CS 846: ATRE by Prof. Daniel Berry

REQUIREMENTS ENGINEERING FOR ML-BASED SOURCE CODE TRANSLATION

July 13, 2023

Presented by:

Prithwish Jana
PhD Student,
David R. Cheriton School of Computer Science
: prithwish.jana@uwaterloo.ca

UNIVERSITY OF WATERLOO

The Problem Statement

The Task: Java↔Python Code Translation

```java
import java.io.*;
public class Main
{
    static int unitDigitXRaisedY(int x, int y)
    {
        int res = 1;
        for (int i = 0; i < y; i++) res = (res * x) % 10;
        return res;
    }
    public static void main(String args[])
    {
        System.out.println(unitDigitXRaisedY(4, 2));
    }
}
```

A **Java** program

Translation of a code from one high-level language to another

```python
def unitDigitXRaisedY(x, y):
    res = 1
    for i in range(y):
        res = (res * x) % 10
    return res
if __name__ == '__main__':
    print (unitDigitXRaisedY(4 ,2))
```

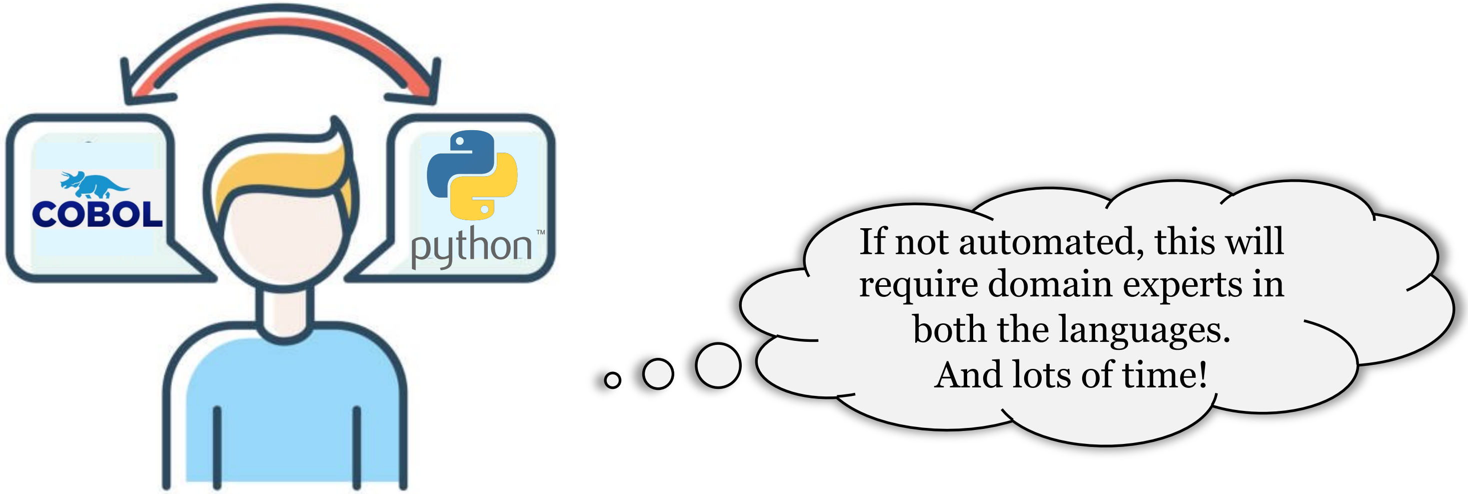
An "*equivalent*" program in **Python**

- Let ***S* = source language** (e.g., Java) and ***T* = target language** (e.g., Python)
- **Code Translation Learning Problem**
  - To *learn* a language translator f_{ST}: $S \rightarrow T$, which when provided with a ***S***-program produces a ***T***-program, that is runtime (input-output) equivalent to the former.
  - In this work, focussing on translation between Java & Python

UNIVERSITY OF WATERLOO

Motivation: Significance of Automated Code Translation

- Automatic translation of code from one high-level language to another
  - An important software engineering research area
  - Large legacy codebase getting transformed to a modern language e.g., COBOL to Python



  - Applications in code migration[1] and cross-platform interoperability[2]

[1] B. G. Mateus, M. Martinez, and C. Kolski, "Learning Migration Models for Supporting Incremental Language Migrations of Software Applications," Information and Software Technology, vol. 153, 2023.
[2] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, M. Luján, and H. Mössenböck, "Cross-language Interoperability in a Multi-language Runtime," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 40, no. 2, pp. 1–43, 2018

Challenges in Java↔Python Translation

- Java is a **statically-typed** language
  - For statically-typed language, type of variables known at compile-time
  - All kinds of checking can be done by compiler → a lot of trivial bugs caught at an early stage
- Python is **dynamically-typed** language
  - Interpreted language; Type is associated with run-time values, not named variables
- Java and Python belong to same programming paradigm i.e., OOP
  - But **huge differences in syntax** and **programming style**

JavaScript & Java have C-like syntax

Python has a different syntax (e.g., tabs for nesting)

```
//JavaScript
class Rectangle {
    #height, #width;
    constructor(height, width) {
        this.#height = height;
        this.#width = width;
        console.log("Created");
    }
}
```

```
//Java
class Rectangle {
    private int height, width;
    public Rectangle (int height, int width) {
        this.height = height;
        this.width = width;
        System.out.println("Created");
    }
}
```

```
#Python
class Rectangle(object):
    def __init__(self, height, width):
        self.__width = width
        self.__height = height
        print("Created")
```

Easier to translate

Difficult to translate

The Earlier Approaches

Early Rule-Based Transpilers

- Source-to-source translator / Transcompiler / Transpiler [3][4][5]
  - Rule-based & handcrafted → thus, **quite expensive** to build
  - Uses traditional concepts such as parsing and abstract syntax trees
  - Vary by the intricacies and difficulty level of constructs that it can handle
  - Long list of equivalences between the two languages → **translation requirements**

- Abstract Methods
- Anonymous Inner Classes
- Arrays
- Basic Syntax Differences
- Casting
- Collections
- Comments
- Constants, Fields, and Local Variables
- Constructors
- Default Parameters
- Enums
- Equality
- Exception Handling and Try-With-Resources
- •••
- Ternary Conditional Operator
- Type Discovery

If-Else

Java	Python

if (conditionA)
{
}
else if (conditionB)
{
}
else
{
}
	if conditionA:
	    pass
	elif conditionB:
	    pass
	else:
	    pass

Strings

Java	Python
String s = initValue;	s = initValue
int i = s.indexOf(y);	i = s.find(y)
i = s.lastIndexOf(y);	i = s.rfind(y)
i = s.length();	i = len(s)
boolean b = s.contains(y);	b = y in s
s = s.substring(i);	s = s[i:]
s = s.substring(i, j);	s = s[i:j]
b = s.endsWith(y);	b = s.endswith(y)
b = s.startsWith(y);	b = s.startswith(y)
s = s.toLowerCase();	s = s.casefold()
s = s.toUpperCase();	s = s.upper()
s = s.stripLeading();	s = s.lstrip()
s = s.stripTrailing();	s = s.rstrip()

Java vs Python 'import'

Java	Python
import foo.*;	from foo import *
import ack.Bar;	from ack import Bar

Casting

Java	Python

void casts()
{
    x = (int)y;
    x = (float)y;
    x = (String)y;
}
	def casts(self):
	    x = int(y)
	    x = float(y)
	    x = str(y)

[3] T. Melhase et al., "java2python: Simple but Effective Tool to Translate Java Source Code into Python." https://github.com/natural/java2python.
[4] "py2java: Python to Java Language Translator." https://pypi.org/project/py2java/
[5] T. S. Solutions, "The Most Accurate and Reliable Source Code Converters." https://www.tangiblesoftwaresolutions.com/.

EARLY RULE-BASED TRANSPILERS (CONTD...)

- Source-to-source translator / Transcompiler / Transpiler
  - Handcrafting exhaustive set of rules → **too tedious**, too many requirements to satisfy
  - There's **no guarantee** that **translating piece-by-piece** will generate something 'meaningful'
    - 'meaningful' → a piece of code that works

So, certain requirements are necessary to be chalked out

Many such tools come with a disclaimer stating limitations:
"*...translated codes **should not be expected** to compile and run readily.*" [4]

[4] "py2java: Python to Java Language Translator." https://pypi.org/project/py2java/

The Proposed Method
From a Requirements Engineering Perspective

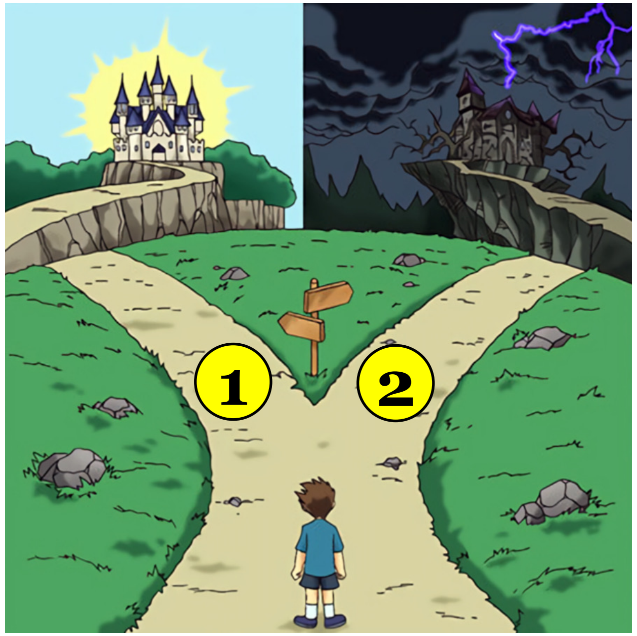
PROGRAM TRANSLATION: TOP-LEVEL BASIC REQUIREMENTS

```java
import java.io.*;
public class Main
{
    static int unitDigitXRaisedY(int x, int y)
    {
        int res = 1;
        for (int i = 0; i < y; i++) res = (res * x) % 10;
        return res;
    }
    public static void main(String args[])
    {
        System.out.println(unitDigitXRaisedY(4, 2));
    }
}
```

A **Java** code

```python
def unitDigitXRaisedY(x, y):
    res = 1
    for i in range(y):
        res = (res * x) % 10
    return res
if __name__ == '__main__':
    print (unitDigitXRaisedY(4 ,2))
```

An "equivalent" code in **Python**



Two top-level requirements:

1 The target-language code should be *syntactically correct*

(We just need a **T**-compiler, and the code should compile)

Easy to chalk out requirements ☺

2 The target-language code should be *runtime (input-output) equivalent* to the source-language code

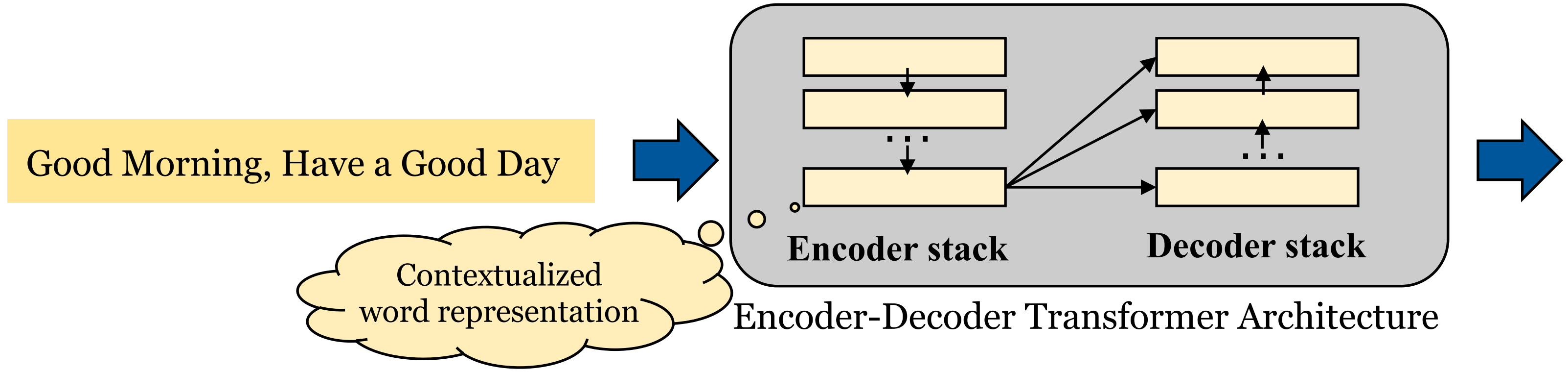
(Given the same set of console inputs or no input, the outputs are the same)

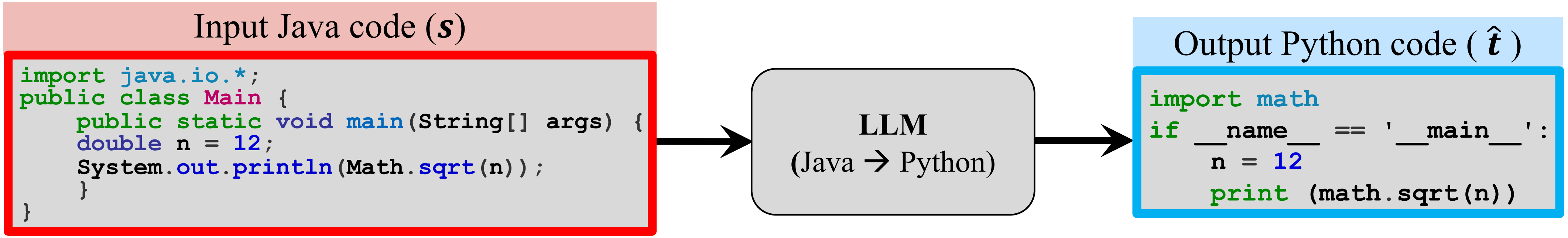
Difficult ☹

UNIVERSITY OF WATERLOO

Program Translation: Implementation Requirements

- Recently, **Large Language Models (LLMs)**[6] revolutionized **Natural Language translation**





- Here, we train LLMs for **Programming Language translation**
- Basically, the LLM will serve as the translator function f_{ST}: $\boldsymbol{S} \rightarrow \boldsymbol{T}$

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," Advances in Neural Information Processing Systems (NeurIPS), vol. 30, 2017.

Requirements for Train-Corpus

TRAIN-CORPUS (C) REQUIREMENTS

- Like any supervised learning set-up, we need data!
  - Pairs of equivalent Java & Python codes, mined from various competitive coding websites

Java Code	Equivalent Python Code
public class Improve { static int calculateSquareSum (int n) { if (n <= 0) return 0 ; int fibo [ ] = new int ...	def calculateSquareSum (n) : NEW_LINE INDENT fibo = [0] * (n + 1) NEW_LINE if (n <= 0) : NEW_LINE ...
import java . util . * ; public class GFG { static int val (char c) { if (c >= '0' && c <= '9') return (int)...	def val (c) : NEW_LINE INDENT if (c >= '0' and c <= '9') : NEW_LINE INDENT return int (c) NEW_LINE DEDENT ...
...	...

Data Sources:

Project Euler .net, GeeksforGeeks, CODEFORCES, code jam, LeetCode, AtCoder

C.R1 There should be **at least 30,000 pairs** of Java & Python codes (higher the better)
- That's the standard to train a transformer-based LLM architecture

C.R2 For all pairs, **both** the Java & Python codes should be **syntactically correct**

C.R3 For all pairs, **both** the Java & Python codes should be **runtime (input-output) equivalent**
- Can be verified from the set of stringent test-cases in the corresponding data source

C.R4 For all pairs, the Java & Python codes **should not be too long** i.e., **at most 512 tokens**
- LLMs accept and produce tokenized sequences (each word ≡ 1 or more token IDs), which have an upper-limit for length

Train-Corpus (C) Requirements (Contd...)

Java Code	Equivalent Python Code
`public class Improve { static int calculateSquareSum ( int n ) { if ( n <= 0 ) return 0 ; int fibo [ ] = new int ...`	`def calculateSquareSum ( n ) : NEW_LINE INDENT fibo = [ 0 ] * ( n + 1 ) NEW_LINE if ( n <= 0 ) : NEW_LINE ...`
`import java . util . * ; public class GFG { static int val ( char c ) { if ( c >= '0' && c <= '9' ) return ( int )...`	`def val ( c ) : NEW_LINE INDENT if ( c >= '0' and c <= '9' ) : NEW_LINE INDENT return int ( c ) NEW_LINE DEDENT ...`
...	...

Data Sources:

Project Euler .net, GeeksforGeeks, Codeforces, code jam, LeetCode, AtCoder

C.R5 For each **source-language feature**, that we want the trained LLM to learn to translate, the **number of examples (pairs)** in the dataset should count to **at least 1% of total corpus**

E.g.,

- Arrays
- Basic Syntax Differences
- Casting
- Collections
- Comments

- Increment and Decrement Operators
- Inheritance
- Interfaces
- Java vs Python 'import'
- Lambdas
- Logical and Bitwise Operators

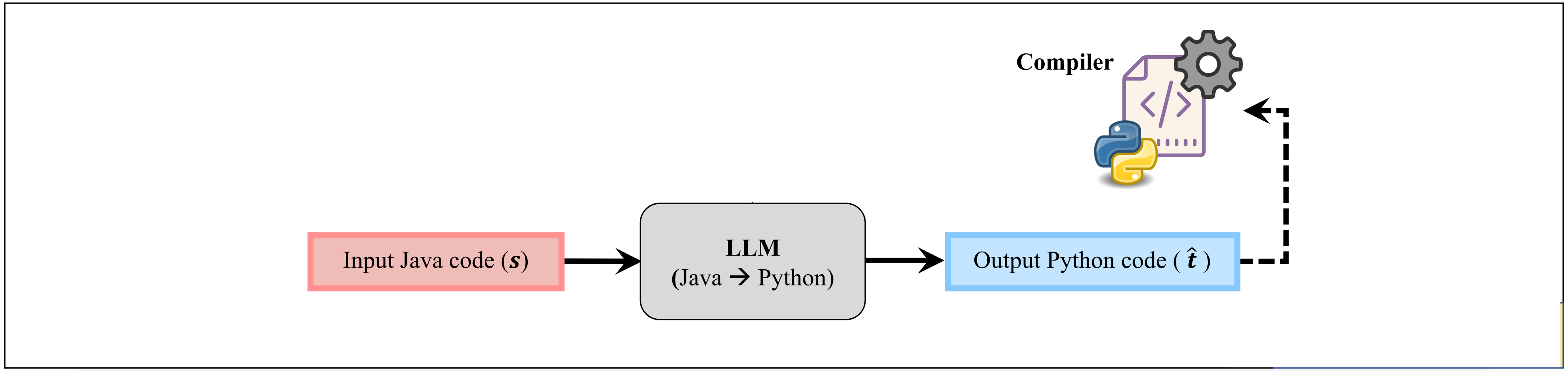
...

This ensures the training corpus is *representative enough of the real-world*

UNIVERSITY OF WATERLOO

Compilation Requirements for the Learned Large Language Model (LLM)

Learned LLM (L) Requirement: Compilation



Top-level Requirement I:

The translated target-language (T) code should be *syntactically correct*

L.R1 The output target-language code ($\hat{t}$) should pass error-free by **T**-compiler

- For Java as **T**, we use **`javac`** compiler
- For Python as **T**, we use the **`pylint`**[7] **static code analyzer** (as Python is an interpreted language**)**

[7] "pylint: Python code Static Checker." https://pypi.org/project/pylint/

Runtime Equivalence Requirements for the Learned Large Language Model (LLM)

How To Check Runtime Equivalence?

Top-level Requirement II:

Target-language (T) code should be *runtime (IO) equivalent* to source-language (S) code

Three major ways to check equivalence:

- Manually writing an exhaustive suite of test-cases
  - **Too tedious**, **chances of missing** essential test cases
- Boundary-value analysis
  - Tests at boundaries between partitions of input values → **low test coverages**
- Fuzzing / Fuzz-Testing
  - Injects random invalid inputs into a system → **rely on pure luck to find bugs**

- Instead, perform **automated unit-testing** of individual functions
- There is Symflower for Java, but no such popular tool for Python

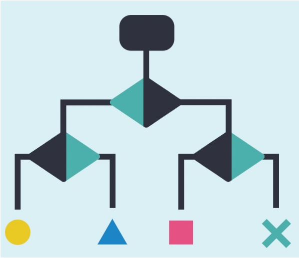
UNIVERSITY OF WATERLOO

How To Check Runtime Equivalence? Exhaustive Unit-Testing

Top-level Requirement II:

Target-language (T) code should be *runtime (IO) equivalent* to source-language (S) code

- Solver-based analysis through Symflower
  - Automated tool to generate JUnit tests for each function in a code
  - Support for Java only, not Python
  - Symbolic execution **computes necessary inputs** for a function to **execute all relevant paths (exhaustive)** in its **control-flow graph**
  - Generates **J-Unit tests** for all functions in a Java code



```
public class Main
{
    static int minLettersNeeded(int n)
    {
        if (n % 26 == 0) return (n / 26);
        else return ((n / 26) + 1);
    }
    public static void main(String args[])
    {
        int n = 52;
        System.out.println(minLettersNeeded(n));
    }
}
```

A Java code

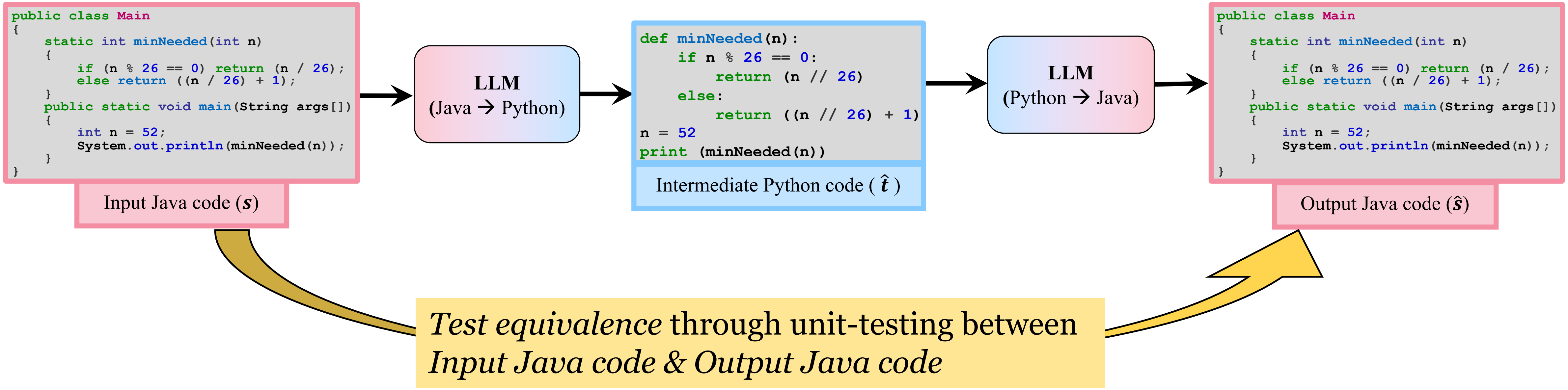
```
@Test
assertEquals(Main.minLettersNeeded(0), 0)

@Test
assertEquals(Main.minLettersNeeded(1), 2)
```

J-Unit tests for a function

Ensuring Runtime Equivalence: Exhaustive Unit-Testing

- As we said, solver-based analysis through *Symflower*: applicable only for Java
- So, employ two back-to-back LLMs: Java → Python and Python → Java

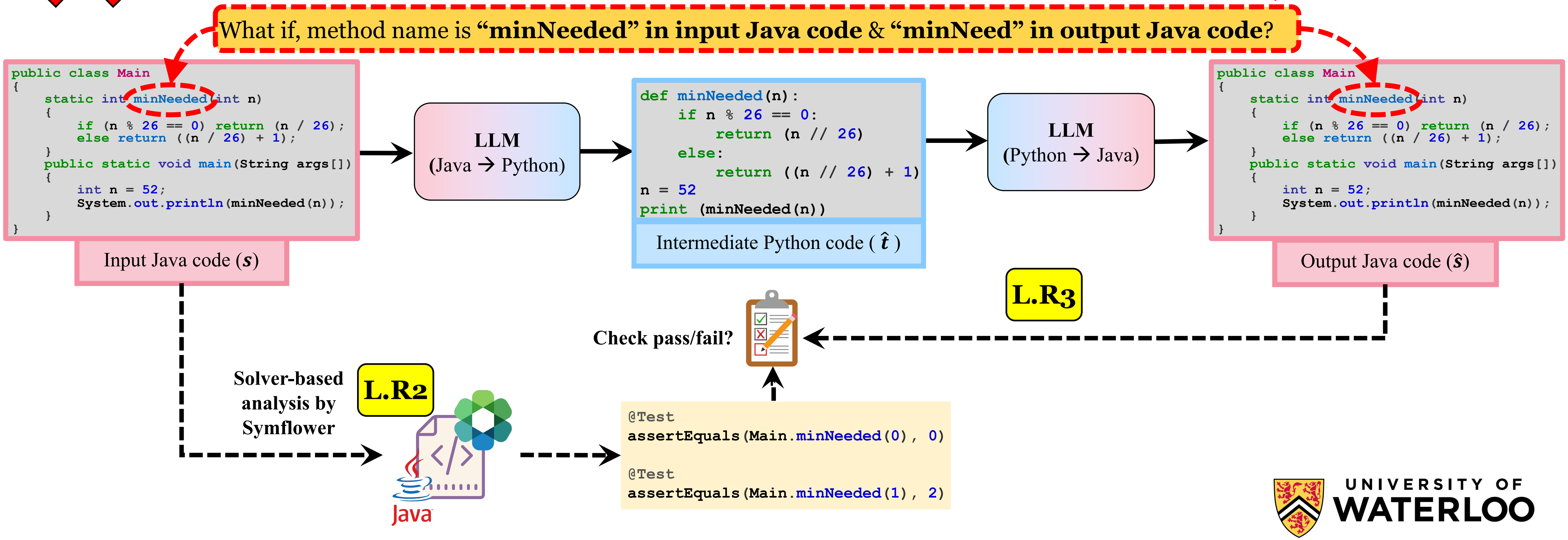


Learned LLM (L) Requirement: Runtime Equivalence

L.R2 For each method $\boldsymbol{p}$ of input Java code ($\boldsymbol{s}$), Symflower generates J-Unit tests $\{u_p\}$

~~**L.R3**~~ All $\{u_p\}$ should pass, on the method $\boldsymbol{p}$ of output Java code ($\hat{\boldsymbol{s}}$)

- Even though input & output Java codes equivalent, this requirement will fail
- Need to **relax the requirement**

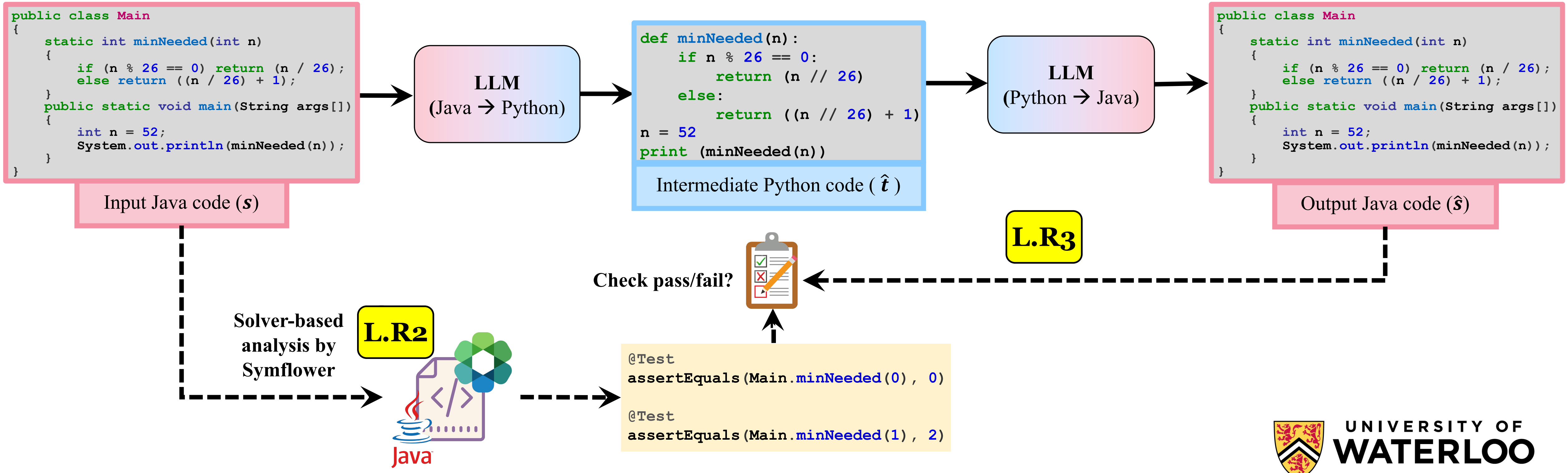


Learned LLM (L) Requirement: Runtime Equivalence (Relaxing L.R3)

L.R2 For each method p of input Java code (s), Symflower generates J-Unit tests $\{u_p\}$

L.R3 All $\{u_p\}$ should pass, on method p^* of output Java code ($\hat{s}$), where $p^* = \text{argmax}_{p^* \in \hat{s}}\, \text{JaccardSimilarity}(p, p^*)$

Another issue: What if the methods p, p^* do not return anything? They just print something on console



Learned LLM (L) Requirement: Runtime Equivalence (Console Outputs)

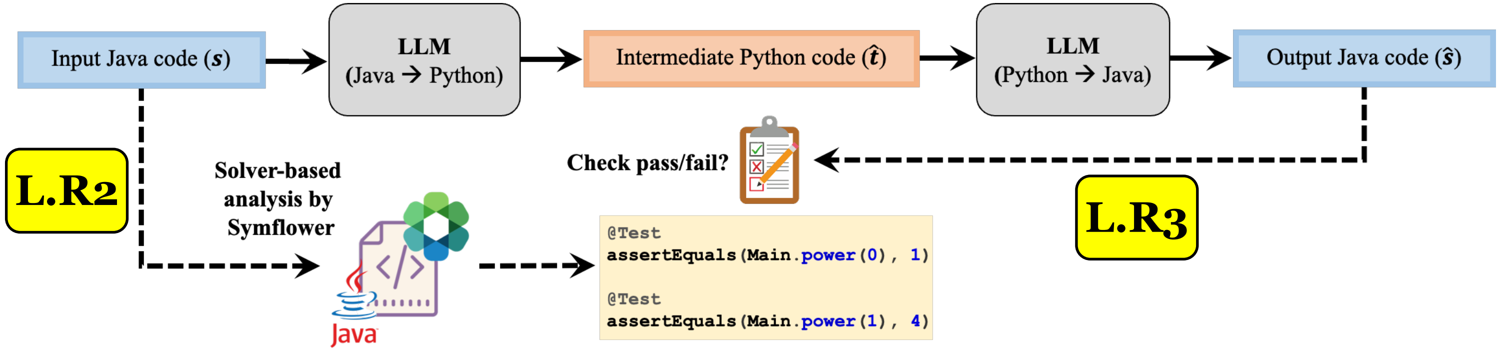
L.R2 For each method $\boldsymbol{p}$ of input Java code ($\boldsymbol{s}$), Symflower generates J-Unit tests $\{u_{\boldsymbol{p}}\}$

L.R3 All $\{u_{\boldsymbol{p}}\}$ should pass, on method $\boldsymbol{p}^*$ of output Java code ($\hat{\boldsymbol{s}}$), where $\boldsymbol{p}^* = \text{argmax}_{\boldsymbol{p}^* \in \hat{\boldsymbol{s}}}\, \text{JaccardSimilarity}(\boldsymbol{p}, \boldsymbol{p}^*)$

L.R4 Let $\boldsymbol{p}_{out}$ and $\boldsymbol{p}^*_{out}$ be the respective console outputs → $f_{matched}(\boldsymbol{p}_{out}, \boldsymbol{p}^*_{out})$ should be True
Here, $f_{matched}$ is a string-matching function that is:

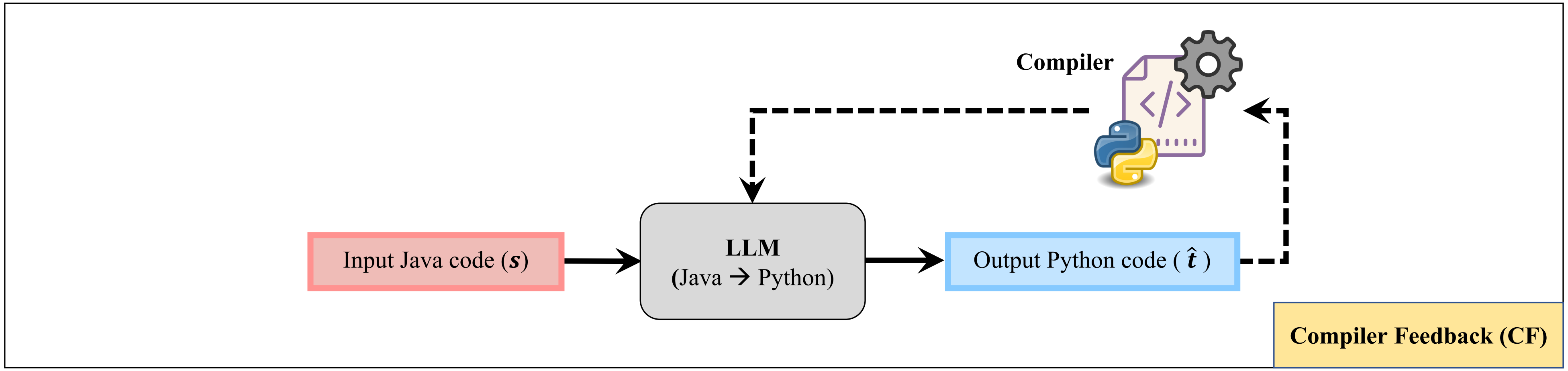
- case-insensitive → $f_{matched}$(“`CS846-ATRE`”, “`cs846-atre`”) = True
- ignores whitespaces → $f_{matched}$(“`good morning`”, “`goodmorning`”) = True
- disregards punctuations (only when they are not a major portion of the output) → $f_{matched}$(“`Hi! Bro.`”, “`Hi Bro`”) = True
- takes numeric or floating-point values to a common representation → $f_{matched}$(“`3.1415`”, “`3.1`”) = True

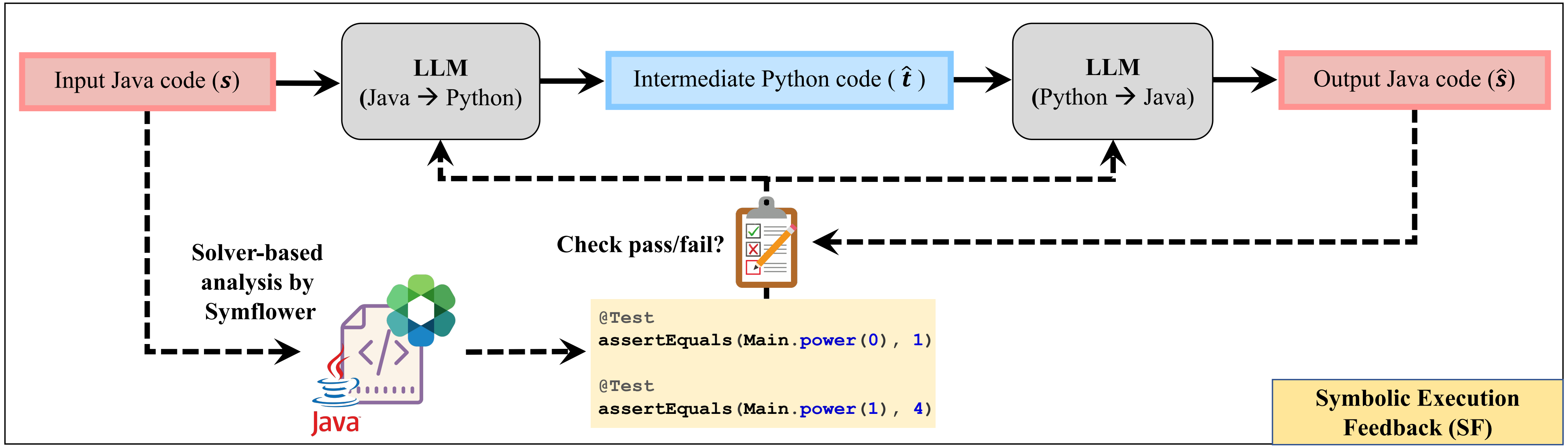
These are because: to evaluate code equivalence, we do not need a strict string-matching function



UNIVERSITY OF WATERLOO

How to Train the LLM to Follow such Requirements?



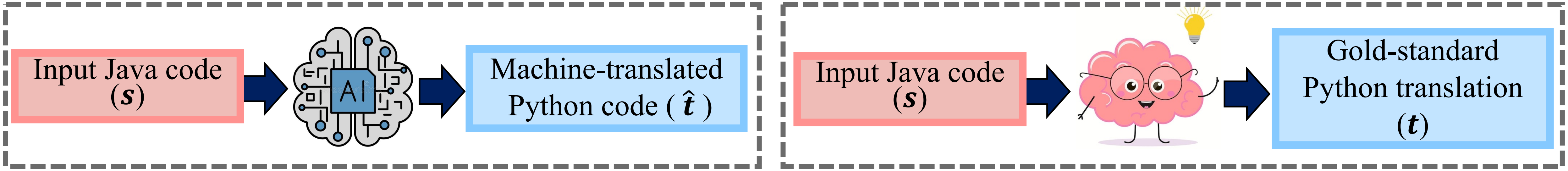


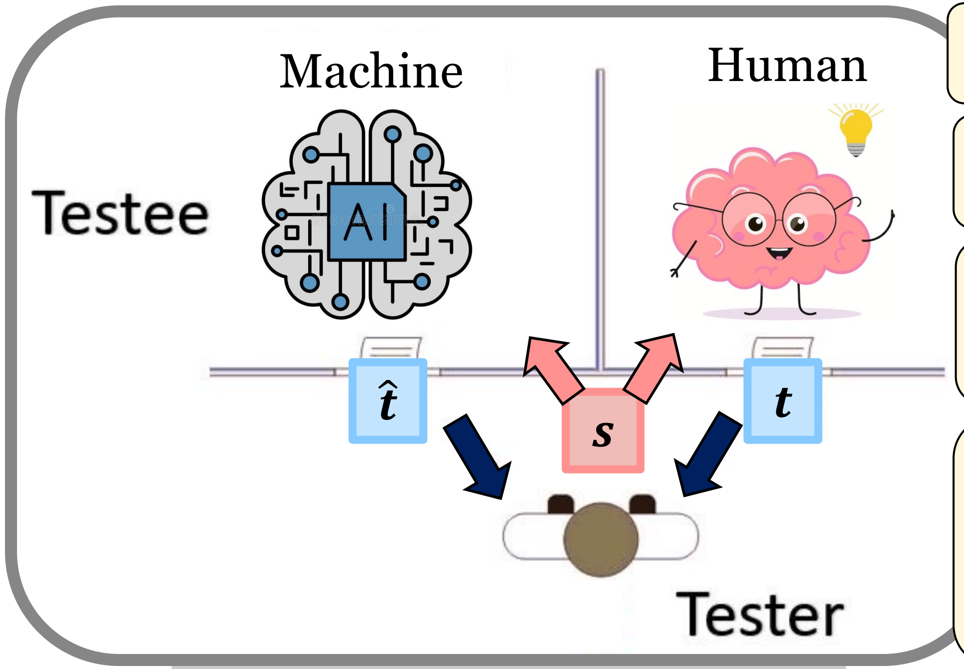
- During training with cross-entropy loss, **provide feedbacks to the LLM**: whether requirements satisfied or not?
- **Compiler Feedback (CF)** increases compilation rate of output code
- **Symbolic Execution Feedback (SF)** increases the runtime equivalence rate of output code

UNIVERSITY OF WATERLOO

How To Know That The LLM is Trained Well-Enough?

Evaluation Metrics



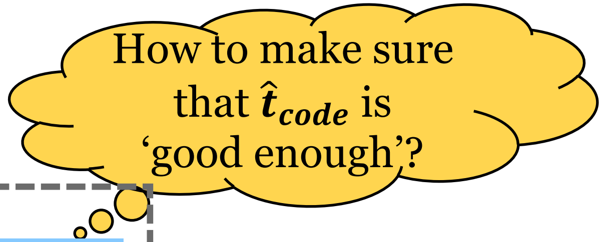


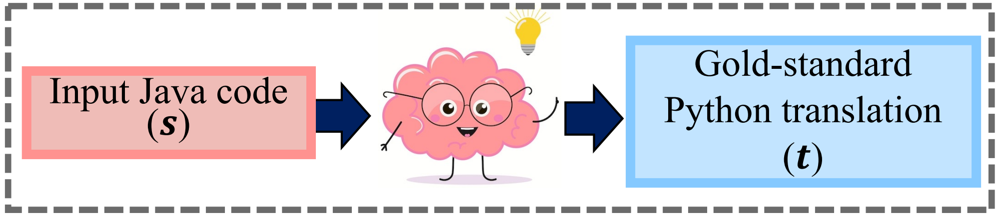
Code-Translation Turing Test

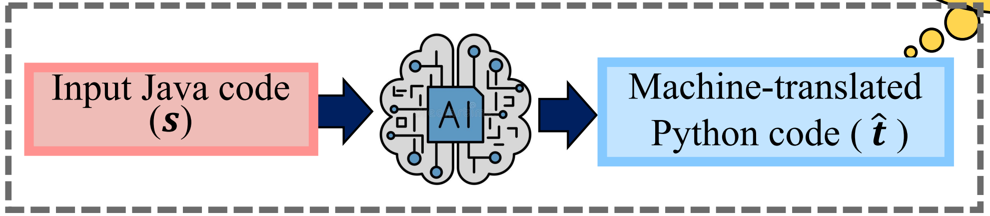
1. Tester unaware about who is behind which window
2. Tester submits Input Java code s through the windows
3. Machine / AI and Human returns Python codes $\hat{t}$ and t resp.
4. If Tester finds both translations to be equally 'good enough', the machine is successful in fooling the Tester

- **Qualitatively:** Have to make sure that our LLM should perform well in Turing Test
- To evaluate **quantitatively** whether it satisfies the basic requirements, we need some **metrics**

Traditional Evaluation Metrics







- **ExactMatch**
  - a Boolean score based on perfect match

$$\mathbf{EM}(t_{code}, \hat{t}_{code}) = \begin{cases} 1, & \text{if } t_{code} = \hat{t}_{code} \\ 0, & \text{otherwise} \end{cases}$$

- **BLEU** → *Bilingual Evaluation Understudy*
  - computes 'closeness' with gold-standard translation through n-gram overlaps; penalizes short predictions

$$\mathbf{BLEU}(t_{code}, \hat{t}_{code}) = \text{BrevityPenalty} + \text{GeometricAvgPrecision.}$$
$$= \min\left(1 - \frac{|t_{code}|}{|\hat{t}_{code}|}, 0\right) + \left(\frac{p_1 + p_2 + p_3 + p_4}{4}\right)$$

p_1 (1-gram precision) $= \frac{5}{8}$

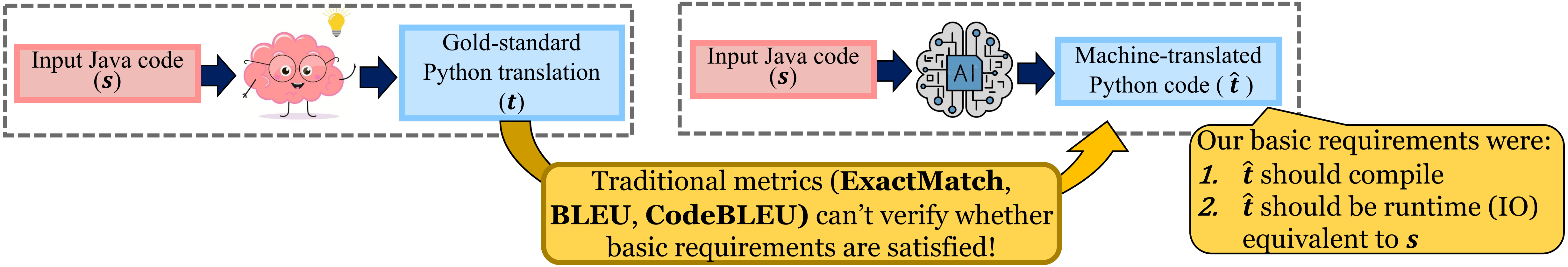
p_2 (2-gram precision) $= \frac{4}{7}$

- **CodeBLEU** → *BLEU, extended for codes*
  - checks closeness + syntactic & semantic features
  - Mean of BLEU, weighted n-gram match (**WM**), syntactic AST match (**SM**) & semantic Data-flow match (**DM**)

$$\mathbf{CodeBLEU}(t_{code}, \hat{t}_{code}) = \left(\frac{\mathbf{BLEU} + \mathbf{WM} + \mathbf{SM} + \mathbf{DM}}{4}\right)$$

WM: BLEU, where keywords (*for*, *int*, *public*, etc.) have higher weights
SM: %age of sub-tree matches in Abstract Syntax Tree of t_{code} and $\hat{t}_{code}$
DM: %age of sub-graph matches in Data Flow Graph of t_{code} and $\hat{t}_{code}$

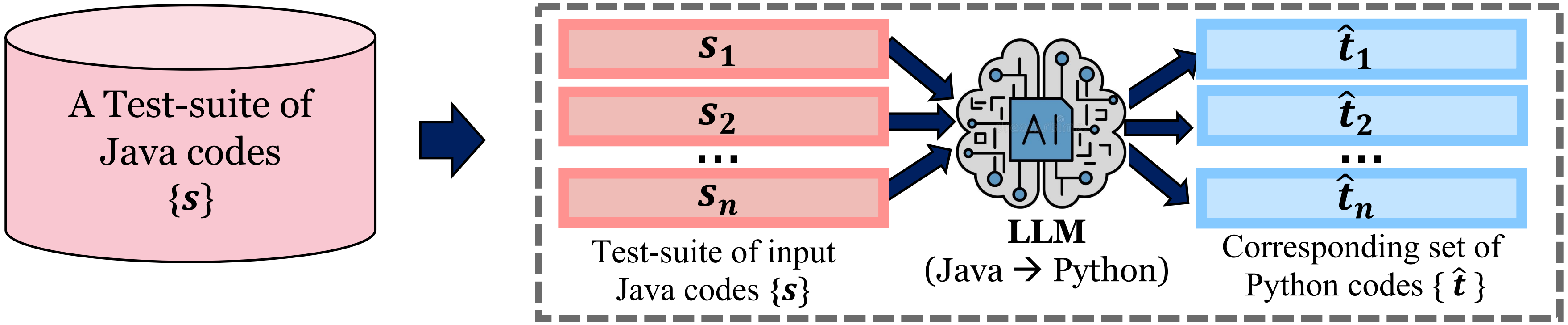
Traditional Evaluation Metrics: Not Suitable for Our Requirements



- **ExactMatch**
  - Doesn't make much sense for Code Translation. Too strict, there can be multiple correct translations
  - Low **ExactMatch**$(\boldsymbol{t_{code}}, \hat{\boldsymbol{t}}_{\boldsymbol{code}})$ score $\not\Rightarrow$ $\hat{\boldsymbol{t}}_{\boldsymbol{code}}$ doesn't satisfy basic requirements
  - E.g. $\hat{\boldsymbol{t}}_{\boldsymbol{code}}$ = `print("Hello" + "!")` satisfies requirements of $\boldsymbol{t_{code}}$ = `print("Hello!")`. Still, **ExactMatch**$(\boldsymbol{t_{code}}, \hat{\boldsymbol{t}}_{\boldsymbol{code}}) = 0$
- **BLEU**
  - More relevant for Natural Language translation (e.g. English to French)
  - Computes 'closeness' of $\hat{\boldsymbol{t}}_{\boldsymbol{code}}$ and $\boldsymbol{t_{code}}$. High **BLEU**$(\boldsymbol{t_{code}}, \hat{\boldsymbol{t}}_{\boldsymbol{code}})$ score $\not\Rightarrow$ $\hat{\boldsymbol{t}}_{\boldsymbol{code}}$ compiles
  - E.g. $\boldsymbol{t_{code}}$ = `print("Hello!")` compiles but, $\hat{\boldsymbol{t}}_{\boldsymbol{code}}$ = `print(Hello!")` doesn't. Still, **BLEU**$(\boldsymbol{t_{code}}, \hat{\boldsymbol{t}}_{\boldsymbol{code}}) \cong 100\%$
- **CodeBLEU**
  - Computes 'closeness' of $\hat{\boldsymbol{t}}_{\boldsymbol{code}}$ and $\boldsymbol{t_{code}}$, giving priority to Abstract Syntax Tree match & Data-Flow Graph match
  - Like BLEU, $\hat{\boldsymbol{t}}_{\boldsymbol{code}}$ may not compile even though high AST or Data-Flow match

UNIVERSITY OF WATERLOO

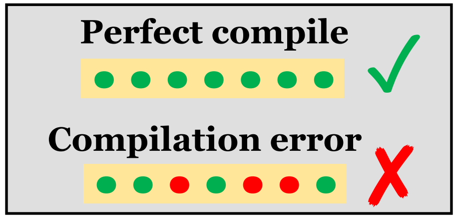
Proposed Metrics to Validate Compilation Requirements

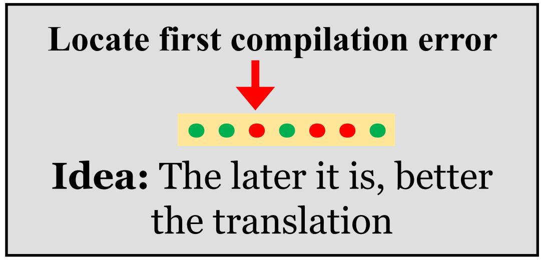


Our compilation requirement:
- $\hat{t}$ should compile

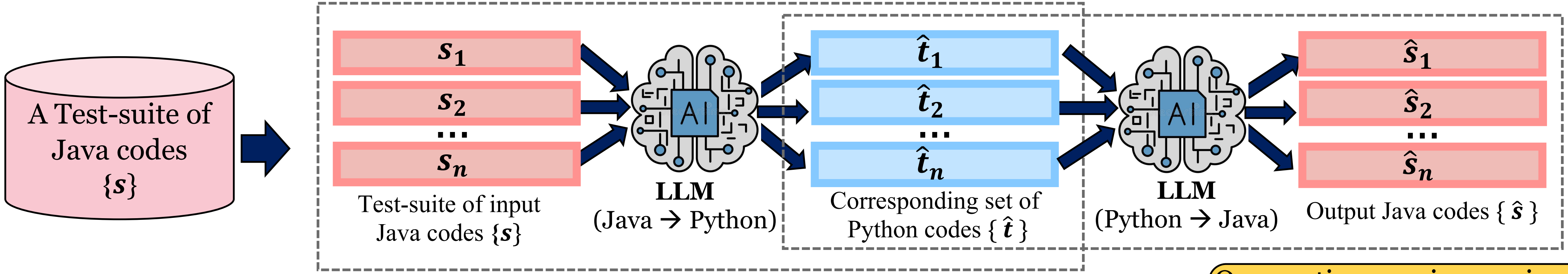
- Create a representative test-suite $\{s\}$ of Java codes, that solve range of algorithmic problems
- **Compilation Accuracy (CompAcc)**
  - %age of predicted translations that compiles correctly i.e., $\textbf{CompAcc}(\{\hat{t}\}) = \frac{|\{\hat{t}_i : \hat{t}_i \text{ compiles by T compiler}\}|}{|\{s\}|}$
- **Average First Error Position ($\textbf{errPos}_{\textbf{1st}}$)**
  - Fine-grained version of **CompAcc**, relating to closeness of translations from a perfect compilation
  - $\textbf{errPos}_{\textbf{1st}}(\{\hat{t}\}) = \frac{\sum_{i=1}^{|\{s\}|} \textbf{errPos}_{\textbf{1st}}(\hat{t}_i)}{|\{s\}|}$

$$\text{,where } \textbf{errPos}_{\textbf{1st}}(\hat{t}_i) = \begin{cases} 100 \times \frac{\text{position of first token in } \hat{\mathbf{t}}_{\mathbf{i}} \text{ responsible for a compilation error}}{|\hat{\mathbf{t}}_{\mathbf{i}}|+1} & \text{, if } \hat{\mathbf{t}}_{\mathbf{i}} \text{ doesn't compile} \\ 100 & \text{, otherwise} \end{cases}$$





Proposed Metrics to Validate Runtime-Equivalence Requirements



Our runtime-equiv. requirement:
- $\hat{t}$ should be runtime (IO) equivalent to s

- It is difficult to check IO-equivalence between { **s** } and { $\hat{t}$ } without manually-created test-cases
  - As an alternative, we approximate **RunEqAcc**({ **s** }, { $\hat{t}$ }) using **RunEqAcc**({ **s** }, { $\hat{s}$ })
- **Runtime-Equivalence Accuracy (RunEqAcc)**
  - Number of J-Unit tests ($\mathcal{J}_{s_i}$) on input Java code $\mathbf{s_i}$ that are successful on output Java $\hat{s}_i$, averaged over whole test-suite
  - $\mathbf{RunEqAcc}(\{\,\mathbf{s}\,\},\{\,\hat{s}\,\}) = \frac{\sum_{i=1}^{|\{s\}|} \mathbf{RunEqAcc}\,(s_i,\hat{s}_i)}{|\{s\}|}$,where $\mathbf{RunEqAcc}(s_i,\hat{s}_i) = \frac{\varepsilon + \sum_{j\in\mathcal{J}_{s_i}}\left(1_{j(\hat{s}_i)\equiv\text{success}}\right)}{\varepsilon + \left|\mathcal{J}_{s_i}\right|} \times 100$
  - $\varepsilon \approx 0^+$, used to avoid zero-division error

COMPUTING METRICS TOLERANCE : LESS THAN PERFECT DEFINITION OF "PERFECT"

Abbreviations

SM: Syntactic Match
DM: Data-flow Match
EM: Exact Match
CompAcc: Compiler Accuracy
RunEqAcc: Runtime Equivalence Accuracy
errPos$_{1st}$: Average First Error Position

Traditional metrics

Proposed metrics based on the requirements

Java → Python

Method / Tool	Model	BLEU	CodeBLEU	Wt. n-gram	SM	DM	EM	*CompAcc*	*RunEqAcc*	$errPos_{1st}$
Transpilers	java2python [3]	17.54	20.31	22.04	16.05	22.99	0	41.46	3.32	28.62
	TSS CodeConv [5]	24.44	41.87	**57.84**	39.66	**46.06**	0	58.30	0.45	**54.26**
Recent competing tools	CodeBERT [8]	51.13	34.97	-	34.35	29.24	0.47	92.80	0.4	-
	GraphCodeBERT [9]	57.93	39.04	-	37.99	32.16	0.74	92.86	2.0	-
	CodeGPT [10]	46.32	30.22	-	32.17	22.23	1.58	79.60	2.8	-
	CodeGPT-adapted [10]	44.29	29.28	-	31.59	20.38	1.84	80.15	2.4	-
	PLBART-base [11]	**63.10**	46.15	-	**42.18**	37.90	1.89	**96.44**	**6.8**	-
	CodeT5-base [12]	62.68	**46.24**	-	41.71	37.89	**2.52**	91.75	6.0	-
	TransCoder-ST [13]	55.41	43.77	-	41.59	36.06	1.84	94.85	5.6	-

Python → Java

Method / Tool	Model	BLEU	CodeBLEU	Wt. n-gram	SM	DM	EM	*CompAcc*	*RunEqAcc*	$errPos_{1st}$
Transpilers	py2java [4]	48.59	41.56	**50.46**	52.83	14.38	0	0	0	**1.61**
Recent competing tools	CodeBERT [8]	35.05	33.16	-	41.09	31.52	0	54.10	0	-
	GraphCodeBERT [9]	38.26	36.93	-	42.26	32.69	0	66.80	0	-
	CodeGPT [10]	48.94	38.01	-	42.74	34.08	0.68	40.65	**2.0**	-
	CodeGPT-adapted [10]	47.99	36.73	-	42.99	28.32	0.84	46.74	0.8	-
	PLBART-base [11]	**69.65**	48.77	-	54.21	30.91	**1.00**	**78.26**	0.8	-
	CodeT5-base [12]	60.84	**50.34**	-	**55.06**	**39.57**	0.89	68.70	1.6	-
	TransCoder-ST [13]	66.02	48.60	-	53.33	31.70	0.95	72.43	**2.0**	-

Tolerance p-values for our proposed method

	Java → Py	Py → Java
CompAcc	95%	75%
RunEqAcc	47.5%	37.5%
errPos$_{1st}$	70%	60%

Noteworthy observation: Python → Java (dynamically- to statically-typed) is more difficult than Java → Python!

UNIVERSITY OF WATERLOO

Conclusion & Future Scope

- This work is focused at requirement engineering for an LLM-based code translation
  - **Java** ↔ **Python** translation: two OOP languages, but syntactically very different
  - To produce **syntactically-correct** & **runtime-equivalent** translations
  - Proposed **new metrics** (*CompAcc*, *RunEqAcc*, $errPos_{1st}$) to verify whether LLM satisfies requirements
  - Computed **tolerance p-values** for the metrics

- In Future:
  - Add more requirements for **algorithmic-level equivalence**: compute-time, space-time
  - Unit-testing **can't guarantee equivalence for unhandled exceptions** → need more requirements
    - E.g, out of bound, zero division that are not handled by some try-catch logic
  - Identify **language features that are not translatable** e.g. pointers, multiple inheritance
    - Might need more requirements on what kind of codes can be translated
  - Translate between two **languages of different prog. paradigm** (e.g. Java for OOP → Racket for Functional)
    - Evaluate if this requires new requirements
  - Not all **new requirements may not be achievable** by our existing LLM pipeline
    - Might need more explicit teaching for the LLM

References

[1] B. G. Mateus, M. Martinez, and C. Kolski, “Learning Migration Models for Supporting Incremental Language Migrations of Software Applications,” Information and Software Tech., vol. 153, 2023.

[2] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, M. Luján, and H. Mössenböck, “Cross-language Interoperability in a Multi-language Runtime,” ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 40, no. 2, pp. 1–43, 2018

[3] T. Melhase et al., “java2python: Simple but Effective Tool to Translate Java Source Code into Python.” https://github.com/natural/java2python .

[4] “py2java: Python to Java Language Translator.” https://pypi.org/project/py2java/

[5] T. S. Solutions, “The Most Accurate and Reliable Source Code Converters.” https://www.tangiblesoftwaresolutions.com/ .

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention Is All You Need,” Adv. in Neural Info. Processing Systems (NeurIPS), vol. 30, 2017.

[7] “pylint: Python code Static Checker.” https://pypi.org/project/pylint/

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in Findings of the Association for Computational Linguistics: EMNLP 2020, (Online), pp. 1536–1547, Association for Computational Linguistics, Nov. 2020.

[9] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “GraphCodeBERT: Pre-training Code Representations with Data Flow,” in 9th International Conference on Learning Representations (ICLR), 2021.

[10] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al., “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation,” arXiv preprint arXiv:2102.04664, 2021.

[11] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Unified Pre-training for Program Understanding and Generation,” in Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, (Online), pp. 2655–2668, Association for Computational Linguistics, June 2021.

[12] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation,” in Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, (Online and Punta Cana, Dominican Republic), pp. 8696–8708, Association for Computational Linguistics, 2021.

[13] B. Roziere, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, “TransCoder-ST: Leveraging Automated Unit Tests for Unsupervised Code Translation,” in International Conference on Learning Representations (ICLR), 2022.

THANK YOU!

UNIVERSITY OF
WATERLOO