Strust-v1: Performance Metric for COBOL-to-Java Generative AI

George Weale
Columbia University
New York, NY, USA
george.weale@columbia.edu

***Abstract*—Enterprises worldwide rely on critical COBOL systems that require modernization to languages such as Java. The verification of functional equivalence between source COBOL and target Java code remains a challenge in modernization projects, without a standardized quantification methods. This paper introduces Strust, a framework that provides objective metrics for functional equivalence in COBOL-to-Java conversion through containerized differential testing. Strust executes COBOL and Java code within isolated containers, applies identical inputs, and compares outputs to detect functional discrepancies. We present the version-1 implementation called the Verified Snippet Demonstrator. The results confirm the feasibility of automatically executing corresponding COBOL and Java code snippets and comparing their outputs for functional equivalence across diverse test cases. Strust has the potential to reduce risk in modernization projects, enable objective comparisons between conversion tools, and give confidence in automated code transformation processes. The framework fixes the critical verification gap in legacy modernization projects through a reproducible, quantifiable approach to functional equivalence testing.**

I. Introduction

A. Mainframe Modernization

COBOL systems continue to power mission-critical applications across global industries including finance, insurance, government, healthcare, and transportation. These systems process over $3 trillion in daily transactions and run 95% of ATM operations. Organizations have pressure to modernize these legacy systems due to the operational costs of mainframe infrastructure continue to increase while the pool of qualified COBOL developers shrinks as professionals retire. Business requirements demand integration with modern platforms, cloud services, and a need for agility to respond to market changes. The growing COBOL skills gap creates operational risk for organizations unable to maintain and importantly fix these systems.

Modernization strategies are currently just rehosting (lifting and shifting to new infrastructure), replatforming (minimal code changes to run on modern platforms), refactoring (restructuring code while preserving functionality), complete rewrites, and automated code conversion. Automated conversion (especially now with generative AI) to Java is an increasingly common approach due to its balance of cost, time, and risk considerations.

B. The Achilles' Heel: Validation and Verification

Automated conversion creates multiple verification challenges stemming from fundamental differences between COBOL and Java. These languages differ in data type representation, default values, numerical precision, memory management, flow control, error handling, and runtime behavior. COBOL features including REDEFINES clauses, level 88 conditions, PERFORM statements, and implicit type conversions have no direct equivalents in Java.

Environmental dependencies further complicate verification efforts. COBOL programs often depend on mainframe subsystems such as CICS, IMS, JCL, and VSAM. Data representations differ between platforms, from EBCDIC character encoding to packed decimal formats. Subtle semantic differences in execution environments can cause behaviorally non-equivalent programs despite syntactic similarity.

Traditional verification approaches rely on costly, time-consuming manual testing and code review. Large-scale modernization projects involve millions of lines of code, making comprehensive manual verification economically infeasible. Undetected conversion errors that reach production can cause significant business disruption, financial loss, and reputational damage. The industry lacks standardized verification methodologies that provide quantitative metrics of functional equivalence.

C. Lack of Standardized Quality Metrics

Current quality assessment approaches for code conversions have limitations. Organizations typically rely on vendor claims regarding conversion accuracy without independent verification. Projects has extensive bespoke testing cycles that consume significant time and resources. Static analysis tools provide limited insight into functional equivalence, focusing on structural metrics rather than runtime behavior. This situation creates several challenges.

First, organizations cannot objectively compare results from different conversion tools or service providers. Second, projects struggle to establish clear quality thresholds for acceptance decisions. Third, iterative improvements lack quantifiable measurement. Fourth, risk assessment remains subjective rather than data-driven. The industry needs independent, objective, and widely accepted metrics for functional equivalence that specifically address conversion quality.

D. Proposed Solution: The Strust Framework

This paper introduces Strust, a framework that provides an independent service for quantifying functional equivalence in COBOL-to-Java conversions. Strust implements containerized differential testing to validate that converted Java code produces identical outputs to the original COBOL for identical inputs. The framework eliminates environmental variables through containers that provide isolated, reproducible execution environments for both COBOL and Java code.

Strust focuses explicitly on functional equivalence as the primary indicator of conversion correctness. The framework defines functional equivalence as producing bit-for-bit identical outputs for matching inputs across all execution paths, capturing the essence of preserving business logic through modernization. Containerized differential testing makes this verification by normalizing execution environments, systematically applying inputs, and precisely comparing outputs.

II. Background and Related Work

A. COBOL Language Characteristics and Conversion Challenges

COBOL (Common Business-Oriented Language) was designed for business data processing and retains a distinctive structure that presents specific conversion challenges. COBOL programs organize code into divisions including the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION. The DATA DIVISION defines data structures using level numbers for hierarchical relationships, while the PROCEDURE DIVISION contains executable statements organized into sections and paragraphs.

Several COBOL features create particular difficulties for conversion to Java. The REDEFINES clause allows multiple record layouts to occupy the same memory location, a part of the language without direct equivalence in Java's type system. The structured PERFORM verb allows complex control flow including inline, out-of-line, and PERFORM VARYING constructs that must translate to Java loops and method calls. COBOL's implicit type conversions makes operations between disparate data types without explicit casting. The GO TO statement creates non-structured control flow that requires transformation to structured equivalents in Java.

Data representation differences between platforms create additional challenges. COBOL systems typically use EBCDIC character encoding while Java uses Unicode. COBOL's computational fields include COMP-3 packed decimal format that stores two decimal digits per byte plus sign, requiring custom Java classes to maintain precision and behavior. COBOL programs often rely on field justification rules and implicit padding that must be explicitly implemented in Java.

File handling has more complications as COBOL should work with various file organizations including sequential, indexed, and relative files with specific access methods. Java's I/O framework differs fundamentally, requiring custom implementation of equivalent functionality. Environment-specific features including JCL parameters, system utilities, and mainframe subsystems such as CICS and IMS introduce external dependencies that complicate conversion.

B. Existing COBOL Modernization Tools and Platforms

The COBOL modernization market includes has established tools and platforms that provide automated conversion capabilities. Micro Focus Enterprise Developer offers COBOL-to-Java transformation as part of its Enterprise Suite. AWS Mainframe Modernization utilizes Blu Age technology to convert COBOL to Java through pattern-based refactoring(Fine tuned Ai models take its place). Advanced's Modern Systems implements rule-based translation with customizable transformation engines. TSRI JANUS Studio uses model-driven architecture to change COBOL to object-oriented Java. Raincode provides a compiler-based approach that preserves COBOL semantics while generating Java bytecode.

These tools use different quality assurance approaches including pattern libraries, transformation rules, semantic analysis, and code generation templates. However, they universally lack independent validation of functional equivalence. Each vendor implements proprietary validation approaches without published metrics or standardized assessment methodologies. Cloud providers including AWS, Microsoft Azure, and Google Cloud Platform rely on partner technologies or acquired tools for modernization capabilities, inheriting the same verification limitations.

The verification gap creates a market situation where organizations cannot objectively compare conversion quality across vendors. This lack of standardized metrics complicates procurement decisions and increases project risk through uncertainty regarding conversion fidelity.

C. Software Quality Metrics and Static Analysis

Traditional software quality metrics provide limited insight into functional equivalence for converted code. Lines of Code (LOC) measures program size but not behavioral correctness. Cyclomatic Complexity quantifies control flow complexity but cannot verify equivalent execution paths. Halstead metrics evaluate program volume and difficulty but ignore runtime behavior. The Maintainability Index combines metrics to assess maintainability but provides no indication of functional preservation.

These metrics face fundamental limitations for conversion assessment. Complexity metrics may increase legitimately during conversion as COBOL constructs changes into multiple Java statements. Structural metrics like class coupling emerge in object-oriented transformations without indicating correctness. Static analysis can identify potential issues like unreachable code or type mismatches but cannot guarantee equivalent behavior at runtime.

Static analysis tools for COBOL include Micro Focus Enterprise Analyzer and CAST Application Intelligence Platform, which analyze program structure and dependencies. Similar tools for Java include SonarQube, PMD, and Checkstyle. While these tools incease code quality assessment, they cannot verify that converted code preserves the original program's behavior across all execution paths. Static analysis provides necessary but insufficient verification for conversion projects.

Formal methods and symbolic execution are verification approaches through mathematical proof of program properties.

These techniques can theoretically verify functional equivalence but face practical limitations for large-scale COBOL programs due to computational complexity and mainframe environmental dependencies. Strust is a more pragmatic approach through dynamic testing that scales to real-world conversion projects.

D. Containerization Technologies

Docker containerization are for COBOL-to-Java comparison testing. Containers encapsulate applications with their dependencies in isolated environments, ensuring consistent execution regardless of underlying infrastructure. This isolation eliminates environmental variables that might obscure functional differences between COBOL and Java implementations.

Containers have several advantages for verification testing. They provide reproducible execution environments that eliminate "works on my machine" inconsistencies. Containers package all required dependencies including compilers, runtime libraries, and system utilities. Containerization makes automation through orchestration tools and APIs. Docker's layered filesystem optimizes resource utilization when testing multiple program variants.

These make containerization the foundation for Strust's differential testing approach. Containers create controlled environments for both COBOL and Java execution, eliminating platform differences as potential causes of behavioral variation. This isolation focuses verification specifically on the correctness of the code conversion process.

E. Identifying the Gap

The combination of related work reveals a gap in modernization verification. While numerous tools enable COBOL-to-Java conversion, standardized methods for verifying functional equivalence remain absent. Static analysis provides structural insights but cannot confirm behavioral equivalence. Traditional testing approaches require extensive manual effort. Formal methods face scalability limitations for real-world applications.

Strust fills this gap by combining differential testing with containerization to provide automated, independent, and quantifiable functional equivalence assessment. The framework isolates the specific question of conversion correctness from other quality concerns. By focusing on identical outputs for identical inputs, Strust provides an objective measure of the most aspect of modernization: preserving the exact behavior of business-critical applications.

III. The Strust Methodology

A. Core Principle: Differential Execution and Comparison

Strust aims for a formal definition of functional equivalence in the context of COBOL-to-Java conversion. Two programs demonstrate functional equivalence when they produce identical relevant outputs for identical inputs under controlled conditions across all possible execution paths. This definition focuses on observable behavior rather than implementation details, recognizing that structural differences between languages necessitate different implementation approaches to achieve identical functionality.

The differential testing workflow comprises five phases:

1. Input Provisioning: The framework prepares identical input data for both the COBOL and Java programs. Inputs include files, environment variables, command-line arguments, and any other data sources that affect program execution.

2. Parallel Execution: Strust executes the COBOL program in Container A and the Java program in Container B under controlled conditions. Each container provides an isolated environment with appropriate runtime components and compilers.

3. Output Capture: The framework captures all relevant outputs from both executions. Outputs include generated files, console output (stdout/stderr), and optionally database changes or other side effects.

4. Output Comparison: Strust compares corresponding outputs from the COBOL and Java executions to identify discrepancies. Comparison algorithms apply appropriate normalization to eliminate insignificant differences.

5. Result Aggregation: The framework calculates functional equivalence metrics based on comparison results across all test cases. Metrics quantify the degree of behavioral correspondence between the original and converted code.

B. Component Deep Dive

1) Test Case Definition: Test cases establish the basis for functional equivalence assessment through well-defined inputs and expected outputs. Input data derives from multiple sources depending on application context. Manually created test data targets specific program features or known edge cases. Generated inputs leverage techniques such as boundary value analysis, equivalence partitioning, and combinatorial testing to maximize path coverage. Production data samples provide realistic testing scenarios while respecting data privacy requirements.

Test case definitions specify the exact format and content of input files, including record layouts, field types, and data values. Parameters include command-line arguments, environment variables, and configuration settings that influence program behavior. Database state mounting creates consistent starting conditions for programs that interact with persistent storage.

Expected output points define what constitutes relevant output for equivalence comparison. Primary outputs include generated files with specified formats and content. Console output captures diagnostic information and user interaction. Database changes record modifications to persistent storage. Memory structures track changes to program state. The test case definition explicitly correspondes between COBOL and Java outputs for comparison purposes.

2) Environment Orchestration: The COBOL environment container includes the GnuCOBOL compiler (or IBM Enterprise COBOL via ZD&T/Wazi for strict compatibility), required runtime libraries, copybook inclusion paths, and appropriate data file encoding (EBCDIC). Configuration options specify dialect-specific settings to match the original

mainframe environment. The container includes necessary subsystem emulations for CICS, IMS, or other mainframe components required for program execution.

The Java environment container includes the correct JDK version, build tools (Maven/Gradle), and required runtime dependencies. Configuration includes classpath setup, JVM parameters, and any environment-specific settings. The container includes implementations of mainframe-specific functionality required by the converted Java code.

Volume mounting creates data transfer between the host system and containers. Input data transfers to containers through mounted volumes or direct copying. Output data retrieves from containers through the same mechanisms. This approach maintains isolation while enabling data flow for testing purposes.

3) Execution Control: The execution control layer manages container lifecycle and program invocation through a scripting/orchestration layer. Container management includes building images, creating containers, starting execution, monitoring status, stopping execution, and cleanup operations. The control layer implements timeout handling to address infinite loops or performance issues.

Input provisioning prepares test data and configuration for container execution. The control layer transfers input files to appropriate container locations, sets environment variables, and prepares command-line arguments. Configuration parameters establish execution conditions including resource limits and timeout thresholds.

Compilation and execution use specific commands within each container. For COBOL, the control layer invokes the compiler (e.g., cobc) with appropriate options followed by execution of the resulting binary. For Java, compilation uses javac followed by execution with the java command and appropriate classpath settings. The control layer captures return codes to detect compilation or execution failures.

The control layer distinguishes between container failures and program failures, maintaining appropriate isolation between infrastructure issues and program behavior. This separation ensures that test results reflect actual program behavior rather than environmental artifacts.

4) Output Capture and Normalization: Output capture systematically collects program outputs for comparison. The framework captures stdout and stderr streams through container output redirection. File output monitoring tracks specified directories and files for changes during program execution. The capture mechanism records metadata including timestamps, sizes, and access patterns.

Normalization eliminates insignificant differences between COBOL and Java outputs. Whitespace normalization standardizes indentation, line breaks, and spacing that might differ between outputs without affecting semantic content. Line ending standardization converts between Windows (CRLF) and Unix (LF) conventions. Character encoding normalization is for differences between EBCDIC, ASCII, and Unicode representations.

Timestamp normalization accounts for execution time differences that appear in log outputs. Floating-point precision normalization looks at the differences in numerical representation between platforms. Order normalization handles cases where output ordering might vary without affecting correctness. The normalization process applies transformation rules consistently to both COBOL and Java outputs to enable meaningful comparison.

5) Equivalence Comparison Engine: The comparison engine implements algorithms for detecting functional differences between normalized outputs. For text files, the engine uses diff algorithms that identify line-by-line variations. Binary file comparison uses byte-level comparison with optional content-aware parsing for structured formats. Structured data comparison applies format-specific rules for formats such as XML, JSON, or CSV.

Tolerance settings address acceptable variation in specific contexts. Numeric comparison applies epsilon-based tolerance for floating-point values to accommodate minor precision differences. Pattern-based comparison helps regular expressions for outputs with variable components. Selective comparison allows excluding specified sections from comparison when they contain inherently variable content like when they have timestamps or random values.

Error handling fixes cases where one process produces no output or terminates abnormally. The comparison engine distinguishes between different error scenarios including compilation failure, runtime exception, timeout, or resource exhaustion.

6) Metric Calculation: Strust produces metrics that quantify functional equivalence based on output comparison results. The fundamental metric reports pass/fail status for each test case based on output equivalence. Aggregate metrics calculate the percentage of passing test cases across the test suite, providing a high-level indication of conversion quality.

The framework works with metrics for more nuanced analysis. Weighted scoring assigns importance factors to test cases based on business importance, execution frequency, or complexity. Coverage metrics incorporate information about code path execution to contextualize test results. Severity classification categorizes discrepancies based on potential business impact.

Future metric development incorporate performance comparison between COBOL and Java implementations. Execution time metrics measure relative speed under comparable conditions. Resource utilization metrics track memory, CPU, and I/O usage. These performance metrics complement functional equivalence measures to provide comprehensive conversion quality assessment.

C. Handling State

Programs with persistent state present specific challenges for equivalence testing. For database interactions, the framework initializes identical database states before execution and compares resulting states after execution. Schema translation ensures equivalent database structures between COBOL and Java environments. Transaction boundary identification isolates specific database operations for targeted comparison.

File-based state management addresses VSAM files and other persistent storage mechanisms. The framework creates

identical initial file states and compares resulting file states after execution. Custom comparators handle file format differences between mainframe and distributed environments.

Mocking frameworks simulate external systems when direct equivalence testing proves impractical. The mocking approach records interaction patterns from both COBOL and Java programs and compares these interaction sequences for equivalence. This technique verifies that both implementations make identical requests to external systems.

IV. POC Implementation: "Verified Snippet Demonstrator"

A. Goals and Scope of the PoC

The Verified Snippet Demonstrator implements a subset of the Strust framework to demonstrate the feasibility of containerized differential testing for COBOL-to-Java comparison. The primary goal is to show proof of the core technical approach rather than delivering a production-ready system. This demonstration focuses on validating that the containerization approach provides suitable isolation for meaningful comparison and that the differential testing detects functional discrepancies.

The implementation supports self-contained COBOL programs without inter-program calls or complex subsystem interactions. Supported COBOL features include basic arithmetic operations, conditional logic (IF/ELSE/EVALUATE), simple iterative constructs (PERFORM), sequential file I/O, and console output. The implementation uses GnuCOBOL for compilation and execution due to its accessibility and open-source nature.

The Java environment uses standard JDK without specialized frameworks. Input provisioning occurs through text files and command-line arguments. Output comparison focuses on exact matches of text output (console and files) after normalization. The proof-of-concept uses manually created Java code that implements equivalent functionality to corresponding COBOL programs, as automatic conversion tools integration exceeds the scope of the initial demonstration.

B. Architecture and Technology Stack

The Verified Snippet Demonstrator implements a layered architecture with clear separation of concerns between components. The orchestration layer uses Python with the Docker SDK to manage container lifecycle and execution flow. This layer has the control logic for test execution and result aggregation.

Docker Engine provides containerization for isolated execution environments. Container definitions use standard Dockerfile syntax to create reproducible environments. The COBOL container builds on a Linux base image with GnuCOBOL 3.1 installation and necessary runtime libraries. The Java container uses OpenJDK 11 on Linux with standard build tools.

The command-line interface accepts parameters for test execution including paths to COBOL and Java source files, input specifications, output definitions, and comparison options. The interface provides flexibility for testing different scenarios while maintaining a consistent execution model. The code bellow example illustrates a typical command invocation:

```
python Strust.py --cobol-source financial_calc
    .cob
        --java-source FinancialCalc.java
        --input-file transaction_data.txt
        --output-file results.txt
        --compare-stdout
        --normalize whitespace,lineendings
```

The architecture maintains clear boundaries between components through well-defined interfaces. This design enables future extension to support additional features while preserving the core differential testing.

C. Implementation Details

The input handling component processes test specifications and prepares execution environments. The orchestrator reads input files, configuration parameters, and source code. The system copies these artifacts to container locations using Docker volume mounting or direct copying through the Docker API. Input preparation includes character encoding conversion where necessary to accommodate differences between host and container environments.

Execution logic implements specific compilation and runtime commands within each container. For COBOL, the orchestrator executes commands similar to:

```
cobc -x -free -o program.exe program.cob
./program.exe < input.txt > output.txt
```

For Java, execution follows this pattern:

```
javac Program.java
java Program < input.txt > output.txt
```

The orchestrator captures return codes from these operations to detect compilation or execution failures. Timeout monitoring stops indefinite execution by terminating containers after a configurable duration.

Output retrieval extracts execution results from containers for comparison. The framework copies output files from container filesystems to the host for analysis. Standard output and error streams capture through Docker container logs. Metadata collection records execution timing, resource usage, and completion status.

The normalization implementation applies transformation rules to standardize outputs before comparison. Whitespace normalization removes leading and trailing whitespace and standardizes internal spacing. Line ending normalization converts all line terminators to a consistent format. Case normalization optionally converts text to a single case when case differences lack semantic significance.

Comparison logic creates an approach to detecting differences. The system first compares file existence to verify that both implementations produce the expected outputs. Size comparison provides a quick initial check for obvious differences. Content comparison performs a detailed analysis of file contents, applying additional normalization as specified. The diff algorithm highlights specific differences when discrepancies occur.

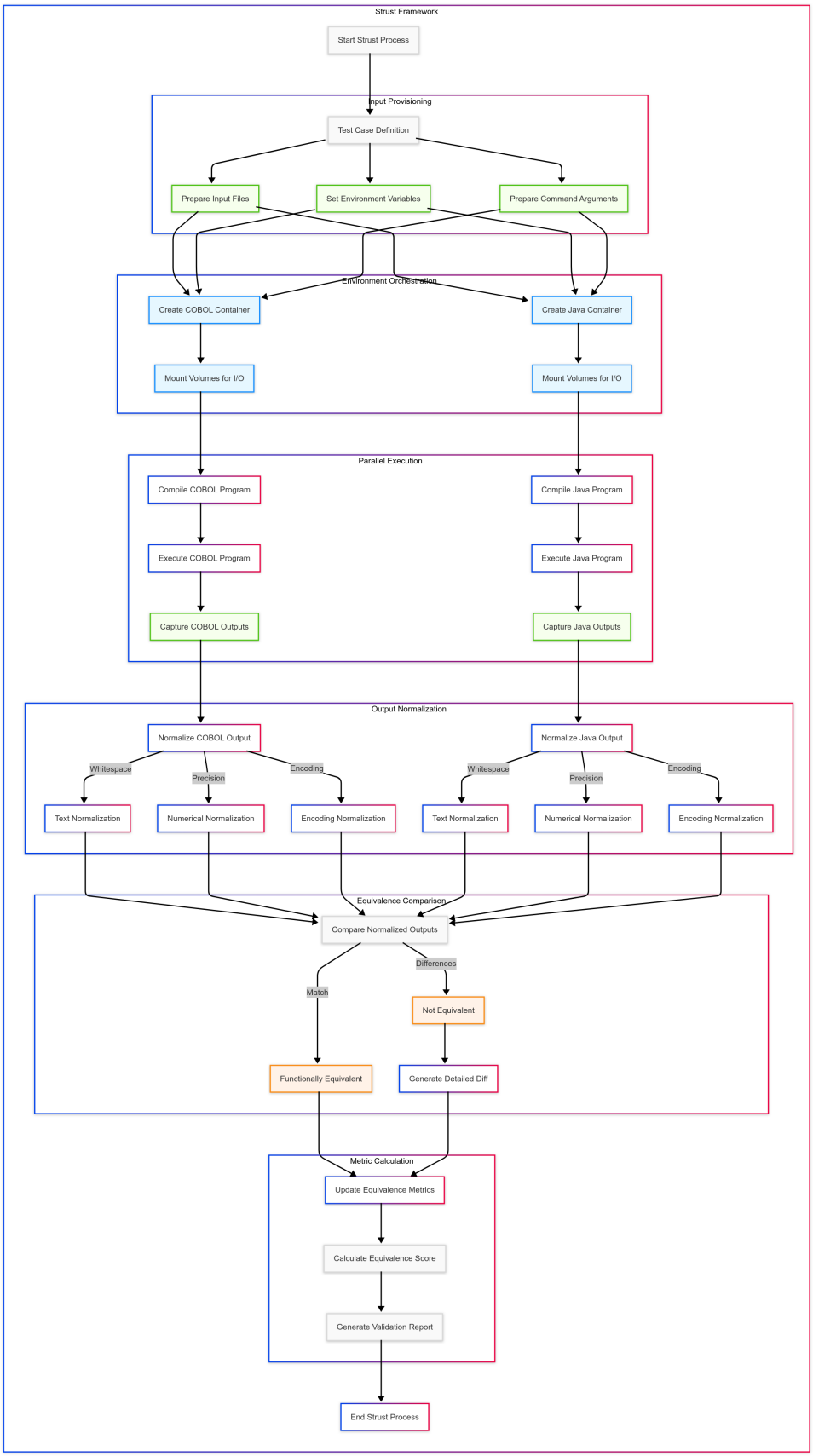

Fig. 1. Strust Workflow Diagram showing the process flow from test case definition through metric calculation. The workflow illustrates parallel execution paths for COBOL and Java with comparison of outputs to determine functional equivalence.

The reporting component generates structured output documenting test results. Summary reporting provides high-level pass/fail status across all test cases. Detailed reporting shows specific differences for failed tests, including line numbers and actual content discrepancies.

D. Example Walkthrough

A concrete example demonstrates the Verified Snippet Demonstrator's operation using a simple interest calculation program. The COBOL implementation reads loan information from a file, calculates interest, and writes results to an output file:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INTEREST-CALC.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT LOAN-FILE ASSIGN TO 'loan.dat'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT RESULT-FILE ASSIGN TO 'result.dat'
    ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD LOAN-FILE.
01 LOAN-RECORD.
  05 LOAN-AMOUNT PIC 9(6)V99.
  05 INTEREST-RATE PIC 9(2)V99.
  05 LOAN-TERM PIC 9(2).

FD RESULT-FILE.
01 RESULT-RECORD.
  05 LOAN-AMOUNT PIC 9(6)V99.
  05 INTEREST-RATE PIC 9(2)V99.
  05 LOAN-TERM PIC 9(2).
  05 TOTAL-INTEREST PIC 9(6)V99.

WORKING-STORAGE SECTION.
01 EOF-FLAG PIC X VALUE 'N'.
01 INTEREST-CALC PIC 9(6)V99.

PROCEDURE DIVISION.
MAIN-PARA.
  OPEN INPUT LOAN-FILE
    OUTPUT RESULT-FILE

  PERFORM UNTIL EOF-FLAG = 'Y'
    READ LOAN-FILE
      AT END MOVE 'Y' TO EOF-FLAG
      NOT AT END PERFORM PROCESS-RECORD
    END-READ
  END-PERFORM

  CLOSE LOAN-FILE
    RESULT-FILE
  STOP RUN.

PROCESS-RECORD.
  COMPUTE INTEREST-CALC =
    LOAN-AMOUNT * INTEREST-RATE * LOAN-TERM /
        100

  MOVE LOAN-AMOUNT TO LOAN-AMOUNT OF
      RESULT-RECORD
  MOVE INTEREST-RATE TO INTEREST-RATE OF
      RESULT-RECORD
  MOVE LOAN-TERM TO LOAN-TERM OF RESULT-RECORD
  MOVE INTEREST-CALC TO TOTAL-INTEREST OF
      RESULT-RECORD

  WRITE RESULT-RECORD
  DISPLAY "Processed loan: " LOAN-AMOUNT
      " with rate: " INTEREST-RATE
      " for years: " LOAN-TERM
      " total interest: " INTEREST-CALC.
```

The a version of an equivalent Java implementation provides that same functionality:

```
import java.io.*;
import java.text.DecimalFormat;

public class InterestCalc {
  public static void main(String[] args) {
    try (BufferedReader reader =
        new BufferedReader(new FileReader("loan.
            dat"));
       PrintWriter writer =
        new PrintWriter(new FileWriter("result.
            dat"))) {

      String line;
      while ((line = reader.readLine()) != null)
          {
        // Parse input record
        double loanAmount =
          Double.parseDouble(line.substring(0, 8)
              );
        double interestRate =
          Double.parseDouble(line.substring(8,
              12));
        int loanTerm =
          Integer.parseInt(line.substring(12, 14)
              );

        // Calculate interest
        double totalInterest =
          loanAmount * interestRate * loanTerm /
              100;

        // Format output record
        DecimalFormat df = new DecimalFormat("
            000000.00");
        String resultRecord =
          df.format(loanAmount) +
          df.format(interestRate) +
          String.format("%02d", loanTerm) +
          df.format(totalInterest);

        // Write result
        writer.println(resultRecord);

        // Display processing message
        System.out.println("Processed loan: " +
            loanAmount +
                " with rate: " + interestRate +
                " for years: " + loanTerm +
                " total interest: " +
                    totalInterest);
      }
    } catch (IOException e) {
      System.err.println("Error processing file:
          " + e.getMessage());
```

```
    }
  }
}
```

The test uses a sample input file containing loan records:

```
010000.005.0010
020000.007.5015
005000.006.2005
```

Execution proceeds through the following steps:

1. The orchestrator creates Docker containers for COBOL and Java environments. 2. Input files transfer to both containers. 3. The COBOL container compiles and executes the COBOL program. 4. The Java container compiles and executes the Java program. 5. Output files and console output retrieve from both containers. 6. Normalization applies to both outputs. 7. The comparison engine checks for equivalence between normalized outputs. 8. The reporting component generates results showing successful equivalence.

The demonstration confirms that both implementations produce identical outputs for the inputs, validating functional equivalence. This example illustrates the core capability of the Verified Snippet Demonstrator to detect whether different language implementations behave identically under controlled conditions.

V. Evaluation and Results

A. Experimental Setup

The evaluation of the Verified Snippet Demonstrator used a controlled environment to assess its effectiveness in detecting functional equivalence. The test environment comprised a Linux Ubuntu 20.04 host with Docker Engine 20.10.8, running on a system with Intel Xeon E5-2670 processor and 16GB RAM. The COBOL container utilized GnuCOBOL 3.1.2 with default configuration settings. The Java container used OpenJDK 11.0.11 with standard class library.

The test suite had ten COBOL programs covering fundamental language features and common business processing patterns. These programs exercised arithmetic operations, conditional logic, iteration, string manipulation, file I/O, and data structure handling. The selection represented typical mainframe application components rather than edge cases or obscure language features. Each program implemented a self-contained business function with clearly defined inputs and outputs.

Each COBOL program paired with a manually created Java implementation designed for functional equivalence. The Java code implemented identical business logic while following Java language conventions and best practices. This approach simulated the expected output of an ideal conversion tool while allowing controlled introduction of discrepancies for testing the detection of the framework.

Test data design covered normal cases, boundary conditions, and edge cases for each program. Input files are various record formats, field combinations, and data values to exercise different execution paths. The test approach emphasized both positive cases (expected equivalence) and negative cases (intentionally introduced discrepancies) to validate both the detection capability and false positive rate of the framework.

B. Evaluation Criteria

The evaluation was focused on the Verified Snippet Demonstrator against defined criteria focused on functional rather than performance optimization. The primary metric measured successful execution and correct functional equivalence determination for each test case. This binary pass/fail metric evaluated whether the framework correctly identified equivalent and non-equivalent implementations.

Secondary metrics included execution overhead introduced by containerization compared to native execution. Time measurements captured container startup duration, compilation time, execution time, and comparison processing. Memory utilization tracking recorded peak memory consumption during different phases of the testing process.

False positive and false negative rates showed quality indicators for the comparison engine. False positives happen when the system incorrectly flags equivalent outputs as different. False negatives occur when the system fails to detect actual functional differences between implementations. These metrics measure the reliability of the equivalence assessment.

C. Results

Table 1 presents the results from the test suite execution, showing the effectiveness of the Verified Snippet Demonstrator in detecting functional equivalence and discrepancies.

TABLE I
Test Results Summary

ID	Program Type	Expected	Result	Notes
T1	Arithmetic	Equivalent	Pass	Basic calculations
T2	String Processing	Equivalent	Pass	Concatenation, substring
T3	Conditional Logic	Equivalent	Pass	Complex conditions
T4	Iteration	Equivalent	Pass	PERFORM VARYING
T5	File I/O	Equivalent	Pass	Sequential processing
T6	Numerical Precision	Non-Equiv	Pass	Detected precision loss
T7	Order Dependency	Non-Equiv	Pass	Detected sequence difference
T8	Error Handling	Non-Equiv	Pass	Different exception behavior
T9	Rounding Behavior	Non-Equiv	Fail	Missed minor difference
T10	Character Encoding	Equivalent	Fail	False positive on EBCDIC

The framework correctly identified functional equivalence in five genuine equivalent pairs (T1-T5) and detected discrepancies in three non-equivalent pairs (T6-T8). Two cases produced incorrect results: T9 failed to detect a subtle rounding difference in financial calculations, and T10 incorrectly flagged an equivalent program pair as different due to character encoding normalization issues.

Overall, the Verified Snippet Demonstrator achieved an 80% success rate in correctly identifying functional equivalence or non-equivalence. The two failure cases highlighted specific areas requiring enhancement: numerical comparison tolerance and character encoding normalization. These findings provide clear direction for refinement of the comparison engine.

The sample output report for a test case with detected discrepancies illustrates the information by the framework:

```
TEST CASE ID: T6 (Numerical Precision)
STATUS: FAIL - Outputs not equivalent

--- Output Difference Report ---
File: result.dat
Line 3:
 COBOL: 005000.006.2005000616.00
 Java: 005000.006.2005000615.87
           ^^

Difference detected in numerical output.
COBOL preserves exact decimal calculation
while Java floating-point introduces
minor precision loss.

--- End Report ---
```

This detailed reporting allows users to understand the specific nature of functional discrepancies, distinguishing between business logic differences and minor technical variations.

D. Validation of Feasibility

The experimental results demonstrate the technical feasibility of using containerized differential testing for verifying functional equivalence between COBOL and Java implementations. The Verified Snippet Demonstrator successfully executed the core workflow of the Strust framework, including environment isolation, parallel execution, output capture, normalization, comparison, and reporting.

The 80% accuracy rate in equivalence detection provides a strong baseline for further refinement. The results verify that Docker containerization creates suitable isolation for controlled execution comparison while still enabling efficient test execution. The identified limitations in the current implementation represent implementation details rather than fundamental methodological flaws, indicating that the core approach remains sound.

The proof-of-concept confirms that automated functional equivalence testing for COBOL-to-Java conversion is achievable through the Strust. This validation creates the foundation for expanding the framework to address more complex scenarios and integration with production conversion tools.

VI. Discussion

A. Interpretation of Results

The experimental results validate the core mechanism of containerized differential testing for functional equivalence verification. The successful detection of both equivalence and non-equivalence across multiple test cases confirms that the approach can reliably identify conversion quality issues. The 80% accuracy rate demonstrates effectiveness while highlighting specific areas for refinement.

Docker containerization proved effective for environment isolation and reproducibility in this context. The containers successfully eliminated platform-specific variations that might obscure genuine functional differences. The containerized approach enabled consistent execution environments for both COBOL and Java without requiring specialized hardware or complex infrastructure.

Output comparison sensitivity analysis revealed important insights regarding normalization requirements. The framework demonstrated high sensitivity to numerical precision differences, correctly identifying subtle calculation variations that might impact financial applications. Character encoding normalization requires enhancement to avoid false positives, particularly for applications using non-ASCII character sets. These findings inform specific improvements for the comparison engine.

B. Implications and Potential Impact

Strust fills a gap in modernization validation by providing objective metrics for conversion quality. This framework has different cases that can change how larger companies approach legacy modernization projects.

As a quality assurance gate, Strust integrates into CI/CD pipelines to provide continuous validation of converted code. This integration helps with the early detection of conversion issues when they remain inexpensive to fix. The objective metrics establish clear quality thresholds for acceptance decisions based on evidence rather than subjective assessment.

For tool benchmarking, Strust is an objective comparison between different automated conversion tools or service providers. Organizations can evaluate multiple options using standardized metrics before committing to a specific approach. This capability introduces market transparency that drives quality improvement across the conversion tool ecosystem.

Risk reduction is the largest benefit for this entire project. Strust provides quantifiable confidence metrics regarding conversion quality before production deployment. This evidence-based approach reduces the risk of business disruption from undetected conversion errors and supports more informed risk management decisions.

For consultancies and service providers, Strust is a mechanism to demonstrate conversion quality to clients with objective evidence. This capability creates competitive differentiation by quantifying service quality rather than relying solely on reputation or subjective assessments.

C. Advantages over Existing Approaches

Strust provides several distinct advantages compared to existing conversion validation approaches. This framework offers objectivity and independence by establishing a standardized assessment framework separate from any specific conversion tool. This separation eliminates conflicts of interest in quality assessment and creates consistent evaluation across different tools and projects.

Automation potential significantly reduces the manual effort required for validation compared to traditional testing approaches. The framework makes automated execution of thousands of test cases without human intervention, scaling to large codebases while maintaining consistent assessment quality. This automation reduces both cost and time requirements for comprehensive validation.

Dynamic behavior focus distinguishes Strust from static analysis tools that examine code structure without verifying runtime behavior. By executing code and comparing actual outputs, the framework detects functional discrepancies that static analysis might miss, including subtle semantic differences and execution path variations. This dynamic approach aligns perfectly with the primary concern in modernization: preserving business functionality.

Standardization potential is a significant long-term advantage of the Strust approach. By establishing a common methodology and metrics for functional equivalence, the framework will work for industry-wide standards for conversion quality assessment. This standardization benefits all stakeholders in the modernization ecosystem by creating shared quality expectations and measurement approaches.

D. Addressing Potential Skepticism

Several legitimate concerns require acknowledgment regarding the Strust approach. The framework cannot guarantee 100% coverage of all possible execution paths or inputs. Test case selection remains important to effective validation, requiring domain expertise to identify important scenarios. Strust addresses this limitation by focusing on risk reduction rather than absolute guarantees, significantly improving validation coverage compared to manual approaches.

Some critics might question the ability to recreate mainframe environmental conditions in containers. While perfect replication remains challenging and behind enterprise walls, the containerized approach creates consistent environments for comparative testing that can easily be depoloyed once in the walled garden. The focus on functional equivalence rather than exact platform replication mitigates this concern by emphasizing business outcomes rather than implementation details.

Strust complements rather than replaces other testing and analysis techniques. The framework works alongside unit testing, integration testing, static analysis, and manual code review to provide comprehensive quality assessment. This complementary approach leverages the strengths of each technique while addressing the specific challenge of functional equivalence verification.

VII. Limitations

A. Scope of the PoC

The Verified Snippet Demonstrator implements a limited subset of the complete Strust framework. The proof-of-concept focuses on self-contained programs without complex inter-program calls or mainframe subsystem dependencies. This limitation restricts the current applicability to simpler conversion scenarios rather than enterprise-scale systems with intricate dependencies.

The implementation supports a subset of COBOL language features including basic arithmetic, conditional logic, iterations, and simple file I/O. Advanced language features including CICS commands, database access, dynamic program calls, and complex data structures remain outside the current scope. This restriction limits validation to core language constructs rather than comprehensive application behavior.

The current implementation uses manually created Java code rather than integrating with commercial conversion tools. This approach demonstrates the comparison framework but does not validate interaction with actual conversion products. Integration with production conversion tools remains necessary for practical application in real modernization projects.

B. Environmental Complexity

The most significant limitation involves handling complex mainframe dependencies that influence application behavior. Mainframe applications frequently depend on subsystems including CICS (Customer Information Control System), IMS (Information Management System), JES (Job Entry Subsystem), and specialized utilities. These dependencies create significant challenges for containerized replication of the execution environment.

The proof-of-concept cannot currently handle applications that depend on CICS transactions, IMS databases, DB2 SQL operations, VSAM file access, or Assembler routine calls. These subsystems define important behavioral characteristics of mainframe applications that require specialized emulation or mocking. The lack of support for these features restricts application to standalone COBOL programs rather than integrated enterprise applications.

Replicating exact mainframe runtime behavior presents ongoing challenges for the containerized approach. Subtle differences in execution order, resource management, and error handling between mainframe and distributed environments can influence program behavior in ways difficult to isolate from conversion issues. These environmental factors require careful management to ensure that detected differences reflect actual conversion problems rather than platform variations.

C. Test Case Generation and Coverage

The current implementation relies on manually created test inputs, limiting practical coverage for complex applications. Automated test data generation remain undeveloped in the proof-of-concept, requiring significant manual effort to create comprehensive test suites. This limitation constrains scalability for large codebases with numerous execution paths.

The framework faces the ”oracle problem” common to all testing approaches: determining correct outputs requires existing knowledge or reference implementations. Strust tries to fix this challenge by using original COBOL programs as the oracle, but this approach assumes correctness of the source programs. Errors in original COBOL code propagate through the comparison process, potentially flagging correct Java conversions that fix legacy bugs as ”non-equivalent.”

Coverage analysis remain limited in the current implementation. The framework lacks mechanisms to measure which portions of code execute during testing or to identify untested execution paths. This limitation creates risk that critical paths might remain untested despite high passage rates on included test cases.

D. Output Comparison Complexity

The comparison engine currently implements basic equivalence checking that faces challenges with complex outputs. Binary file comparison lacks content-aware parsing for proprietary formats, limiting validation to exact matching rather than semantic equivalence. Database state comparison remains unimplemented, preventing validation of programs that modify persistent storage.

Floating-point handling presents specific challenges for numerical applications. The current implementation uses simple epsilon-based comparison that can produce false positives or negatives for complex calculations. Financial applications with precise decimal requirements need enhanced comparison logic to distinguish between significant and insignificant numerical variations.

Non-deterministic outputs including timestamps, random numbers, or machine-specific identifiers create comparison challenges. The current normalization approach handles basic cases but lacks sophisticated pattern matching for complex non-deterministic elements. This limitation requires manual exclusion of such elements from comparison, increasing configuration complexity.

E. Performance Equivalence

Strust focuses exclusively on functional equivalence without addressing performance characteristics. The framework does not currently measure or compare execution time, resource utilization, or scalability between COBOL and Java implementations. This limitation prevents validation of performance requirements that might be critical for time-sensitive applications.

Performance differences between mainframe and distributed environments complicate direct comparison of execution metrics. Operations that perform efficiently on mainframe hardware might show different characteristics on x86 platforms running Java. These platform-specific performance patterns require specialized analysis beyond the current implementation's capabilities.

F. Scalability

The proof-of-concept does not demonstrate scalability to enterprise-scale applications with thousands of programs and complex dependencies. Container startup overhead becomes significant when testing large numbers of small programs. The current implementation lacks parallelization to execute multiple test cases simultaneously, limiting throughput for comprehensive test suites.

Resource consumption increases substantially with program complexity and test case volume. The containerized approach requires significant disk space for Docker images and runtime memory for container execution. These resource requirements may constrain application to environments with substantial computing resources available.

The current implementation lacks integration with continuous integration systems for ongoing validation. This limitation stops the automated validation as part of development workflows, restricting application to periodic assessment rather than continuous quality monitoring.

VIII. Future Work

A. Expanding COBOL Feature Support

Future development incrementally expand support for additional COBOL language features and mainframe constructs. Implementation priorities include complex data structures such as REDEFINES clauses, level 88 conditions, and OCCURS DEPENDING ON tables. Control flow enhancements add support for ALTER statements, GO TO with DEPENDING ON, and complex PERFORM structures.

Interprogram communication support enable testing of CALL statements with parameter passing between programs. This capability requires tracking program linkage and simulating the execution stack across multiple programs. Implementation include both static and dynamic call patterns common in mainframe environments.

Subsystem API support is a critical enhancement for enterprise applications. Future versions create mocking frameworks for CICS commands, IMS calls, and DB2 SQL operations. These frameworks emulate subsystem behavior sufficiently to validate application logic that depends on these interfaces.

B. Enhancing Environmental Simulation

Environmental simulation enhancements focus on improving mainframe subsystem emulation. Integration with commercial emulation platforms such as Micro Focus Enterprise Server or IBM ZD&T provide more accurate replication of mainframe behavior. Custom container images package these emulation environments for consistent deployment.

EBCDIC/ASCII handling improvements address character encoding challenges through comprehensive translation layers. The enhanced implementation handle EBCDIC-specific behaviors including collating sequences, special characters, and data representation variations. Improved normalization eliminate false positives caused by encoding differences.

Job control language (JCL) interpretation enable testing of program execution parameters defined in JCL scripts. This capability parse JCL to extract program arguments, environment settings, and file allocations for container configuration. The implementation support common JCL constructs used to define program execution contexts.

C. Automated Test Data Generation

Automated test data generation significantly help coverage and reduce manual effort. Implementation apply control flow analysis to COBOL source code to identify execution paths and generate inputs targeting each path. Path condition extraction determine input constraints necessary to exercise specific code segments.

Boundary value analysis automation identify data ranges and generate test cases at boundaries and edge conditions. This technique systematically exercise limit conditions that often reveal conversion discrepancies. Implementation support both simple variables and complex data structures.

Production data sampling techniques enable using anonymized production data while maintaining privacy compliance. Implementation provide data masking and

transformation to create realistic test cases based on actual business data patterns while protecting sensitive information.

D. Sophisticated Output Comparison

Database state comparison create schema-aware differential analysis for relational databases. This capability compare table contents, constraints, and relationships between COBOL and Java execution results.

Format-aware comparison add parsers for common output formats including reports, XML, JSON, and proprietary record layouts. These parsers enable semantic comparison that identifies meaningful differences while ignoring formatting variations. Implementation include configuration options for format-specific comparison rules.

Tolerance configuration changes provide fine-grained control over acceptable variations. Users define domain-specific rules for numerical precision, ordering significance, and pattern matching. These configurations reduce false positives while maintaining detection sensitivity for significant discrepancies.

E. Metric Refinement and Dashboarding

Metric enhancements develop more nuanced scoring beyond simple pass/fail assessment. Implementation include weighted scoring based on business criticality, code complexity, and execution frequency. These weighted metrics provide more meaningful quality indicators aligned with business impact.

Coverage integration combine functional equivalence results with code coverage data to contextualize findings. This integration highlight untested regions and prioritize testing efforts based on risk analysis. Implementation support common coverage formats from both COBOL and Java analysis tools.

Visualization and dashboarding present results through interactive interfaces for stakeholder communication. Implementation include trend analysis showing quality improvement over time, drill-down for root cause analysis, and executive summaries for project governance.

F. Performance Comparison

Performance measurement extend the framework beyond functional equivalence. Implementation capture execution timing with microsecond precision in both environments to identify performance variations. Profiling integration record CPU utilization, memory consumption, and I/O operations for comparative analysis.

Load testing create validation under various throughput conditions to assess scalability characteristics. This execute programs with progressively increasing volume to identify performance inflection points. Implementation eventually include configurable load profiles simulating different usage patterns.

Resource utilization analysis compare efficiency metrics between COBOL and Java implementations. This analysis identify opportunities for optimization in converted code and highlight potential performance risks. Implementation include baseline comparison against original mainframe metrics where available.

G. Integration and Usability

API development enable integration with development tools and conversion platforms. RESTful interfaces provide programmatic access to testing for automation workflows. Implementation include webhooks for event notification and results retrieval to support CI/CD integration.

Pipeline integration create plugins for common CI/CD platforms including Jenkins, GitHub Actions, and Azure DevOps. These integrations enable automated validation as part of modernization workflows with quality gates based on equivalence metrics. Implementation include configurable threshold enforcement for build acceptance.

H. Scalability and Cloud Deployment

Architecture optimization increase performance for large-scale testing. Parallel execution leverage container orchestration to process multiple test cases simultaneously. Resource management improvements reduce container overhead through pooling and reuse strategies. These optimizations improve throughput for comprehensive testing.

Cloud deployment templates enable execution on major platforms including AWS, Azure, and GCP. Implementation include infrastructure-as-code definitions for consistent deployment across environments. Auto-scaling adjust resources based on testing volume to optimize cost and performance.

Distributed execution enable geographic distribution of testing workloads for global organizations. Implementation coordinate test execution across multiple regions while maintaining centralized reporting and governance. This capability create follow-the-sun testing models and regional compliance requirements.

I. Extending Language Support

The Strust desgin would applies to other legacy language modernization efforts beyond COBOL. Future work make the framework support PL/I-to-Java conversion through similar containerized differential testing approaches. This extension require PL/I-specific containers and normalization rules while leveraging the existing comparison engine.

RPG address IBM i (AS/400) modernization projects through specialized containers for RPG compilation and execution. This capability enable verification of RPG-to-Java conversions using the same differential testing with language-specific adaptations.

Natural, CA Gen, and other fourth-generation languages represent additional expansion opportunities. Each language require specific runtime environments and comparison configurations while building on the core Strust framework. These extensions address the broader legacy modernization market beyond COBOL-specific conversions.

IX. Conclusion

Legacy system modernization presents a critical challenge for organizations operating COBOL systems that power essential business functions. The verification of functional equivalence between original COBOL and converted Java code is the

foremost risk factor in these projects. Traditional verification approaches rely on manual testing and code review, creating substantial cost and quality challenges.

Strust addresses this verification gap through containerized differential testing that provides objective, quantifiable assessment of conversion quality. The approach executes COBOL and Java code in isolated containers, applies identical inputs, and systematically compares outputs to detect functional discrepancies.

The Verified Snippet Demonstrator proof-of-concept successfully validated the feasibility of this approach. Experimental results demonstrated 80% accuracy in detecting both equivalence and non-equivalence across diverse test cases. The implementation confirmed that containerization provides effective isolation for meaningful comparison while having automated execution and analysis.

Current limitations include restricted language feature support, challenges with complex mainframe dependencies, manual test data generation requirements, and basic comparison capabilities. These limitations reflect implementation stage rather than fundamental methodological flaws, with clear paths for furthering in future development.

Strust has potential to change legacy modernization verification through objective, automated functional equivalence assessment. The framework provides a foundation for standardized quality metrics that can reduce project risk, enable tooling comparisons, and increase confidence in modernization outcomes. This contribution fixes a critical need in an industry facing growing pressure to modernize legacy systems while preserving essential business functionality.

REFERENCES

[1] P. Lawson et al., "The COBOL Landscape 2021: Continued Relevance and Modernization Trends," KPMG International Cooperative, Tech. Rep., 2021.

[2] IBM Corporation, "IBM Z Mainframe Platform," IBM Corporation, Tech. Rep., 2023.

[3] G. Olliffe, "Legacy Modernization Demands Evolutionary Strategy," Gartner Research, Tech. Rep. G00756279, 2022.

[4] Micro Focus, "Enterprise Developer Documentation," Micro Focus International plc, Tech. Rep., 2023.

[5] AWS, "AWS Mainframe Modernization User Guide," Amazon Web Services, Inc., Tech. Rep., 2023.

[6] GnuCOBOL Project, "GnuCOBOL 3.1 Documentation," GNU Project, Tech. Rep., 2022.

[7] S. McConnell, "Code Complete: A Practical Handbook of Software Construction," Microsoft Press, Redmond, WA, 2nd ed., 2004.

[8] Docker Inc., "Docker Documentation," Docker Inc., Tech. Rep., 2023.

[9] E. Larsen and D. Evans, "Differential Testing for Software," in Proc. ICSE, 2018, pp. 549-558.

[10] Oracle Corporation, "Java Platform, Standard Edition Documentation," Oracle Corporation, Tech. Rep., 2023.