}} \rightarrow (LM + RRM)\_java)\). 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 \( LRM1 \) after \( TM \) itself (\( K = 5 \rightarrow 1 \)) as opposed to passing all 5 samples (output of \( TM \)) to \( LRM1 \) and then ranking and selecting the best sample (Section 2.4).

\(^{5}\)https://atcoder.jp/
\(^{6}\)https://onlinejudge.u-aizu.ac.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.

<table>
<thead>
<tr>
<th>Workflow</th>
<th>K</th>
<th>Exec. Acc. (%)</th>
<th>Read. Score (0-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baselines</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def}$ &amp; $LRM_{1} + RRMI_{trans}$ &amp; 5 &amp; 50.31 &amp; 76.71 &amp; 0.643 &amp; 0.643</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def}$ &amp; $LRM_{trans} \rightarrow RRMI_{trans} \rightarrow LRMI_{trans}$ &amp; 5 &amp; 51.86 &amp; 77.95 &amp; 0.601 &amp; 0.572</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def}$ &amp; $LRM + RRMI_{java}$ &amp; 5 &amp; 51.86 &amp; 75.78 &amp; <strong>0.673</strong> &amp; <strong>0.663</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def}$ &amp; $LRM_{1java} \rightarrow RRMI_{java} \rightarrow LRMI_{2java}$ &amp; 1 &amp; 28.88 &amp; 70.50 &amp; 0.569 &amp; 0.582</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Translation Module Ablation</strong></td>
<td>&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}$ &amp; &amp; 5 &amp; 19.88 &amp; 36.96 &amp; 0.605 &amp; 0.587</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+INS$ &amp; &amp; 5 &amp; 21.43 &amp; 45.34 &amp; 0.408 &amp; 0.412</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+INS_{def}$ &amp; &amp; 5 &amp; 18.94 &amp; 36.65 &amp; <strong>0.649</strong> &amp; 0.262</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O$ &amp; &amp; 1 &amp; 22.05 &amp; 47.83 &amp; 0.303 &amp; 0.459</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O^*$ &amp; &amp; 13 &amp; <strong>64.29</strong> &amp; <strong>72.67</strong> &amp; 0.585 &amp; <strong>0.624</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS$ &amp; &amp; 5 &amp; 45.03 &amp; 63.66 &amp; 0.600 &amp; 0.536</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def}$ &amp; &amp; 1 &amp; 25.47 &amp; 44.72 &amp; <strong>0.525</strong> &amp; <strong>0.471</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+CoT$ &amp; &amp; 5 &amp; 49.07 &amp; 60.25 &amp; 0.614 &amp; 0.539</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def}+CoT$ &amp; &amp; 1 &amp; 9.63 &amp; 48.45 &amp; 0.525 &amp; 0.467</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Refinement Modules Ablation</strong></td>
<td>&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O \rightarrow LRM_{1java}$ &amp; &amp; 5 &amp; 50.93 &amp; 75.47 &amp; 0.559 &amp; 0.495</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O \rightarrow LRM_{1java} \rightarrow RRMI_{java}$ &amp; &amp; 5 &amp; 51.24 &amp; 77.95 &amp; 0.603 &amp; 0.547</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O \rightarrow LRM_{1java} \rightarrow RRMI_{java} \rightarrow LRMI_{2java}^*$ &amp; &amp; 5 &amp; 51.24 &amp; <strong>79.81</strong> &amp; 0.571 &amp; 0.547</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def} \rightarrow LRM_{1java}$ &amp; &amp; 5 &amp; 52.80 &amp; 76.40 &amp; 0.602 &amp; 0.569</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def} \rightarrow LRM_{1java} \rightarrow RRMI_{java}$ &amp; &amp; 5 &amp; <strong>53.42</strong> &amp; 78.88 &amp; 0.645 &amp; <strong>0.609</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def} \rightarrow LRM_{1java} \rightarrow RRMI_{java} \rightarrow LRMI_{2java} (w/o I/O)^*$ &amp; &amp; 5 &amp; 51.24 &amp; 79.50 &amp; <strong>0.652</strong> &amp; 0.594</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Proposed</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$TM_{BASE}+1/O+INS_{def}$ &amp; $LRM_{1java} \rightarrow RRMI_{java} \rightarrow LRMI_{2java}^*$ &amp; 5 &amp; <strong>53.42</strong> &amp; <strong>81.99</strong> &amp; <strong>0.617</strong> &amp; <strong>0.610</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.4.4 Direct translation of COBOL to logically correct and readable Java. We treat one of our ablations, $TM_{BASE}+1/O+INS_{def}$, 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 perform better 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; WizardCoderQ≈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 WizardCoderQ 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 ($TM_{BASE}+1/O+INS$) and elaborate instructions ($TM_{BASE}+1/O+INS_{def}$) shows improvement in EA over BASE + 1/O for WizardCoderQ but not for ChatGPT. Hence, we execute our workflow with both best-performing configurations of $TM_{BASE}+1/O+INS_{def}$ for WizardCoderQ and $TM_{BASE}+1/O$ for ChatGPT. Moreover, RRMI not only improves RS, but also EA for both LLMs ($TM_{BASE}+1/O+INS_{def}$ → $LRMI_{java}$ → $RRMI_{java}$ over $TM_{BASE}+1/O+INS_{def}$ → $LRMI_{java}$) 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 ($TM_{BASE}+1/O+INS_{def}+CoT$ and $TM_{BASE}+1/O+CoT$) 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 $TM_{BASE+1/O+INS_{col}}$ 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 WizardCoder$_Q$ 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. LRM1 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 in 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 LRM1 ($java_{L*}$) (section 2.4).

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. LRM2 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 $RE$. LRM1 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 LRM1 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 \langle JAVA_CODE\_BEGIN\rangle and \langle JAVA_CODE\_END\rangle tags).

<table>
<thead>
<tr>
<th>Bug</th>
<th>TM</th>
<th>LRM1</th>
<th>RRM</th>
<th>LRM2</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>S</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>

(a) Bugs

<table>
<thead>
<tr>
<th>Bin</th>
<th>EA</th>
<th>RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95.89</td>
<td>0.696</td>
</tr>
<tr>
<td>2</td>
<td>93.10</td>
<td>0.624</td>
</tr>
<tr>
<td>3</td>
<td>84.06</td>
<td>0.649</td>
</tr>
<tr>
<td>4</td>
<td>78.95</td>
<td>0.574</td>
</tr>
<tr>
<td>5</td>
<td>56.92</td>
<td>0.490</td>
</tr>
</tbody>
</table>

(b) Difficulty

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

<table>
<thead>
<tr>
<th></th>
<th>$C_C$</th>
<th>$C_M$</th>
<th>$W_C$</th>
<th>$W_M$</th>
<th>$C_0$</th>
<th>$C_W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>3.68</td>
<td>4.8</td>
<td>3.96</td>
<td>4.3</td>
<td>0.36</td>
<td>0.21</td>
</tr>
<tr>
<td>Modularization</td>
<td>3.04</td>
<td>4.64</td>
<td>3.18</td>
<td>4.4</td>
<td>0.39</td>
<td>0.43</td>
</tr>
<tr>
<td>Logical segmentation</td>
<td>3.32</td>
<td>4.96</td>
<td>3.36</td>
<td>4.76</td>
<td>0.21</td>
<td>0.47</td>
</tr>
<tr>
<td>Redundant code</td>
<td>4.06</td>
<td>4.2</td>
<td>4.32</td>
<td>4.12</td>
<td>0.45</td>
<td>0.5</td>
</tr>
</tbody>
</table>

This is observed more for WizardCoderQ, where LRMM2 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 WizardCoderQ (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 WizardCoderQ, 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 WizardCoderQ 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 WizardCoderQ 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 WizardCoderQ is lower than ChatGPT. This is because, for few samples, WizardCoderQ 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. Athiwaratkn 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 Halstead’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. 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

Received 8 December 2023; accepted 15 January 2024

Gandhi et al.