Constructing TransLibEval: A Detailed Procedure Report

Anonymous

Contents

1 Introduction

This process report supplements the main TransLibEval paper by providing a much more detailed and operational description of how the benchmark was constructed. While the paper explains the high-level design of the calibration and counterpart corpora, several implementation-oriented questions naturally arise in the minds of practitioners seeking to reproduce or extend the benchmark. Chief among these questions are how Python libraries were selected from PyPI statistics; how representative functionalities were chosen; how the four method-level tasks were defined for each library; what concrete coding rules were enforced to ensure cross-programming-language parallelism; how test suites were designed, normalized, and verified; how the Python-to-Java/C++ translation workflow was organized and executed; how disagreements during translation were resolved and what rejection criteria were adopted; and finally, how libraries with no feasible translations, such as TensorFlow and spaCy, were handled.

The purpose of this document is to answer those questions through a step-by-step reconstruction of the entire pipeline. We explain the design decisions and constraints that informed each stage; we provide the implementation conventions and coding specifications used by annotators; we describe the human workflow design including roles, responsibilities and arbitration procedures; and we summarize the toolchain configuration and quality assurance mechanisms adopted during construction. The report is intended to give readers sufficient operational detail to reproduce the benchmark, adopt its methodology for related tasks, or critically assess strengths and limitations of the resulting corpora.

2 Overall Workflow Overview

The construction pipeline of TransLibEval comprises three major phases: Phase A, the calibration construction in Python; Phase B, counterpart construction for Java and C++; and Phase C, benchmark finalization and packaging. Figure 1 illustrates the entire pipeline and the dependencies among its components; the figure can be referenced or replaced with a more detailed flowchart as needed in later drafts.

Phase A, the calibration construction phase, begins with the collection of PyPI usage statistics and proceeds through a multi-criteria filtering procedure to select candidate libraries. For each retained library, representative functionalities are identified through a combination of documenta-

tion inspection, example mining, and lightweight static analysis; these functionalities are then decomposed into four method-level tasks that exercise canonical API usage patterns. Each Python task is implemented following a unified coding specification that enforces naming rules, layout constraints, type conventions, and deterministic behavior. Method-level unit tests are constructed for every task to exercise normal operation, edge conditions, exception handling, input validation, and resource constraints. The calibration artifacts then undergo a review, correction, and acceptance loop before proceeding to counterpart construction.

Phase B, the counterpart construction phase, organizes translation work into language-specific teams and centers on establishing library equivalences between Python and the target languages. Translators and verifiers are assigned distinct roles: translators produce line-by-line Java and C++ renditions of the Python calibration tasks, and verifiers perform exhaustive reviews and execute translated test suites to ensure behavioral parity. Where direct library equivalents exist, annotators map Python APIs to their nearest semantic counterparts in Java or C++; where direct mappings do not exist, annotators search for decomposed solutions or minimal library sets to replicate semantics. All translations follow the five translation principles described elsewhere in this report. Verification is performed through execution-based equivalence testing, and any unresolved disagreements are escalated to the lead architect for arbitration.

Phase C, the benchmark finalization phase, consolidates the translations, removes non-translatable samples identified during Phase B, normalizes directory, and re-runs the entire battery of test cases across all three programming languages. The final deliverable includes packaged code and scripts to reproduce the test execution environment.

Overall, the construction required substantial human effort and coordination. The end-to-end effort reported here consumed approximately 500 person-hours: 300 person-hours were devoted to Python calibration construction and 200 person-hours to the Java and C++ counterpart translations and verification. The following sections provide a granular account of each phase, starting with the detailed procedures and validation protocols used in Phase A.

3 Phase A: Python Calibration Construction

Phase A of the TransLibEval benchmark focuses on the systematic construction of a Python-based calibration set. The purpose of this calibration

stage is twofold: (i) to establish a principled foundation for cross-library and cross-language translation tasks, and (ii) to form a controlled environment in which task difficulty, API semantics, and input-output determinism can be rigorously assessed. The resulting Python calibration set ultimately provides the structural template for the full benchmark and ensures that the task space reflects realistic software engineering scenarios while remaining analytically tractable.

This chapter describes the complete methodology for Phase A, including PyPI statistics collection, library filtering, canonical operation extraction, task definition, coding specifications, unit test construction, and engineer-in-the-loop quality assurance. The entire pipeline was executed between 2023–2025 and resulted in 212 validated Python tasks.

3.1 Collecting PyPI Usage Statistics

3.1.1 Data Source

To identify high-impact libraries within the Python ecosystem, we rely on the longitudinal PyPI download dataset aggregated by Flynn (2020–2024), which has become a de facto empirical reference in programming language research. This dataset aggregates installation logs across multiple mirrors, providing fine-grained monthly download counts for over 400,000 Python packages.

3.1.2 Data Cleaning

Raw PyPI download logs include a substantial amount of noise, requiring a multi-step cleaning process. We follow the standard procedure adopted in ecosystem measurement studies:

1. **Mirror Traffic Removal:** Automated downloads from Cloudflare or similar mirrors were removed by discarding user-agent patterns that match known proxies. This prevents inflated popularity scores.

2. **Alias Resolution:** Packages with known aliasing (e.g., `Pillow` vs. the legacy `pil` name) were merged into unified entries.

3. **Temporal Aggregation:** Monthly downloads were aggregated into a four-year cumulative trend score, smoothing seasonal fluctuations.

4. **Category-level Normalization:** Libraries were normalized within their functional categories (e.g., numerical computing, web access, ML/AI), enabling fair cross-domain comparisons.

The cleaned dataset provides a stable foundation for identifying representative libraries.

3.1.3 Library Filtering Criteria

A Python library was retained only if it satisfied all of the following criteria:

Criterion	Threshold
Download score	Top 10% within its category
API stability	$\geq$ 2 years without breaking changes
Documentation quality	Official API docs + runnable examples
Cross-PL ecosystem (optional)	Equivalent libraries in Java/C++

After filtering, we selected 53 libraries that met the conditions. These represent the majority of real-world, continuously maintained Python ecosystems.

3.2 Identifying Representative Functionalities

3.2.1 Data Sources for Functional Analysis

For each selected library, we constructed a functionality profile using four sources:

1. **Official Documentation:** All example scripts, tutorials, and API reference pages were scraped and indexed.
2. **GitHub Usage Patterns:** We collected the most-starred repositories that import the target library, yielding realistic usage patterns.
3. **AST-based API Mining:** We applied an AST miner to compute the frequency of API calls across more than 200,000 source files.
4. **Expert Summarization:** Two engineers manually reviewed the mined results and consolidated common operation patterns.

3.2.2 Classification

Next, the Python libraries we obtain will be classified into 10 major categories. Below is a detailed introduction to each category and a description of the libraries contained within it:

Data Processing This category encompasses libraries for numerical computation and algorithm implementation, comprising 17 Python libraries:

- `schema`: Data structure validation.
- `algorithms`: Fundamental algorithm implementations.
- `bitarray`: Efficient bit array operations.
- `collections`: Extended data structure containers.
- `schematics`: Data type system modeling.
- `dask`: Parallel computing framework.
- `decimal`: High-precision decimal arithmetic.
- `networkx`: Complex network analysis.
- `numpy`: Fundamental numerical computing.
- `pandas`: Data manipulation and analysis.
- `patsy`: Statistical model specifications.
- `datetime`: Date and time operations.
- `polars`: Fast DataFrame library.
- `pyfftw`: Fast Fourier transform wrappers.
- `marshmallow`: Object serialization/deserialization.
- `shapely`: Geometric object operations.
- `sortedcontainers`: Sorted collection types.

Utilities This category includes tools for common utilities, formatting, and extensions, containing 9 Python libraries:

- `asyncio`: Asynchronous I/O programming.
- `autopep8`: Python code formatting.
- `boltons`: Extended standard library utilities.
- `functools`: Higher-order functions.
- `funcy`: Functional programming tools.
- `itertools`: Iterator building blocks.

- `shutil`: File operations utilities.
- `tenacity`: Retry behavior implementation.
- `tqdm`: Progress bar visualization.

Machine Learning This category covers libraries supporting ML model training and inference, with 7 Python libraries:

- `catboost`: Gradient boosting toolkit.
- `joblib`: Lightweight pipelining.
- `lightgbm`: Light gradient boosting machine.
- `mlxtend`: Machine learning extensions.
- `cvxpy`: Convex optimization modeling.
- `scikit-learn`: Machine learning algorithms.
- `statsmodels`: Statistical modeling and testing.

Web Development This category contains tools for networking and web application development, including 5 Python libraries:

- `beautifulsoup4`: HTML/XML parsing.
- `Cerberus`: Lightweight data validation.
- `jsonschema`: JSON schema validation.
- `pydantic`: Data parsing and validation.
- `requests`: HTTP client library.

Visualization This category includes libraries for plotting charts and data visualization, with 5 Python libraries:

- `missingno`: Missing data visualization.
- `matplotlib`: Comprehensive plotting library.
- `seaborn`: Statistical data visualization.
- `bokeh`: Interactive web plotting.
- `plotly`: Interactive graphing library.

NLP This category encompasses libraries for text processing and linguistic analysis, containing 5 Python libraries:

- `Whoosh`: Pure-Python search engine.
- `gensim`: Topic modeling and NLP.
- `python-Levenshtein`: String similarity computation.
- `nltk`: Natural language toolkit.
- `spacy`: Industrial-strength NLP.

Graphics This category covers image processing and computer vision libraries, with 2 Python libraries:

- `opencv`: Computer vision library.
- `PIL`: Python imaging library.

Database This category includes tools for database operations, containing 1 Python library:

- `peewee`: ORM library.

Security This category encompasses encryption and security functionalities, with 1 Python library:

- `pycrypto`: Cryptographic tools.

Other This category contains miscellaneous libraries, with 1 Python library:

- `pyphonetics`: Phonetic algorithms.

3.3 Defining Method-Level Tasks

We enforce strict requirements to isolate method-level semantics and control task difficulty:

- Method-level granularity only; no file-level or class-level contexts.
- Exactly **one** third-party API call per task.
- Deterministic output (randomness permitted only with fixed seeds).
- No external side effects beyond local variables.
- Parameter and return types restricted to primitive types.
- I/O behavior must be completely reproducible.

3.3.1 Task Template

Each task follows the template:

```
class FooTask:
    def bar_operation(self, x: int, y: float) -> float:
        # exactly one third-party API call
        ...
```

The template ensures uniformity across all libraries and simplifies annotation and translation.

3.4 Coding Specifications

3.4.1 Naming Rules (PEP 8)

- Methods and variables: `snake_case`
- Classes: `PascalCase`
- Constants: `SCREAMING_SNAKE_CASE`

3.4.2 Layout Rules

The following structural constraints were imposed:

- Exactly one top-level class.
- Exactly one public instance method.
- No nested classes, no inheritance.
- No comments to avoid semantic hints that may bias LLMs.
- Four-space indentation.

3.4.3 Type Conventions

We require that all method parameters and return values shall be restricted to primitive data types. Composite/Object-level data types are explicitly prohibited. This constraint guarantees that when invoking focal methods externally from test suites, only limited data types need to be considered across diverse PLs, facilitating the test suite construction and the whole evaluation procedure:

- Allowed: `int`, `float`, `str`, `bool`.
- Disallowed: `list`, `dict`, `ndarray`, `DataFrame`, objects.
- No custom classes as parameters or return types.

A special mapping table (omitted here) ensures future alignment with cross-language translation.

3.5 Unit Test Construction

We design tests to evaluate both functional correctness and boundary behavior. Each task receives a corresponding test file `test_xxx.py` with the following structure:

- `setUp`: environment initialization.
- Five test categories:
  1. Normal operation
  2. Edge inputs
  3. Exception handling
  4. Input type validation
  5. Resource constraints

  Tests use only the `unittest` framework and avoid any I/O operations. Each test validates:

- deterministic outputs,
- consistent exception types,
- correct boundary behavior,
- robustness against malformed input.

3.6 Quality Assurance

The quality assurance pipeline involves two engineers:

Engineer A (Implementation). Implements all tasks following the specification.

Engineer B (Review + Execution). Performs:

- Pylint compliance checking,
- dry-run without actual API calls,
- execution with the real third-party library,
- edge-case retesting,
- issue logging and verification.

A lead architect conducts a final audit to ensure sampling balance across categories and libraries.

3.6.1 Final Output

The Python calibration stage produced **212 fully validated tasks**, each satisfying:

- semantic correctness,
- deterministic behavior,
- cross-language translatability,
- and unit-test completeness.

This forms the foundation for the subsequent translation evaluation phases.

4 Phase B: Java and C++ Counterpart Construction

Phase B of the TransLibEval benchmark focuses on constructing the Java and C++ counterpart corpora based on the Python calibration. The objective is to manually translate Python tasks, including their associated test suites, into Java and C++ while preserving semantic correctness, API equivalence, and testability. This chapter details the team organization, library equivalence search, translation rules, workflow, and quality assurance, as well as a discussion of rejected samples.

4.1 Team Organization

The translation process was conducted by a five-engineer team with distinct responsibilities:

- **Lead Architect:** responsible for final arbitration in case of disagreements.
- **Java Team:** one primary translator and one dedicated verifier.
- **C++ Team:** one primary translator and one dedicated verifier.

Each translator independently rewrote Python tasks following the five principles established for TransLibEval, ensuring consistency across programming languages. Verifiers checked translations for correctness, stylistic compliance, and test equivalence. The lead architect provided oversight, resolving any conflicts in type handling, library selection, or test behavior.

4.2 Searching for Library Equivalence Across PLs

4.2.1 Searching Protocol

The translation of Python tasks to Java and C++ often requires replacing third-party library calls with equivalent libraries or API calls. To ensure high-quality translations, translators followed a structured search pipeline:

1. **Official Ecosystem Equivalents:** Search Java (*Maven Central*) or C++ (*NuGet*, *vcpkg*) official libraries for equivalents of the Python APIs.
2. **Community-Maintained Equivalents:** Investigate libraries recommended by developer communities and Q&A sites, such as Stack Overflow and Quora.
3. **Decomposed Multi-Library Approach:** If a direct one-to-one mapping does not exist, translators combined multiple libraries to implement equivalent functionality.
4. **Non-Translatable Candidates:** If no feasible mapping could be found, the task was flagged for potential exclusion.

4.2.2 Corresponding Java and C++ Libraries of Each Category

Data Processing Category This category focuses on numerical computation and algorithm implementation.

Java Libraries:

- `com.fasterxml.jackson.core:jackson-core`: Core Jackson processing abstractions
- `com.fasterxml.jackson.core:jackson-databind`: Data binding functionality
- `com.opencsv:opencsv`: CSV parsing library
- `org.apache.commons:commons-csv`: Apache Commons CSV parser
- `org.apache.commons:commons-math3`: Mathematics and statistics components
- `org.jgrapht:jgrapht-core`: Java graph theory library
- `org.json:json`: JSON processing library
- `org.ojalgo:ojalgo`: Mathematics and linear algebra library
- `tech.tablesaw:tablesaw-core`: DataFrame and visualization library

C++ Libraries:

- `Eigen`: Template library for linear algebra
- `indicators`: Progress bars and indicators
- `nlohmann-json`: JSON parsing and serialization
- `rapidjson`: Fast JSON parser/generator
- `xtensor`: Multi-dimensional arrays with broadcasting

Utilities Category This category provides tools for common utilities, formatting, and extensions.

Java Libraries:

- `com.google.guava:guava`: Google core libraries
- `commons-cli:commons-cli`: Command-line interface parser

- `commons-io:commons-io`: Input/output utilities
- `dev.failsafe:failsafe`: Failure handling and retries
- `io.reactivex.rxjava3:rxjava`: Reactive programming library
- `joda-time:joda-time`: Date and time library
- `me.tongfei:progressbar`: Progress bar implementation
- `org.apache.commons:commons-collections4`: Collection utilities
- `org.apache.commons:commons-lang3`: String and object utilities

C++ Libraries:

- `boost`: Collection of portable C++ libraries
- `CLI`: Command-line interface parser
- `date`: Date and time library
- `fmt`: Formatting library
- `range`: Range library for C++

Machine Learning Category This category supports ML models' training and inference.

Java Libraries:

- `com.github.haifengl:smile-core`: Statistical machine learning
- `nz.ac.waikato.cms.weka:weka-stable`: Weka machine learning toolkit
- `org.deeplearning4j:deeplearning4j-core`: Deep learning framework
- `org.nd4j:nd4j-native-platform`: N-dimensional arrays for Java

C++ Libraries:

- `xgboost`: Optimized gradient boosting library

Web Development Category This category provides tools for networking and web application development.

Java Libraries:

- `com.networknt:json-schema-validator`: JSON Schema validator
- `com.squareup.okhttp3:okhttp`: HTTP client library
- `io.projectreactor:reactor-core`: Reactive programming foundation
- `jakarta.validation:jakarta.validation-api`: Bean validation API
- `org.apache.httpcomponents:httpclient`: HTTP components client
- `org.everit.json:org.everit.json.schema`: JSON Schema validator
- `org.jsoup:jsoup`: HTML parsing and manipulation
- `org.springframework:spring-context`: Spring framework core

C++ Libraries:

- `libcurl`: Client-side URL transfer library
- `libxml2`: XML parsing and manipulation

Visualization Category This category includes libraries for plotting charts and visualizing data.

Java Libraries:

- `org.jfree:jfreechart`: Chart library for Java
- `org.knowm:xchart`: Lightweight charting library

C++ Libraries:

- `plplot`: Scientific plotting library

NLP Category This category encompasses libraries for text processing and NLP tasks.

Java Libraries:

- `edu.stanford.nlp:stanford-corenlp`: Stanford CoreNLP toolkit
- `org.apache.commons:commons-text`: Text manipulation utilities
- `org.apache.lucene:lucene-core`: Search engine library
- `org.apache.opennlp:opennlp-tools`: Natural language processing

- `org.tartarus:snowball`: Snowball stemmers

C++ Libraries:

- `fasttext`: Library for text classification

Graphics Category This category covers image processing and computer vision.

Java Libraries:

- `org.locationtech.jts:jts-core`: Geometry topology suite

C++ Libraries:

- No specific C++ libraries mapped for this category

Database Category This category includes tools for database operations.

Java Libraries:

- `org.jdbi:jdbi3-core`: SQL convenience library
- `org.sql2o:sql2o`: Database access library

C++ Libraries:

- `sqlite3`: SQL database engine

Security Category This category encompasses encryption and security functionalities.

Java Libraries:

- `commons-codec:commons-codec`: Encoding and decoding utilities
- `org.bouncycastle:bcprov-jdk18on`: Cryptography provider

C++ Libraries:

- `cryptopp`: Crypto++ encryption library

Other Category This category contains miscellaneous libraries.

Java Libraries:

- No specific Java libraries mapped for this category

C++ Libraries:

- `clang`: C language family frontend
- `llvm`: Compiler infrastructure

4.3 Translation Rules (Five Principles)

To ensure high-quality, consistent translations, all tasks and test suites were translated following five core principles:

Principle 1 (Naming Convention): We define a series of naming convention rules for various elements in Java and C++ programs. For example, classes are required to be named in PascalCase (e.g., TaskExecutor) while constants are required to be named in SCREAMING_SNAKE_CASE (e.g., DEFAULT_TIMEOUT) for both Java and C++. Methods and variables in Java are required to use camelCase (e.g., calculateTotal), while those in C++ use snake_case (e.g., validate_input).

Principle 2 (Type Mapping): As a dynamically-typed language, Python does not explicitly declare variable types, while Java and C++ are statically-typed. We establish a uniform standard for type mapping: Python's int and float are consistently mapped to int and double in both Java and C++. For dynamically assigned Python variables, we use Object in Java and auto in C++ to maintain flexibility while ensuring type safety.

Principle 3 (Implementation Layout): We enforce specific layout rules for translated code. Both Java and C++ programs use K&R brace style and 4-space indentation. Java programs are wrapped in class structures, while C++ programs follow header/source separation when appropriate. All comments from the original Python code are removed to avoid providing semantic hints that could bias translation models.

Principle 4 (Library Equivalence): When translating Python tasks involving third-party libraries, we prioritize direct library equivalents where available. For Python built-in libraries like math and os, we map to java.lang.Math and java.lang.System in Java, and cmath and cstdlib in C++. When direct equivalents don't exist, we search for minimal library sets through programming Q&A sites and AI assistants to achieve functional equivalence. All dependencies are managed using Maven for Java and NuGet for C++.

Principle 5 (Test Equivalence): The original Python tests using unittest framework are translated to maintain behavioral parity. We use JUnit for Java and GoogleTest for C++, preserving the same test structure, assertion logic, and edge case coverage. Input-output mappings, exception handling, and resource validation are consistently maintained across all three languages to ensure test equivalence.

4.4 Translation Workflow

The translation workflow consisted of multiple tightly coordinated steps:

1. **Translation of Python Code:** The primary translator rewrote the Python task into Java/C++ while adhering to the five principles.
2. **Translation of Test Suite:** The associated unit tests were translated concurrently to ensure parity in functional verification.
3. **Verification:** The verifier reviewed the translated code for stylistic compliance, type correctness, API usage, and test equivalence. The verifier compiled and executed all tests to confirm correctness.
4. **Dispute Resolution:** Any disagreements between translator and verifier were escalated to the lead architect for arbitration.

Common disagreement cases included:

- Unexpected floating-point rounding differences
- Library method behavioral mismatches
- Divergent exception types
- C++ resource lifetime management issues

4.5 Rejected Samples

During translation, twelve Python tasks were rejected due to irreconcilable issues:

- **Category A — No Equivalent Library:** Libraries like `spaCy` had no comparable Java/C++ implementation with similar semantic granularity.
- **Category B — API Divergence Too Large:** Libraries like `TensorFlow` exhibited fundamentally different return types, shape representations, or execution models (eager vs. graph mode), preventing faithful translation.
- **Category C — Inevitable Composite Types:** Certain tasks inherently produced structured data, violating the primitive-only principle imposed on TransLibEval.

After eliminating these twelve tasks, **200 triplets** remained as fully validated cross-language parallel benchmark samples.

4.6 Summary

Phase B established a robust Java and C++ counterpart corpus, complementing the Python calibration. The combination of structured search protocols, five translation principles, multi-level verification, and lead architect arbitration ensured that the resulting tasks were semantically correct, consistent, and reproducible. These 200 triplets now serve as the core of the cross-PL benchmark for evaluating LLM-based code translation.

5 Phase C: Final Packaging and Benchmark Quality Assurance

After building the Python calibration and Java/ c++ corresponding corpora, Stage C focuses on the final packaging and strict benchmarking QA to ensure repeatability, usability, and consistency in all three programming languages. This stage is crucial for establishing a benchmark that can be evaluated by both individuals and LLMS for the reliable use of the pipeline. Among them, our packaging has already been placed on the Github repository. The usage method is guided by the README, so it will not be elaborated in this document.

5.1 Quality Assurance Workflow

A multi-tiered QA pipeline was implemented:

1. **Automated Linting:** Code in all languages was checked against style guides (PEP 8 for Python, Google Java Style, C++ Core Guidelines).
2. **Automated Test Execution:** All unit tests were executed on clean environments to verify reproducibility.
3. **Manual Spot-Check:** A lead engineer reviewed a random subset of 20% of tasks to ensure adherence to translation principles.
4. **Discrepancy Resolution:** Any detected deviations or inconsistencies were corrected.
5. **Final Audit:** Lead architect performed a complete review to certify the benchmark before public release.

Through these steps, TransLibEval achieved a high standard of correctness, traceability, and maintainability.

6 Manual Benchmark Extension Guidelines

TransLibEval is designed to be extensible to other programming languages (PLs) or additional libraries beyond the original Python, Java, and C++ implementations. This section provides a step-by-step guideline for users who wish to extend the benchmark manually, using any programming language as the calibration source.

6.1 Extension Workflow Overview

The extension process mirrors the original three-phase construction pipeline:

1. **Phase A:** Construct a calibration set in a source PL.
2. **Phase B:** Translate the calibration set into one or more target PLs.
3. **Phase C:** Package the extended benchmark and perform quality assurance.

6.2 Phase A: Source Calibration Construction

- **Library Selection:** Choose third-party libraries (TPLs) with high usage, stable APIs, and good documentation in the source PL.
- **Task Definition:** For each library, define method-level tasks that:
  - Contain exactly one TPL API call.
  - Use only primitive types for parameters and return values (e.g., `int`, `float`, `str`, `bool`).
  - Exhibit deterministic behavior.
- **Coding Specifications:** Enforce consistent naming, layout, and type conventions (e.g., PEP 8 for Python).
- **Unit Test Construction:** Write comprehensive unit tests covering normal operation, edge cases, and exception handling.

6.3 Phase B: Target Language Translation

- **Library Mapping:** Identify equivalent libraries in the target PL(s) using official repositories and community resources.

- **Translation Principles:** Adhere to the five translation principles (naming, type mapping, layout, library equivalence, test equivalence) as defined in Section 4.3.
- **Verification:** Ensure semantic and behavioral parity through manual review and test execution.
- **Rejection Criteria:** Exclude tasks that cannot be faithfully translated due to lack of library equivalents or inherent type mismatches.

6.4 Phase C: Packaging and QA

- **Directory Structure:** Follow the standardized layout of TransLibEval for easy integration.
- **Quality Assurance:** Perform linting, automated testing, and manual spot-checks to ensure consistency and correctness.

By following this structured approach, researchers and practitioners can reliably extend TransLibEval to new languages or libraries, maintaining the benchmark's rigor and cross-language comparability.

7 Conclusion

This process report provides a reproducible and thorough description of how TransLibEval was constructed, revealing practical details not covered in the main paper. By documenting calibration construction, counterpart translation, and final packaging procedures, we ensure transparency for readers and reproducibility for future researchers.

Key takeaways include:

- Manual calibration and translation require a structured workflow, strict adherence to coding principles, and rigorous QA to maintain cross-PL consistency.
- A multi-tiered verification and dispute resolution mechanism ensures high-quality benchmark samples.
- Comprehensive standardized directory layouts facilitate reproducibility, traceability, and automation.

- Certain Python libraries remain challenging to translate due to API divergence or inherent type complexity, highlighting limitations in cross-language benchmark design.

Overall, TransLibEval establishes a transparent, reproducible, and high-quality benchmark for evaluating LLM-based code translation. It provides not only the dataset but also a detailed procedural blueprint for constructing similar multi-language code translation benchmarks in the future.