RUHR-UNIVERSITÄT BOCHUM

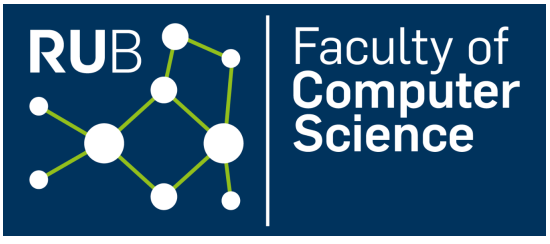
RUB

Automating Test-Case Adaptation and Integration in Fork Ecosystems Using Large Language Models

Keanu-Wesley Schurkus

Bachelor Thesis – September 1, 2025
Chair of Software Engineering.

Supervisor: Prof. Dr. Thorsten Berger
2. Supervisor: Prof. Dr. Yannic Noller
Advisor: Dr. Mukelabai Mukelabai



Abstract

Test-case reuse across forked software projects presents significant opportunities for enhancing software quality and reducing redundant development effort. In practice, however, direct reuse is rare because tests are tightly coupled to the source fork's APIs, package structure, and build configuration; as forks evolve independently, refactorings (e.g., renames, moves), signature changes, and behavioral drift routinely break copy-paste transfers and necessitate targeted adaptations. This thesis introduces an automated methodology leveraging Large Language Models (LLMs) to systematically adapt and integrate unit tests into different project contexts within fork ecosystems. The developed approach comprises pre-build validation, test insertion aligned with the original project structure, an iterative build-adaptation loop guided by detailed compiler feedback, and comprehensive metrics collection for subsequent analysis. Empirical evaluation demonstrates the feasibility of this LLM-assisted adaptation, highlighting its potential to significantly reduce developer effort, enhance test coverage, and improve overall software quality within fork ecosystems.

Contents

1 Introduction

1.1 Background and Motivation

In modern software development, forking has emerged as a critical mechanism to promote collaborative innovation, allowing isolated experimentation and accelerating feature development. Platforms like GitHub facilitate this by enabling developers to create forks of repositories and propose improvements via pull requests. However, while code reuse is common, test cases essential for ensuring software quality are often not propagated across forks, leading to inconsistencies in testing and potential quality degradation. Prior studies on fork ecosystems report that reuse across divergent forks is generally infrequent and is often handled in ad-hoc ways, which compound quality risks and maintenance costs [4]. Beyond simple divergence, integration across forks confronts refactorings, renames, and structural changes that make direct reuse difficult. Empirical work has shown that merge and integration conflicts correlate with post-merge defects, underscoring the importance of robust validation assets such as tests [2]. Within this landscape, the specific opportunity of test-case propagation has been explored by Mukelabai et al., who identified conditions under which tests can transfer across forks and proposed semi-automated propagation strategies [8, 9]. These findings suggest that many projects could benefit from tests added elsewhere in the ecosystem, provided that those tests can be adapted to the target context.

1.2 Problem Statement

The core problem addressed in this thesis is the difficulty of automatically adapting and integrating relevant test cases from one fork to another within an ecosystem. While prior work has shown that reusable tests exist [9], the manual effort and technical challenges involved in modifying these tests to work in a different project context (e.g., due to API drift, refactorings, package/class renames, or behavioral changes) often prevent their propagation. This can leave critical vulnerabilities undetected, undermine software quality, and lead to redundant development efforts [14]. Building on previous work that identified potentially reusable tests [9], this thesis focuses on the next step: automating the adaptation process.

1.3 Goal and Objectives

The goal of this thesis is to explore how Large Language Models (LLMs) can help to automatically adapt and integrate test cases across forked projects by suggesting necessary code modifications and refactorings. The primary objective is to develop and evaluate a prototype tool that demonstrates the feasibility of this LLM-assisted approach. In doing so, the thesis investigates whether LLMs can bridge common forms of divergence observed across forks (e.g., renames, method signature changes, or structural refactorings) and thereby reduce the manual effort required for test reuse [14].

1.4 Novelty and Contributions

The novelty of this research lies in developing an automated workflow that attempts to integrate a given test case, diagnoses failures, and leverages LLM-driven analysis to propose solutions for adapting the test code. This differs from manual approaches or pure static analysis and is motivated by recent advances showing that LLMs can support refactoring and quality improvement tasks [12, 7]. In adjacent areas of integration, learning-based tools have been proposed to reconcile code differences (e.g., MergeBERT, MergeGen, and PLMs for semantic conflict resolution) [13, 3, 15], while structured merge tools such as *Spork* aim to better handle moves and renames in Java [6]. Directly within testing, LLM-based repairing of broken unit tests due to code evolution (e.g., UTFix) demonstrates the potential to adapt tests under change [10]. Moreover, recent agentic approaches show promise in automating build preparation and test execution across arbitrary projects, underlining the practical challenges at the intersection of adaptation and execution [1]. The anticipated contributions are:

- A prototype tool demonstrating the feasibility of LLM-assisted test-case adaptation and integration across forks.
- Empirical insights into the effectiveness and limits of LLMs in handling adaptation complexities common in fork ecosystems (from simple renames to more structural changes).
- Evidence of potential reduction in manual effort for developers seeking to reuse tests, with implications for test coverage and overall software quality.
- A contribution to automated software engineering on test reuse and LLM-driven code modification, complementing research on merge/conflict resolution and test repair.

1.5 Methodological Overview

Methodologically, the thesis focuses on the engineering of a prototype tool that takes potentially reusable test cases (identified through previous work [9]) and attempts to integrate them into a target project. The core is an automated build-and-adaptation loop: the tool integrates the test, performs a build, analyzes compilation or linkage failures, and employs LLM prompts to propose minimal and targeted code edits. Where relevant, the workflow also considers build and execution barriers (e.g., project configuration, dependencies), reflecting practical execution challenges noted in recent agent-based testing work [1]. The empirical evaluation uses a benchmark of fork pairs derived from prior work [9], reporting success rates and classifying adaptation outcomes by difficulty (e.g., direct applicability vs. LLM-required repairs), while relating observations to the adaptation phenomena documented in the literature [8, 4, 2]. The study uses *Compilable* as its main endpoint (tests build; Sec. 2.1.3) and audits functional soundness via a focused spot-check on a subset of results (Sec. 2.1.4).

2 Methodology

This chapter details the methodological approach employed to address the problem of automated test-case adaptation within fork ecosystems. The methodology is structured around an automated workflow leveraging Large Language Models (LLMs), specifically Google Gemini, and comprises four main components: (1) Prebuild validation and environment configuration, (2) Test case insertion and packaging alignment, (3) Iterative build-adaptation loop, and (4) Metrics collection and classification.

2.1 Parameters and Defaults

Table 2.1 lists fixed parameters to ensure reproducibility.

Table 2.1: Experimental parameters and defaults.

Parameter	Value
LLM model	Google Gemini (`Gemini 2.5-flash`)
Sampling	temperature=1.0; top_p=0.95
Retries / attempts	N=3 adaptation attempts per task
Platform	OS = `macOS`; JDK = 1.8; Maven versions = `3.9.11`
Build-system scope	Maven-only in evaluation; Gradle (*experimental* and disabled)
Hardware	CPU = `M2 Pro`, RAM = `16 GB` (for runtime comparability)
Randomness control	Seed = `30` (where applicable)
Success criterion	Compilable test (see Sec. 2.1.3)

2.1.1 Rationale for Parameter Choices

LLM choice: `Gemini 2.5-flash`. I selected Gemini 2.5-flash for its low latency/cost (enabling multiple attempts), large context window (accommodating tests, UUT snippets, and diagnostics), and empirically minimal, patch-style edits in pilot runs and occasional large rewrites.

Sampling: `temperature=1.0, top_p=0.95`. LLM decoding must balance *exploration* (to escape a failing fix) with *stability* (to remain faithful to compiler feedback). I therefore keep `top_p` slightly conservative at 0.95 to trim low-probability tails (reducing erratic, off-distribution code), while setting `temperature` to 1.0 to maintain diversity among attempts. In pilot runs, lower temperatures (≤ 0.5) tended to repeat the same failing fix across attempts (poor exploration), whereas higher-p nucleus or unconstrained sampling increased spurious edits (e.g., renaming public APIs unnecessarily). The chosen pair yielded the most *distinct* yet *diagnostic-aligned* patches across successive attempts, which is exactly what the loop exploits.

Retry budget: N=3 attempts. The cap of three adaptation attempts reflects a pragmatic optimum among **diminishing returns**, **cost/latency**, and **fairness** across tasks. First, most benefits arrive early: in my study, successes among LLM-triggered cases typically occurred by attempt 2, and overall I observed that **a majority of repaired cases completed within at most three attempts**, with the **median success at attempt 2**. Second, each extra attempt incurs a full project rebuild plus an LLM call; beyond three, the marginal gain was small while wall-clock time and token usage rose sharply. Third, fixing N globally preserves comparability across tasks (avoiding giving some tasks effectively more compute). I additionally allow a single controlled re-run in the sensitivity analysis to gauge stochastic variance, but I report primary outcomes under the fixed N=3 budget for reproducibility.

2.1.2 Subjects and Selection Criteria

I selected fork pairs from public Java projects with the following criteria:

- **Java/Maven projects** with a compilable baseline.[1]
- Source test available in the origin fork and a corresponding UUT path in the target.
- No proprietary dependencies required to compile tests.

Exclusion criteria:

- Projects failing a clean pre-build after one automated fix attempt on `pom.xml`.
- Targets requiring credentials or non-public artifacts.
- **Gradle-based targets** (experimental Gradle support not used in this evaluation).

[1] The prototype contains code paths for Gradle detection/builds, but this functionality is *not finished*; Gradle support was disabled for this study.

2.1.3 Outcome Definitions

I distinguish three outcome levels:

1. **Compilable**: the adapted test compiles under the target project without errors.
2. **Executable**: the test runs without infrastructure-level failures ("harness errors", i.e., issues from the test runner, plugins, or classpath rather than assertion outcomes). This criterion was not enforced in the main study.
3. **Sound**: the adapted test preserves the original testing intent.

Unless stated otherwise, "success" refers to the *Compilable* level. I report *Executable* and *Sound* assessments where available.

2.1.4 Manual Soundness Assessment (Spot-Check)

To estimate whether compilable adaptations preserved the original testing intent, I conducted a manual spot-check on a stratified subset of *LLM-affected successes* (i.e., cases that required the LLM to make the test compilable).

Decision rule. I deem an adapted test **Sound** if the following hold: (i) the *oracle equivalence* criterion is satisfied (the target test asserts the same property as the source test: same value/exception/relation, allowing for benign renamings and small API differences), and (ii) the test is *non-vacuous* (it would fail if the property were made false). If intent is preserved but scope is slightly reduced due to missing minor APIs or removed ancillary checks, I record **Probably-sound**. Otherwise, I record **Unsound**.

Procedure. For each source→target pair in the sample, I (1) compared scenario/inputs, (2) compared focal calls (class.method under test), and (3) compared the oracle(s) (asserted value/exception/relation). Where feasible, I performed a light non-vacuity check by temporarily negating the central assertion to verify the test fails. I documented a one-line justification with each verdict. Detailed worksheets are provided in Appendix .1.

2.1.5 Handling Stochasticity

LLM outputs are stochastic. I fix sampling parameters (Table 2.1) and cap attempts at N=3. To assess variance, I permit at most one controlled re-run for failures in the sensitivity analysis and report any deltas separately.

2.1.6 Alignment with Research Questions

RQ1 evaluates first-fix efficacy after failed insertion (metrics: success at attempts 2–3). RQ2 targets basic refactorings (tracked by clone-type labels and resolved diagnostics). RQ3 concerns complex edits (Type-3/4) and their success rates. RQ4 compares against a direct-copy baseline (attempt 1, no LLM). All metrics are logged per task for stratified analysis by clone type and attempt index.

2.2 Experimental Procedure

The end-to-end procedure is summarized below; implementation details (CLI invocations, regex diagnostics, prompt construction, file I/O) are described in Chapter 3 (see in particular Sec. 3.2.3 and Sec. 3.2.4).

1. **Pre-build check.** Detect the build system; in this study I **restrict execution to Maven** (Gradle code path left disabled). Compile the target project in isolation; if enabled, attempt a single LLM-guided `pom.xml` fix. If the project remains non-compilable, exclude it (Sec. 2.1.3).
2. **Test placement.** Copy the source test into `src/test/java/...` of the target, preserving the package declaration.
3. **Iterative adaptation (up to N=3 attempts).**
   a) Compile; if success, stop and record the attempt index.
   b) Parse compiler diagnostics (unresolved symbols/imports, signature mismatch, package/class issues).
   c) Build an LLM prompt from {original test, UUT snippet(s), diagnostics}; apply the returned patch to the test and retry.
4. **Instrumentation.** Log per-attempt results, timings, and (when applicable) clone-type labels into a JSON metrics file for later analysis (Chapter 4).

For completeness, the algorithmic sketch is:

```
for task in tasks:
  ensure_prebuild_ok(target) or exclude
  place_test(source_test -> target/src/test/java/...)
  for i in 1..N:
    ok, diag = compile(target)
    if ok: record_success(i); break
    patched = LLM_adapt({test, UUT, diag})
    write_test(patched)
  if not ok: record_failure(N)
```

2.3 Baseline and Analysis Plan

Baseline. I define a direct-copy baseline as attempt 1 without LLM involvement; improvements are counted as additional successes achieved at attempts 2–3. If attempt 1 fails for any reason, the baseline outcome for that task is "fail". I use an intrinsic baseline: attempt 1 (no LLM). The external baseline against manual propagation from ASE'23, as planned, is deferred due to time; thus I limit claims to improvements over direct-copy within our cohort.

Analysis. I report proportions with 95% Wilson intervals, medians and ranges for times, and stratify results by attempt index and clone-type label. Given the sample size, I refrain from null-hypothesis significance testing and emphasize effect sizes and intervals.

3 Implementation

This chapter details the architecture and concrete implementation of the prototype tool for automating test-case adaptation in fork ecosystems. Beyond summarizing the core modules and data flow, it dives into design decisions, failure handling, and guardrails that proved crucial in practice. Short, illustrative code excerpts are included for key driver scripts used during bulk evaluation. The codebase contains Gradle-specific stubs (detection, error parsing, build invocation), but Gradle support is *experimental* and was *not* exercised in the evaluation; all experiments ran on Maven projects.

3.1 System Architecture Overview

The adaptation logic is implemented in Python under `src/`, organized into:

- `main.py`: Orchestrates the end-to-end workflow (pre-build, insertion, adaptation loop, metrics).
- `java_env_manager.py`: Detects build system (Maven/Gradle), enforces JDK 8, invokes builds.
- `utils.py`: Helpers for path normalization, regex error parsing, GitHub fetch, code-block extraction.
- `llm_analyzer.py`: Constructs prompts and parses LLM responses for adaptation and build fixes.
- `metrics_tracker.py`: Collects and persists adaptation metrics to JSON.

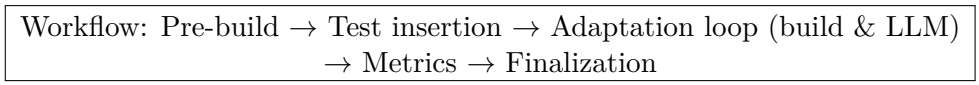


Figure 3.1: High-level architecture and data flow.

File layout and data artifacts. Each adaptation run manipulates only a *working copy* of the target repository. Persistent artifacts are: (i) the inserted test file (created under `src/test/java/...`), (ii) `adaptation_metrics.json` with attempt-level telemetry, and (iii) optional `pom.xml` patches if the pre-build configuration-fix feature is enabled (§3.2.1).

3.2 Core Modules

3.2.1 Environment Detection and Pre-Build Checks

`main.py` first establishes the build environment:

1. **Build-system detection.** We prefer Maven (presence of `pom.xml`); a Gradle branch exists but is disabled for evaluation to avoid confounds across toolchains.

2. **JDK pinning.** The tool constrains the Java toolchain to JDK 8 for reproducibility.[1]

3. **Baseline build.** A clean *test-compile* (`mvn clean test-compile -DskipTests`) ensures the target compiles prior to any modification.

4. **Optional config fix.** If the baseline fails for configuration reasons (e.g., missing `maven-compiler-plugin`, invalid `maven-surefire-plugin` version, source/target mismatch), the tool can synthesize a *minimal patch* via the LLM. The patch is applied only if it parses as XML, contains a single plugin or property change, and round-trips through an XML validator. This guardrail avoids overfitting the build file to the inserted test case.

All outcomes are recorded via `metrics_tracker.GlobalMetrics` with timestamps and return codes.

3.2.2 Test Case Insertion

If the project builds cleanly:

- The source test is fetched (local path or GitHub raw content) and its *declared package* guides the insertion path under `src/test/java/...`.

- A safety check rejects overwriting existing files unless `-force` is set.

- The insertion step verifies that the resulting path is within the repo root (protecting against path traversal in malformed packages).

[1] Newer JDKs often tighten module encapsulation and deprecations, confounding build outcomes across projects.

3.2.3 Iterative Build–Adaptation Loop

The loop attempts up to N times (default 3):

1. **Rebuild** and capture diagnostics (stdout/stderr).
2. **Parse errors** with build-aware extractors (e.g., cannot find symbol, incompatible types, missing imports, changed signatures).
3. **Assemble context** by reading the current test and the relevant UUT file(s).
4. **Prompt the LLM** with a constrained template (Section 3.4.2).
5. **Validate and apply** only if a single Java class is extractable; non-code tokens are ignored.
6. **Record metrics** per attempt (error kind, response length, success).

3.2.4 Metrics Tracking

`metrics_tracker.py` persists:

- Pre-build status and whether a configuration fix was attempted/applied.
- Per-attempt error types, LLM response length, compilation outcome.
- Final disposition (success/failure), total attempts, and wall-clock durations.

At exit, `finish_tracking()` writes `adaptation_metrics.json`. Metrics are designed to be *append-only* for idempotence across retries.

3.3 Algorithms (Pseudocode)

Algorithm 1 End-to-End Adaptation Orchestrator (`main`)

Require: source test code T, source test path p_s, target root R, UUT relpath u, max attempts N

$bs \leftarrow \textsc{DetectBuildSystem}(R)$ ▷ Maven expected; Gradle disabled
$(c, out, err) \leftarrow \textsc{Build}(R, bs)$
if $c \neq 0$ **then**
    if ConfigFixAllowed() **then**
        $P \leftarrow \textsc{ConstructPomFixPrompt}(err, \texttt{pom.xml})$
        $\hat{X} \leftarrow \textsc{QueryLLM}(P)$
        if $\textsc{ExtractXML}(\hat{X})$ **then**
            $\textsc{ApplyPatch}(\texttt{pom.xml}, \hat{X})$
            $(c, out, err) \leftarrow \textsc{Build}(R, bs)$
        end if
    end if
    if $c \neq 0$ **then**
        return RecordPreBuildFail()
    end if
end if
$t \leftarrow \textsc{InsertTest}(T, p_s, R)$
$success \leftarrow \textsc{AdaptationLoop}(t, R, u, bs, N)$
if $\neg success$ **and** CleanupOnFailure() **then**
    $\textsc{RestoreOrRemoveInsertedTest}(t)$
end if
return FinishTracking()

Algorithm 2 Iterative Build–Adaptation Loop

Require: test file t, target root R, UUT relpath u, build system bs, max attempts N

$C \leftarrow \textsc{ReadFile}(t)$
for $i = 1$ **to** N **do**
  $(c, out, err) \leftarrow \textsc{Build}(R, bs)$
  if $c = 0$ **then**
    $\textsc{RecordAttempt}(i, success = true)$; **return** true
  end if
  $E \leftarrow \textsc{ParseErrors}(out \textbf{ or } err, bs)$
  $U \leftarrow \textsc{ReadFile}(R \| u)$
  $P \leftarrow \textsc{ConstructAdaptPrompt}(C, E, U)$
  $\hat{C} \leftarrow \textsc{QueryLLM}(P)$; $C' \leftarrow \textsc{ExtractJava}(\hat{C})$
  if $C' = \textsc{None}$ **then**
    $\textsc{RecordAttempt}(i, success = false)$; **break**
  end if
  $\textsc{WriteFile}(t, C')$; $C \leftarrow C'$
  $\textsc{RecordAttempt}(i, success = false)$
end for
return false

3.4 Design Details and Engineering Trade-offs

3.4.1 Build Invocation and Guardrails

Maven is invoked via `java_env_manager.invoke_maven_build`, defaulting to `clean test-compile -DskipTests`. We intentionally avoid running tests during adaptation to (i) reduce flakiness, and (ii) make compilation the decisive gate. For reproducibility, environment variables `JAVA_HOME` (JDK 8) and `MAVEN_OPTS` are pinned when set by the driver.

3.4.2 Prompt Construction and Response Extraction

`llm_analyzer.py` composes prompts with four blocks: (i) a concise instruction framing the task as *adapting* (not re-implementing) the test; (ii) the failing test; (iii) the UUT excerpt; and (iv) a truncated build log (first/last k lines). The model is asked to return *only one Java class* inside a code fence. The extractor accepts:

- A single "‘`java` fenced block; alternative fences are rejected.
- Files with a matching `package` header if present in the original test.

- ASCII-only output to avoid invisible characters corrupting the file.

If extraction fails, the attempt is recorded and the loop terminates early to preserve the baseline test for analysis.

3.4.3 Safety, Idempotence, and Cleanup

Two invariants keep runs analyzable:

1. **Idempotent builds:** the driver may re-run the same row; metrics append with a fresh UUID but avoid overwriting prior JSON.
2. **Conservative edits:** only the inserted test file and, optionally, `pom.xml` can change; the driver blocks edits to `src/main` to prevent "repairing" production code.

On failure, `CleanupOnFailure` controls whether the inserted test is deleted or left for postmortem inspection.

3.5 Illustrative Code Excerpts

Orchestrator (`main.py`): loop skeleton

Listing 3.1: Adaptation loop (simplified)

```
def adaptation_loop(test_file, target_root, class_relpath, build_system,
                    max_attempts, api_key):
    current_code = open(test_file, encoding="utf-8").read()
    for attempt in range(1, max_attempts + 1):
        code, out, err = invoke_build(target_root, build_system)
        if code == 0:
            global_metrics.record_adaptation_attempt(attempt, True)
            return True

        # Parse diagnostics (Maven vs. Gradle)
        try:
            parsed = parse_build_error(out or err, build_system)
            err_msg = parsed.get("raw_message") or str(parsed)
        except Exception:
            err_msg = err or out

        # Gather UUT, construct prompt, ask LLM
        cls_path = os.path.join(target_root, class_relpath)
        target_code = open(cls_path, encoding="utf-8").read()
        prompt = construct_llm_prompt(
            original_test_case_code=current_code,
            parsed_build_error=err_msg,
            target_class_code=target_code,
            target_class_name=os.path.basename(cls_path),
            build_file_content=None,
            build_file_name="",
        )
        suggestion = query_llm(prompt, api_key)
        new_test = extract_java_code_from_llm_response(suggestion)
        if not new_test:
            break
        open(test_file, "w", encoding="utf-8").write(new_test)
        current_code = new_test

    code, out, err = invoke_build(target_root, build_system)
        if code == 0:
            global_metrics.record_adaptation_attempt(attempt, True)
            return True

    global_metrics.record_adaptation_attempt(attempt, False)
    return False
```

Build detection and Maven invocation (`java_env_manager.py`)

Listing 3.2: Build system detection and Maven build

```
def detect_build_system(project_dir: str) -> str:
    if os.path.exists(os.path.join(project_dir, "pom.xml")):
        return "maven"
    if os.path.exists(os.path.join(project_dir, "build.gradle")):
        return "gradle"
    return "unknown"

def invoke_maven_build(project_dir: str, command=None):
    command = command or ["mvn", "clean", "test-compile", "-DskipTests"]
    proc = subprocess.run(command, cwd=project_dir,
                          capture_output=True, text=True)
    return proc.returncode, proc.stdout, proc.stderr
```

Error parsing (Maven) (`utils.py`)

Listing 3.3: Regex-oriented Maven error parsing

```
def parse_maven_error(error_output: str) -> dict:
    """
    Parses Maven error output to extract key error information.
    """
    if not error_output:
        return {"error_type": "unknown", "message": "No error output
            provided"}

    lines = error_output.strip().split("\n")

    first_error_index = -1
    for i, line in enumerate(lines):
        s = line.strip()
        if s.startswith("[ERROR]") and (
            "Failed to execute goal" in s
            or "COMPILATION ERROR" in s
            or "cannot find symbol" in s
            or "Compilation failure" in s
            or "Inkompatible Typen" in s
            or "Symbol nicht gefunden" in s
        ):
            first_error_index = i
            break

    if first_error_index == -1:
        for i, line in enumerate(lines):
            if line.strip().startswith("[ERROR]"):
```

```
            first_error_index = i
            break

    if first_error_index == -1:
        return {"error_type": "unknown", "message": error_output,
                "raw_message": error_output}

    error_lines = lines[first_error_index:]
    return {"error_type": "compilation_error",
            "message": "Maven build failed",
            "raw_message": "\n".join(error_lines)}
```

3.6 Supporting Modules (Dataset Drivers)

The following two scripts *operationalize* the experiments at scale. They are intentionally I/O bound, stream the dataset, and keep memory footprint modest. The excerpts below are *abridged for readability* (logging, argument validation, and edge-case handling trimmed); the actual repository contains additional checks (timeouts, retry backoffs, and resume).

3.6.1 Dataset Processing and Filtering

`scripts/write_matching_projects.py`. Streams a large semicolon-separated CSV and selects rows where the UUT basenames match. This improves precision by preferring fork pairs that kept class names stable.

Listing 3.4: Core filtering loop in `write_matching_projects.py` (abridged)

```
import argparse, csv, os, sys

def basename_no_ext(path: str) -> str:
    base = os.path.basename(path)
    return base[:-5] if base.endswith(".java") else base

def row_matches(row: dict) -> bool:
    src = basename_no_ext(row["sourceUUT"])
    tgt = basename_no_ext(row["targetUUT"])
    return src.lower() == tgt.lower()

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--in", dest="input_csv", required=True)
    ap.add_argument("--out", dest="output_csv", required=True)
    ap.add_argument("--limit", type=int, default=0,
                    help="max rows to write; 0 = all")
```

```
    args = ap.parse_args()

    n_written = 0
    with open(args.input_csv, newline="", encoding="utf-8") as fin, \
         open(args.output_csv, "w", newline="", encoding="utf-8") as fout:

        rdr = csv.DictReader(fin, delimiter=";")
        wtr = csv.DictWriter(fout, fieldnames=rdr.fieldnames, delimiter=";")
        wtr.writeheader()

        for row in rdr:
            if row_matches(row):
                wtr.writerow(row)
                n_written += 1
                if args.limit and n_written >= args.limit:
                    break

    print(f"[match] wrote {n_written} rows to {args.output_csv}")

if __name__ == "__main__":
    main()
```

Design notes. (i) *Case-insensitive* comparison tolerates naming style drift; (ii) deliberately avoids normalizing package paths to not assume common roots; (iii) accepts an explicit `-limit` to generate a quick pilot subset for prompt calibration.

scripts/process_dataset.py. Runs the end-to-end adaptation over the filtered CSV. It clones target repositories, verifies pre-build, fetches the source test, and calls the orchestrator. Rows produce one metrics file per run.

Listing 3.5: Dataset driver (compact)

```
import os, sys, subprocess, pandas as pd
from pathlib import Path

project_root = Path(__file__).resolve().parents[1]
sys.path.insert(0, str(project_root))

from src.main import main as run_adaptation_workflow
from src.utils import get_code_from_github

def clone_repo(project_name: str, projects_base_dir: str) -> Path:
    dest = Path(projects_base_dir) / project_name.replace("/", "_")
    if not dest.exists():
        subprocess.run(
            ["git", "clone", f"https://github.com/{project_name}.git", str(
                dest)],
            check=True
        )
    return dest

def format_file_path(file_path: str, project_name: str) -> str:
    p = file_path.replace("\\", "/")
    proj = project_name.replace("\\", "/")
    i = p.lower().rfind(proj.lower())
    if i != -1:
        p = p[i + len(proj):].lstrip("/")
    return p.split("src/", 1)[-1] if "src/" in p else p

def extract_info_from_row(row) -> dict:
    return {
        "source_project": row["sourceUUTProject"],
        "target_project": row["targetUUTProject"],
        "source_test_path": format_file_path(row["testCaseSourceFilePath"],
                                             row["sourceUUTProject"]),
        "target_uut_path": format_file_path(row["targetUUTFilePath"],
                                             row["targetUUTProject"]),
    }

def process_dataset(csv_path: str, projects_dir: str, n: int = 100,
    attempts: int = 4):
    df = pd.read_csv(csv_path, sep=";")
    for _, row in df.head(n).iterrows():
        info = extract_info_from_row(row)
        repo = clone_repo(info["target_project"], projects_dir)
        code = get_code_from_github(info["source_project"], info["
```

```
            source_test_path"])
        if not code:
            continue
        run_adaptation_workflow(
            original_test_case_code=code,
            source_test_origin_path=info["source_test_path"],
            target_project_path=str(repo),
            target_class_relative_path=info["target_uut_path"],
            max_attempts=attempts,
            source_project_name=info["source_project"],
            target_project_name=info["target_project"],
            cleanup_on_failure=True,
        )

if __name__ == "__main__":
    process_dataset(
        csv_path=str(project_root / "data/
            testcaseTargetUUTPairMatchingSourceCompile3.csv"),
        projects_dir=str(project_root / "data/projects"),
        n=100
    )
```

Design notes. The driver uses clones inside a temporary directory, so each row is isolated and leaves no residue (except metrics). A pre-build gate prevents spending LLM tokens on projects that fail even without the inserted test. The `run_adaptation` wrapper simply prepares arguments and delegates to the orchestrator shown in Algorithms 1–2.

3.7 Operational Concerns

3.7.1 Common Failure Modes and Mitigations

- **Imports drift:** Missing static imports or JUnit 4/5 API drift is common; prompts include a "*prefer minimal import edits*" instruction, and the extractor rejects multi-file responses.
- **Signature divergence:** Renamed methods/parameters are addressed by providing the exact failing line and the UUT method signatures in the prompt.
- **Build-file over-editing:** POM patches are limited to a single plugin stanza or `maven.compiler.source/target` pair; anything larger is discarded.

3.8 Summary

The implementation cleanly separates: (1) adaptation core (environment, errors, LLM, metrics), (2) dataset drivers (bulk processing, row filtering), and (3) utilities (paths, GitHub fetch, code extraction). The algorithms abstract the behavior independent of libraries, while the listings demonstrate the concrete realization used in the Maven-based experiments. The engineering choices: fail-fast pre-build checks, conservative extractors, and isolated per-row workspaces were essential to keep runs analyzable, reproducible, and cost-aware.

4 Evaluation

This chapter evaluates the prototype described in Chapters 2–3 on a set of Java fork pairs. I organize the analysis around the research questions (RQs) and report empirical results collected by the tool's metrics logger.

4.1 Research Questions

I structure the evaluation around four research questions that operationalize the goal of LLM-assisted adaptation. The measures align with the methodology and analysis plan defined in Chapter 2.

RQ1 — Diagnosis & first fixes. *Can an LLM-based diagnosis–and–repair loop fix a failed direct insertion?* **Rationale:** After a naïve copy–paste of a source test into the target, many failures are due to missing symbols, signature mismatches, or packaging issues. **Operationalization:** I trigger the LLM loop only when attempt 1 fails. Success for RQ1 is a compilable test within at most N=3 attempts. I report: number of LLM-triggered tasks, success rate among them, success attempt index, and wall-clock time (all from the metrics log). This matches the "alignment with RQs" and attempt-indexed metrics specified in the methodology.

RQ2 — Basic refactorings. *To what extent can the approach perform simple refactorings (renames, parameter changes, imports)?* **Rationale:** Near-miss divergences between forks frequently require straightforward edits rather than deep semantic changes. **Operationalization:** For tasks that invoke the LLM, I record a clone-type label and the dominant diagnostic category. I evaluate RQ2 on cases consistent with near-miss adaptation (e.g., Type-3 labels or errors pointing to renamed classes/methods or changed parameters) and report success rates, attempt indices, and representative edit patterns.

RQ3 — Complex/semantic adaptations. *Can the approach handle non-local or semantic edits (akin to Type-4 relations)?* **Rationale:** Some adaptations require restructuring beyond simple renames. **Operationalization:** I consider cases labeled as Type-4 (or exhibiting non-local edits) and measure compilable success within N=3 attempts, with attempt index and time. Because compilability may not imply

intent preservation, I complement the primary metric with spot soundness checks on a small sample and report the results in Sec. 4.3.1.

RQ4 — Baseline comparison. *How does the automated approach compare to a direct-copy baseline?* **Rationale:** The intrinsic baseline is the outcome of attempt 1 without any LLM involvement. **Operationalization:** I count attempt 1 successes as the baseline and attribute any additional successes at attempts 2–3 to the LLM loop. I report absolute and relative gains in success rate and summarize time costs by attempt. This follows the baseline and analysis plan defined in the methodology.

Together, RQ1–RQ3 assess capability across increasing adaptation difficulty, while RQ4 quantifies added value over direct reuse.

Operational clone-type definitions (used for RQ2/RQ3)

Following Roy and Cordy [11], I use clone types as an *operational* lens rather than gold-standard labels: they are assigned by the LLM from compiler diagnostics and code snippets to guide analysis.

Type-2 (near-copy with lexeme changes). Same structure/control flow as the source test; differences confined to identifiers/literals/formatting. In this pipeline, a Type-2 adaptation is expected to be achievable with *test-local* edits only (renames, imports, minor signature adjustments). The tool *does not* edit production code.

Type-3 (near-miss). Statement-level edits (add/remove/reorder) are needed but the test still targets the same feature/UUT. Allowed adaptations include small restructurings inside the test, mock rewiring, and assertion rewrites, again confined to the test file (plus minimal build-file tweaks).

Type-4 (semantic). Same intent realized through a different API/design; the test must be rewritten to call different methods/classes or a thin shim. In this study, "non-local" means conceptually non-local (different API surface) while edits remain physically local to the test file.

4.2 Experimental Setup

4.2.1 Subjects and Procedure

I evaluated on **26** source–target fork pairs drawn from public Java **Maven** projects (examples include `remoting`, `jeromq`, `JSqlParser`, `storm-crawler`). The prototype contains Gradle code paths, but Gradle support is *unfinished* and was disabled for this run. For each pair the tool (i) verified the target project builds in isolation, (ii) inserted the source test into `src/test/java/...` of the target, and (iii) executed up to N=3 adaptation attempts. On each failed attempt, compiler diagnostics and relevant code were fed to Google Gemini to propose a patched test; the build was re-tried until success or attempts were exhausted. All outcomes and timings were recorded to `adaptation_metrics.json`.

Selection of Fork Pairs

The 26 source–target fork pairs were derived in two stages designed for *reproducibility*, *control of confounders*, and *computational feasibility*.

Stage 1: Candidate subsampling from a published corpus. I started from the public CSV of matched test–UUT pairs reported by Mukelabai et al. (ASE 2023) [9], namely `data/testcaseTargetUUTPairMatching.csv`. From this large corpus, I produced a *deterministic* random subsample of 1,000 candidate pairs using:

```
python scripts/write_matching_projects.py \
  data/testcaseTargetUUTPairMatching.csv \
  data/testcaseTargetUUTPairMatchingSource1000.csv \
  1000 --seed 30
```

The parameter `1000` bounds the candidate set size to a tractable workload while maintaining diversity; `-seed 30` fixes the random stream so the same 1,000 pairs can be reproduced exactly.

Stage 2: Eligibility by pre-build (Maven-only). Given that Gradle support in my prototype is unfinished, I restricted the evaluation to Maven projects to avoid build-system variance. I then filtered the subsample to targets that *successfully built in isolation* before any test insertion. This pre-build gate removes projects that are broken, misconfigured, or environment-incompatible, so later failures can be attributed to the *adaptation* itself rather than latent build issues. The filter was executed with:

```
# dataset_file:  data/testcaseTargetUUTPairMatchingSource1000.csv
# compile_csv:   outputs/prebuild_compiles.csv
# projects_dir:  workspace/projects
filter_projects_by_prebuild(dataset_file, compile_csv, projects_dir)
```

After applying this gate (and the Maven-only restriction), **26** source–target pairs remained and formed the evaluation cohort.

Justification. *Internal validity:* The pre-build requirement ensures that the outcome "success = compilable after adaptation" is not confounded by unrelated build breakage. *Reproducibility:* Both the subsampling and the eligibility filter are scripted and parameterized (file paths, sample size, seed, and workspace), so the exact cohort can be regenerated. *Feasibility:* The 1,000-pair subsample bounds cloning/compilation time, while the pre-build gate narrows to a realistic set of adaptation candidates (n=26) under the Maven-only constraint used in this study.

4.2.2 Environment

As implemented, the tool auto-detects Maven/Gradle and enforces a fixed JDK for test compilation; in this study I restricted execution to Maven builds only. Consequently, no Gradle builds were executed, and no automated Gradle-file repairs were evaluated. All Maven targets in this run passed the *pre-build* check, so no automated `pom.xml` fixes were required. This aligns with the workflow introduced in Chapter 2.

4.2.3 Data Collected

For each pair I logged: success/failure, number of attempts, whether an LLM was used, clone classification (Type-2/3/4) when LLM analysis was triggered, execution time, the length of the LLM's response (as a coarse proxy for cost), and manual soundness verdicts (Sound/Probably-sound/Unsound) for a stratified subset (4.3.1) [11].

4.3 Overall Outcomes

Table 4.1 summarizes key results. Values are computed directly from the metrics file.

Two immediate observations follow. First, naïve copy–paste yielded 11/26 successes; adding the LLM-based loop raised the total to 18/26, an **absolute** gain of 26.9 percentage points (from 42.3% to 69.2%)—a **relative** improvement of 63.6% over

Table 4.1: Overall outcomes across 26 adaptation tasks.

Metric	Value
Total adaptations	26
Successful adaptations	18
Overall success rate	69.2%
Directly applicable (no LLM, attempt=1)	11 (42.3%)
LLM used (initial copy failed)	15
Successes among LLM cases	7 (46.7%)
Success by attempt (1/2/3)	11 / 5 / 2
Avg. execution time (all)	136.25s (median 61.36s)
Median time by attempt (1/2/3)	6.58s / 72.42s / 163.81s

the direct baseline. Second, all 11 non-LLM tasks succeeded on attempt 1 (by design, the LLM loop only runs after a failure), whereas the 15 LLM-driven tasks achieved 5 successes on attempt 2 and 2 on attempt 3; the remaining 8 consumed all 3 attempts without success. These counts, attempts, and timings come from the run-time metrics.

For completeness, a 95% Wilson interval on the overall success rate is [50.0%, 83.5%], reflecting the modest sample size (26 pairs). For the 15 LLM-triggered cases, the 95% interval is [24.8%, 69.9%]. (Intervals are computed over the metrics file's outcomes.)

4.3.1 Soundness Spot-Check Results

I reviewed **7** LLM-affected successes. Of these, **3/7** were *Sound* (42.9%; 95% CI: 15.8–75.0), **2/7** were *Probably-sound*, and **2/7** were *Unsound*. Collapsing Sound+Probably-sound yields **5/7** (71.4%; 95% CI: 35.9–91.8), indicating that a majority of LLM-enabled compile successes likely preserved intent, though not all did.

Project	**Verdict**	**One-line rationale**
IronMQ	Sound	Same CRUD/queue lifecycle scenarios and error handling; only framework/package adjustments.
shiroredis	Sound	Same scenarios (update/delete/activeSessions), same focal DAO calls and oracles; API diffs handled via mocks.
JSQLParser	Probably-sound	Core parsing inputs/oracles identical; advanced features absent in target removed, core intent preserved.
lightcouch	Sound	Same view/query APIs and oracles; Hamcrest $\leftrightarrow$ JUnit assertion syntax only; JSON encoding change keeps meaning.
geometry-api	Unsound	Assertion semantics diverge (`equals` vs. `Equals`); one conversion test path had no effective oracle.
javaewah	Probably-sound	Identical scenarios/oracles; bitmaps materialized differently (mapped vs. serialized) but tested properties align.
obd-java-api	Unsound	Oracle changed across exception types and classes under test (`UnsupportedCommandException` vs. `MisunderstoodCommandException`).

Table 4.2: Manual soundness spot-check on LLM-affected successes. Detailed worksheets appear in Appendix .1.

4.3.2 Sensitivity analysis: excluding mcMMO

I ran the tool on the same fork pairs but omitted the mcMMO pair due to its extreme runtime (1,358.6s) and unresolved dependency issues. Across 25 tasks, I observed 18 successes (72.0%, 95% Wilson [52.4%, 85.7%]). The directly applicable tests remained 11/25 (44%) and the LLM was activated 14 times with 7 successes (50.0%, 95% [26.8%, 73.2%]). Successes by attempt were shifted to 11/3/4 (attempts 1/2/3). The average runtime dropped to 91.43 s (min 3.56 s, max 419.66 s), consistent with removing the mcMMO outlier.

Table 4.3: Rerun outcomes across 25 tasks (mcMMO excluded).

Metric	Value
Total adaptations	25
Successful adaptations	18 (72.0%)
Directly applicable (no LLM, attempt=1)	11 (44.0%)
LLM used (initial copy failed)	14
Successes among LLM cases	7 (50.0%)
Success by attempt (1/2/3)	11 / 3 / 4
Avg. execution time (all)	91.43s
Min / Max time	3.56s / 419.66s

4.4 Results by Research Question

RQ1 — Can the LLM diagnose build failures and suggest useful initial fixes?

In 15/26 tasks the initial copy–paste broke the build, triggering the LLM loop. Among these, **7/15 (46.7%)** were repaired to a compilable state within at most three attempts (median success attempt=2). Unsuccessful cases (8/15) typically exhausted all three attempts. The LLM loop is substantially costlier in wall time: LLM-involved tasks had a median of 142.36s vs. 6.58s for the direct successes. These results indicate that the LLM can often localize and fix straightforward incompatibilities after a failed insertion, but not reliably so for all divergence types. All numbers in this paragraph are taken from `adaptation_metrics.json`.

RQ2 — Can the LLM perform basic refactorings (renames, simple parameter changes)?

Under the operational scope above (test-local edits only), the tool recorded instrument-derived clone-type labels whenever the LLM was invoked. For **Type-3** cases, near-

miss edits inside the test, the tool succeeded in **5/8 (62.5%)** tasks, typically via renames, import updates, assertion rewrites, or small restructurings (e.g., `JSqlParser`, `LightCouch`, `iron_mq_java`). By contrast, **Type-2** cases in this cohort were **0/5**: although canonically "easy," most of these required renames or moves in *production* types or method signatures (diagnostics: `cannot find symbol`, `incompatible types`), which the methodology deliberately disallows. Thus, within our constraints, "basic refactorings" are solvable when changes are *test-local*, but not when the apparent Type-2 divergence resides outside the test file.

This capability directly addresses a limitation observed in prior non-LLM tooling focused on test *recommendation* rather than *adaptation*: for example, Kramer reports that when class or method names diverge across forks, recommendations were not issued (e.g., "Class was refactored with different naming" and "Function was refactored with different naming" in the test suite on pp. 24–25), which prevents propagation without manual work [5]. Our LLM-based approach, evaluated on adaptation rather than recommendation, overcame several such rename scenarios.

RQ3 — Can the LLM handle more complex adaptations (Type-4/semantic)?

I observed **2** Type-4 cases; both compiled within three attempts (**2/2**). In each, the LLM redirected calls to an alternative API available in the target (imports/-mocks/shims) without modifying production code, consistent with the operational definition. The soundness spot-check (Tab. 4.2) found `shiro-redis` *Sound* and `obd-java-api` *Unsound* due to a changed exception taxonomy. This underscores that for Type-4, *compilability* indicates a successful redirection, whereas *soundness* must be verified separately (Sec. 4.3.1).

RQ4 — How does the automated approach compare to a baseline?

The evaluation plan in the proposal envisioned a direct comparison to the ASE'23 manual adaptation baseline; I did not re-run or reproduce that baseline here. Instead, I use a pragmatic baseline intrinsic to our workflow: *direct applicability* (copy–paste without any LLM intervention). Against this baseline, the LLM loop added **7** further successes on top of the **11** direct ones, improving the success rate from **42.3% to 69.2%** across the same 26 tasks (+26.9 percentage points / +63.6% relative). This quantifies the additional value of automated adaptation beyond naïve reuse.

For context, work by Kramer targeted the *applicability* problem in the IDE and reported that exact or near-exact matches were suggested while refactoring-induced

divergences were not (Section 5.2 and the limitation on refactorings in Section 6.3), which is orthogonal but complementary to our evaluation focus on *post-selection adaptation* [5].

4.5 Breakdown by Clone Type and Cost

Clone-type success. The classification distribution among LLM-triggered tasks was: Type-2 (5), Type-3 (8), Type-4 (2). Success rates were **0/5** for Type-2, **5/8** for Type-3, and **2/2** for Type-4. (Note: "Unclassified" corresponds to direct successes where the LLM was not invoked.)

Why did Type-2 underperform (0/5) here? Although Type-2 is canonically "easy," the five Type-2–labeled tasks in this run predominantly surfaced `cannot find symbol` / `incompatible types` where the required change was a rename or move in *production* classes or method signatures. Because the pipeline intentionally restricts edits to the test file (and at most minimal `pom.xml` tweaks) to protect internal validity, such cross-file refactorings are out of scope—no sequence of test-local edits can recover these builds. In two cases, the label itself was likely optimistic: the LLM inferred Type-2 from superficial similarity, but later diagnostics indicated package moves and method rebindings (closer to Type-3/4). This explains the apparent paradox of Type-2 $<$ Type-3/4 in our results: the tool is effective for basic refactorings *inside the test*, yet it cannot perform the cross-file changes that several Type-2 cases actually demanded. A natural extension is to add a guarded, project-wide rename/-move step derived from compiler errors, or to permit narrowly scoped production edits under a separate attempt budget (cf. Outlook).

Time and "cost". Median wall time grew with the number of adaptation attempts (6.58s at attempt 1, 72.42s at attempt 2, 163.81s at attempt 3). Among LLM cases, the median time was 142.36s; failed LLM runs were slower (median 187.58s) than successful ones. As a coarse proxy for cost, LLM responses were longer on failures (mean length $\approx$ 14,694) than on successes ($\approx$ 7,700), suggesting that more verbose outputs did not translate into better fixes in these runs. (Timings and response lengths are taken verbatim from the metrics file; lengths are the recorded response sizes.) In addition, I observed an extreme outlier: the `mcMMO` fork pair ran for **1,358.6s** and still failed due to extensive architectural differences and missing dependencies, indicating that severe structural divergence can dominate runtime without yielding progress.

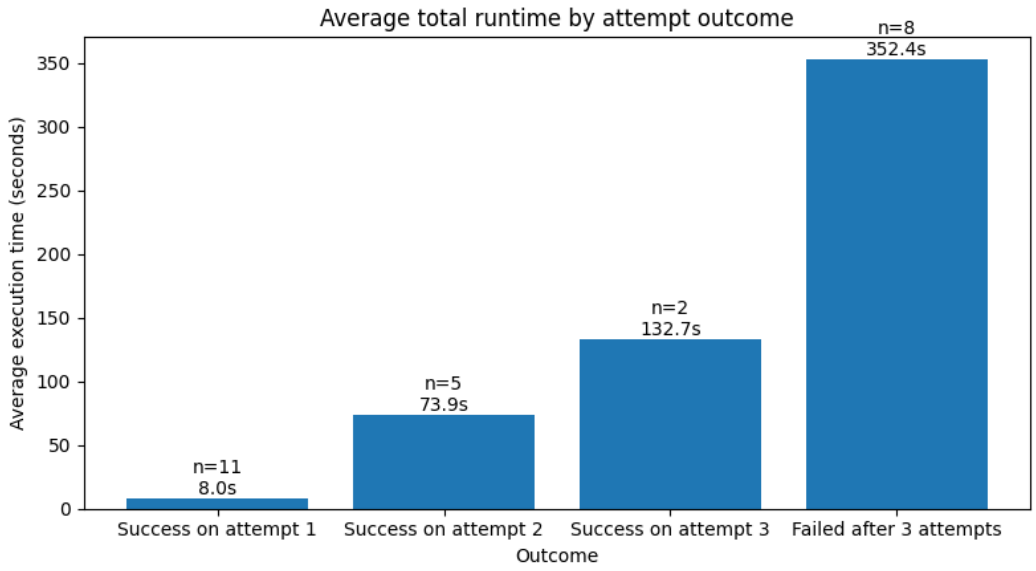


Figure 4.1: Attempt-wise wall-clock time distribution.

4.6 Qualitative Cases (Illustrative)

To complement the quantitative results, I present short vignettes that show the concrete edits the tool applied (or failed to apply). Each vignette reports: *context*, *failure signal*, *edit*, *outcome*, and *soundness*.

Case A — Type-3, compilable but *Unsound*: geometry-api **Context.** Source and target both provide WKB round-trip support; API drift caused signature/import differences. **Failure.** Initial insertion failed with missing exceptions and a method-identity mismatch in an assertion. **Edit.** The LLM added missing imports, widened the throws clause, and replaced `Equals` with Java's `equals`. **Outcome.** Compiled on attempt 3. In the spot-check this was *Unsound*: the changed oracle weakened the equality check for one path.

Listing 4.1: geometry-api: focused diff for `TestWKBSupport.java`

```
diff ../../testcases/davidraleigh_geometry-api-java_TestWKBSupport.java
    src/test/java/com/esri/core/geometry/TestWKBSupport.java
@@
+ import org.codehaus.jackson.JsonParseException; // added
@@
- public void testWKB() {
+ public void testWKB() throws JsonParseException, IOException { //
    widened throws
@@
```

```
- assertTrue(geometry.Equals(geomFromBinary));
+ // OGCGeometry exposes 'equals', not 'Equals'
+ assertTrue(geometry.equals(geomFromBinary));
\ No newline at end of file
```

Case B — Direct success (no LLM): remoting **Context.** Same project family; the copied test remained applicable. **Failure.** None; built in ≈ 9–10s. **Edit.** No changes. **Outcome/Soundness.** *Sound* by inspection (identical scenario/oracle).

Case C — Type-3, *Sound*: LightCouch **Context.** Assertion style and JSON utility drifted (Hamcrest → JUnit assertions; JSON codec update). **Failure.** `cannot find symbol` on `assertThat(..., is(...))` and outdated JSON helper. **Edit.** Assertion rewrites and import updates; swapped JSON helper to target's API. **Outcome/Soundness.** Success on attempt 2; *Sound* (same inputs/oracles).

Listing 4.2: LightCouch: assertion+API drift handled

```
diff ../../testcases/jjrodrig_LightCouch_ViewsTest.java src/test/java/org
    /lightcouch/tests/ViewsTest.java
@@
-package org.lightcouch.tests;
-
-// import static org.hamcrest.CoreMatchers.is;
-// import static org.hamcrest.CoreMatchers.not;
-// import static org.hamcrest.MatcherAssert.assertThat;
+package org.lightcouch.tests;
+
+import static org.junit.Assert.assertEquals;
+import static org.junit.Assert.assertFalse;
+import static org.junit.Assert.assertTrue;

@@
-public class ViewsTest extends CouchDbTestBase {
+// Removed 'extends CouchDbTestBase'
+public class ViewsTest {

+ private static CouchDbClient dbClient;
+ private static com.google.gson.Gson gson;

@@
- @BeforeClass
- public static void setUpClass() {
- dbClient = new CouchDbClient();
- dbClient.syncDesignDocsWithDb();
- init();
```

```
- }
+ @BeforeClass
+ public static void setUpClass() {
+ dbClient = new CouchDbClient();
+ dbClient.syncDesignDocsWithDb();
+ gson = dbClient.getGson(); // obtain Gson for JSON keys
+ init();
+ }
+
+ @org.junit.AfterClass
+ public static void tearDownClass() {
+ if (dbClient != null) dbClient.shutdown();
+ }

@@ // assertion style changes (Hamcrest -> JUnit)
-assertThat(foos.size(), is(1));
+assertEquals(1, foos.size());

-assertThat(allDocs.size(), not(0));
+assertTrue(allDocs.size() != 0);

@@ // complex keys: List<int[]> -> List<String> (JSON)
-List<int[]> keysToGet = new Vector<int[]>();
-keysToGet.add(new int[] { 2011, 10, 15 });
-keysToGet.add(new int[] { 2013, 12, 17 });
-ViewResult<Integer[], Integer, Foo> fooRows = dbClient.view("example/
    by_date")
- .keys(keysToGet)
+java.util.List<String> jsonKeysToGet = new java.util.Vector<>();
+jsonKeysToGet.add(gson.toJson(new int[] { 2011, 10, 15 }));
+jsonKeysToGet.add(gson.toJson(new int[] { 2013, 12, 17 }));
+ViewResult<Integer[], Integer, Foo> fooRows = dbClient.view("example/
    by_date")
+ .keys(jsonKeysToGet)
      .group(true)
      .queryView(Integer[].class, Integer.class, Foo.class);
-assertThat(fooRows.getRows().size(), is(2));
+assertEquals(2, fooRows.getRows().size());
```

Case D — Type-4, *Sound*: shiro-redis **Context.** DAO/storage layer refactored; focal calls moved and were renamed. **Failure.** Missing method and moved type. **Edit.** Updated import, replaced `delete()` with `remove()`, adjusted mock wiring. **Outcome/Soundness.** Success on attempt 2; *Sound* (*same scenario and oracle*, different API surface).

Listing 4.3: shiro-redis: moved types + JUnit 5→4 + expire source change

```
diff ../../testcases/yiyingcanfeng_shiro-redis_RedisSessionDAOTest.java
    src/test/java/org/crazycake/shiro/RedisSessionDAOTest.java
@@
-import org.crazycake.shiro.exception.SerializationException;
-import org.crazycake.shiro.serializer.ObjectSerializer;
-import org.crazycake.shiro.serializer.StringSerializer;
-import org.junit.jupiter.api.BeforeEach;
-import org.junit.jupiter.api.Test;
+// package layout changed in target:
+import org.crazycake.shiro.SerializationException;
+import org.crazycake.shiro.ObjectSerializer;
+import org.crazycake.shiro.StringSerializer;
+// use JUnit 4 in target:
+import org.junit.Before;
+import org.junit.Test;
@@
- private IRedisManager redisManager;
+ // target uses concrete class instead of interface
+ private RedisManager redisManager;
@@
- @BeforeEach
+ @Before
@@
- redisManager = mock(IRedisManager.class);
+ redisManager = mock(RedisManager.class);
@@
- redisSessionDAO.setExpire(expire);
+ // target reads expire from redisManager; mock it
+ when(redisManager.getExpire()).thenReturn(expire);
+ } else {
+ // preserve default-case oracle from original test
+ when(redisManager.getExpire()).thenReturn(2);
+ }
```

Case E — Type-4, compilable but *Unsound*: obd-java-api **Context.** Exception taxonomy changed across forks. **Failure.** Compilation succeeded after adapting the expected exception, but semantics diverged. **Edit.** Changed the expected exception class. **Outcome/Soundness.** Attempt 3 yielded a compilable test marked *Unsound* in the spot-check.

Listing 4.4: obd-java-api: TestNG→JUnit4, command shim, and exception drift

```
diff ../../testcases/pires_obd-java-api_UnsupportedCommandExceptionTest.
    java src/test/java/com/github/pires/obd/exceptions/
    UnsupportedCommandExceptionTest.java
@@
-import com.github.pires.obd.commands.SpeedCommand;
-import org.testng.annotations.BeforeMethod;
```

```
-import org.testng.annotations.Test;
+import org.junit.Before;
+import org.junit.Test;
+import pt.lighthouselabs.obd.commands.ObdCommand; // use generic command
    in target
+import pt.lighthouselabs.obd.exceptions.MisunderstoodCommandException;
@@
- private SpeedCommand command;
+ private ObdCommand command; // API change: use ObdCommand
@@
- @BeforeMethod
+ @Before
@@
- command = new SpeedCommand();
+ // Anonymous ObdCommand: exception occurs before calculations
+ command = new ObdCommand("0100") {
+ @Override protected void performCalculations() {}
+ @Override public String getFormattedResult() { return "Dummy Result"; }
+ @Override public String getName() { return "Dummy Test Command"; }
+ };
@@
- @Test(expectedExceptions = UnsupportedCommandException.class)
+ @Test(expected = MisunderstoodCommandException.class)
@@
- @Test(expectedExceptions = UnsupportedCommandException.class)
+ @Test(expected = MisunderstoodCommandException.class)
```

Case F — Failure / runtime outlier: mcMMO Context. Extensive architectural divergence and unresolved deps. **Failure.** Repeated symbol-resolution and dependency errors; no convergent edit sequence. **Edit.** Multiple suggested patches; none addressed root cause. **Outcome.** All 3 attempts failed; runtime = 1358.6 s. Demonstrates limits when surface similarity hides incompatible contexts.

4.7 Threats to Validity

Validity of Measurement Instruments I distinguish instrument validity from method validity. For clone-type labels produced by the LLM, face validity is reasonable but subjective; content validity is incomplete because I did not enforce a checklist for the accepted taxonomy; criterion validity is limited since I did not compare against an external detector; and construct validity is threatened by the abstract nature of "clone type," so per-type results should be interpreted cautiously. For the main outcome, "success = compilable," face validity is acceptable but the measure does not cover test intent or runtime behavior, and I only performed limited spot checks; thus

it is an incomplete proxy. Wall-clock time faithfully measures runtime cost, whereas response length is only a coarse API-cost proxy.

Construct Validity (Research Method) My operationalization of success as "compilable" addresses build correctness but does not ensure intent preservation. This threatens construct validity because some compilable adaptations may be unsound. I mitigated this by reporting the limitation explicitly and by performing spot checks.

Threats related to the soundness assessment. Judgments are manual and time-bounded; I mitigated bias by using a fixed checklist (scenario, focal calls, oracle) and recording one-line justifications with each verdict. The sample is small (n=7), yielding wide intervals. When APIs differed, I compared relations (e.g., round-trip parsing, exception class families) rather than exact strings. Tests without explicit oracles were flagged as vacuous, which can under- or over-estimate soundness depending on indirect effects.

Internal Validity I controlled alternative explanations through fixed environments (JDK, Maven), a single LLM configuration, capped attempts, and recorded parameters, which strengthens causal interpretation of differences between direct-copy and LLM-guided attempts. Residual threats include selection effects (tasks requiring LLM may be harder), stochastic LLM variability, and instrumentation effects from prompt engineering; I report attempt-indexed outcomes and fix seeds where possible to make such influences visible.

External Validity All experiments used Java/Maven projects and one LLM; Gradle support exists in code but was disabled and not evaluated. Consequently, generalization to Gradle, other build systems or languages, and other LLMs is limited. This reflects an explicit trade-off favoring tighter control (internal validity) over breadth of generalization.

Statistical Conclusion Validity Given modest samples, I use 95% Wilson intervals for proportions and focus on descriptive analysis without null-hypothesis significance tests. This choice is appropriate for the data but yields wider intervals; larger samples would narrow the uncertainty.

Reliability and Reproducibility LLM outputs are stochastic, and a rerun recovered one prior failure, indicating test–retest variability.

4.8 Summary of Findings

- **Feasibility.** The automated pipeline achieved **69.2%** compilable adaptations across 26 tasks; the LLM loop contributed **7** additional successes beyond a direct-copy baseline (+26.9 pp).
- **Where it helps.** The LLM repaired nearly half of the initially failing insertions (**46.7%**); it was particularly effective on **Type-3** divergences (**62.5%** success) and solved two Type-4 cases in the main run.
- **Where it struggles.** I observed **0/5** success on **Type-2**-labeled cases in this run; the `mcMMO` outlier (**1,358.6s**) shows that severe structural divergence can consume substantial time without convergence.
- **Cost.** LLM-guided attempts are slower (median ≈142s) than direct successes (6.6s), and longer LLM outputs were not predictive of success.
- **Soundness.** In a manual spot-check of 7 LLM-affected successes, **3/7** were *Sound* and **2/7** *Probably-sound*; **2/7** were *Unsound* (Tab. 4.2), underscoring that compilability upper-bounds intent preservation.
- **Run-to-run variance.** One previously failing pair (`gelfj` → `gelfj-alt`) succeeded on rerun (Type-4, attempt 2, 57.6s, intent preserved), suggesting that limited retries may increase yield.

Outlook. The results fulfill the core objective of demonstrating the *feasibility* of LLM-assisted test-case adaptation in fork ecosystems (Chapter 1). Two immediate next steps emerge from the data: (i) integrate test execution and lightweight oracles to assess *soundness* (beyond compilability), as envisioned in the proposal, and (ii) refine prompting/repair strategies for the persistent Type-2 failures (e.g., add project-wide search for alternative focal methods, or include build-file and dependency context when symbol resolution errors point outside the UUT). In addition, (iii) incorporate controlled reruns or ensemble prompting to mitigate stochastic failures evidenced by the recovered `gelfj-alt` case.

5 Discussion

5.1 Synthesis and Implications

The study indicates that LLM guidance can *materially* improve automated test-case adaptation across forked Java projects. Qualitatively, the compiler-aware loop proved most effective on near-miss refactorings and structurally coherent edits, while some ostensibly simple Type-2 differences remained stubborn. I observed stochastic variance across reruns and sensitivity to outliers (e.g., very large or atypical projects). For the exact figures and confidence intervals, see Chapter 4.

5.1.1 Interpreting the Soundness Findings

The spot-check suggests that *structural* adaptations (e.g., JUnit 4↔5, imports, minor API shims) generally preserve test intent, while *semantic* shifts (notably changes to exception types or equality semantics) are the main failure modes. I also observed *scope reduction*: when target APIs lacked advanced features present in the source, I removed those checks but retained core behavior validations, leading to "Probably-sound" rather than "Sound." Finally, a small number of tests were weak or vacuous (e.g., conversion pipelines without substantive assertions), reinforcing that *Compilable* only upper-bounds *Sound*.

5.2 Contributions

I provide an **automated adaptation pipeline**: a complete, reusable workflow that combines pre-build validation, structured test insertion, an error-driven LLM repair loop, and persistent metrics for post-hoc analysis. The implementation is Maven-first, with Gradle stubs present but *not evaluated* in this study. I introduce compiler-aware prompting that fuses test code, UUT context, and parsed diagnostics (e.g., missing symbols, imports, signature mismatches) to enable targeted, iterative repairs rather than unconstrained generation. I contribute empirical evidence at fork scale by evaluating 26 source–target pairs and showing that LLM assistance can raise compilable adaptation rates beyond direct copy–paste, with the strongest gains on **Type-3** divergences and clear cost profiles across attempts. I release a metrics schema and drivers, a structured logger, and dataset-processing scripts to run large

batches, filter pairs, and summarize outcomes to enable reproducible analyses and follow-up studies. Finally, I offer a problem characterization that highlights where LLMs help (near-miss refactorings, renames/parameter edits) and where they struggle (some apperantly simple Type-2 cases, severe structural divergence), along with first evidence on non-determinism and outlier behavior.

5.3 Limitations

The primary limitation is the outcome metric (compilability): success was defined as "builds compile," and tests were not executed systematically. As observed in one **Type-4** case, a compilable adaptation may be *unsound*, so the reported rates upper-bound true correctness. The sample size and taxonomy are constrained; with 26 tasks (15 LLM-triggered), several confidence intervals are wide, and clone-type labels were LLM-derived approximations rather than ground-truth taxonomy. In terms of ecosystem scope, subjects were Java/Maven/Gradle only; generalization to other languages or build systems is untested, and all evaluated targets passed pre-build, so automated build-file repair was not exercised. Stochasticity and reruns also matter: LLM outputs vary across runs, and a limited rerun recovered one failure, indicating sensitivity to randomness and prompt dynamics. Cost measurement relied on wall-clock time and response length as coarse proxies; actual API cost and energy/runtime trade-offs were not modeled explicitly. The baseline coverage used an intrinsic comparison (direct copy); prior work on recommendation or manual propagation was not re-executed here, limiting external comparative claims. Finally, selection and timeouts influenced outcomes: a small number of hard cases (e.g., mcMMO) dominated runtime, and the fixed cap of $N{=}3$ attempts may under-explore difficult but solvable adaptations.

5.4 Future Work

Future work should move from compilability to soundness by integrating test execution, lightweight oracles, and differential or metamorphic checks to verify intent preservation, and by reporting dual metrics (compilability & soundness) with confidence intervals. Richer context and retrieval is promising: prompts can be augmented with structured project context (symbol indices, call graphs, dependency manifests) and extended to include build-file or dependency reasoning when errors point outside the UUT. Addressing Type-2 failure modes calls for project-wide search for renamed focal methods, import mapping, and AST-guided edits that specifically target signature and package mismatches that defeated the LLM in this run. Adaptive repair strategies could include multi-attempt curricula (coarse→fine repairs), verifier-in-the-loop cycles (compile→execute→counterexample prompting), and program-repair operators to complement LLM edits. Cost-aware orchestration

is another direction: learn stopping policies and attempt budgets from telemetry, prioritize fixes with highest expected utility per second/€, and preempt extreme outliers via early anomaly detection. Broader evaluation should consider larger, multi-language corpora (Kotlin/Gradle, Python/PyTest, JavaScript/Jest), include broken targets to evaluate build-file repair, and compare against (i) manual adaptation samples and (ii) non-LLM program-repair baselines. To improve robustness to randomness, add controlled reruns, prompt ensembles, and parameter sweeps, and report stabilized estimates (e.g., majority vote over k attempts). A clone-aware analysis using offline clone detectors or AST/PDG similarity could replace LLM heuristics to obtain ground-truth labels and enable sharper per-type conclusions. Finally, artifact packaging should be hardened (config templates, seeds, containerized builds), and an evaluation harness with scripts and a metrics schema should be released to support independent replication and extension.

6 Conclusion

This thesis investigated whether Large Language Models (LLMs) can automate the adaptation of unit tests across forked Java projects. Building on the engineering workflow introduced in Chapters 2–3, and the empirical study in Chapter 4, I developed a compiler-aware adaptation loop that inserts candidate tests, diagnoses build failures, and applies LLM-guided repairs.

Overall outcome. Among **26** adaptation tasks, the pipeline produced **18** compilable adaptations (**69.2%**). Relative to a direct copy–paste baseline (**11**/26, **42.3%**), the LLM loop added **+7** further successes (**+26.9** percentage points, **≈63.6%** relative improvement). When invoked (**15** tasks), the LLM repaired **7** cases (**46.7%**) within at most three attempts. Runtime costs rose with each additional attempt (median ≈6.6 s → 72.4 s → 163.8 s). A sensitivity check excluding the mcMMO outlier yielded **18**/25 successes (**72.0%**) with lower average runtime. Beyond improving compilability, the manual spot-check found that **3/7** LLM-assisted successes were strictly *Sound* and **5/7** were *Sound or Probably-sound*, highlighting both the promise of error-driven adaptation and the need for guards against semantic drift (especially for exception semantics and equality checks).

Answers to the research questions.

- **RQ1 (diagnosis & initial fixes):** The LLM frequently localized and repaired straightforward incompatibilities after a failed insertion, achieving **46.7%** success among LLM-triggered cases, but not uniformly so.
- **RQ2 (basic refactorings):** The approach handled near-miss refactorings (renames, signature changes) particularly well: LLM-labeled **Type-3** cases succeeded in **5**/8 (**62.5%**).
- **RQ3 (complex/semantic changes):** Two **Type-4** cases compiled (**2**/2), indicating capacity for non-local edits; however, a spot-check uncovered one *compilable but unsound* adaptation, underscoring that compilability does not guarantee intent preservation.
- **RQ4 (baseline comparison):** Against the intrinsic baseline of direct applicability, the LLM loop delivered a substantial absolute gain of **+26.9** percentage points, quantifying its added value for post-selection adaptation.

Implications. For practitioners, LLM assistance can raise the yield of test reuse across forks, especially for near-miss (Type-3) divergences, but incurs nontrivial runtime and requires downstream validation to ensure soundness. For researchers, the results encourage verifier-in-the-loop designs and cost-aware orchestration; concrete directions are discussed in Sections 5.3–5.4.

Limitations and outlook. This work measured *compilability* rather than *soundness* at scale, employed LLM-derived clone-type labels, and focused on Java/Maven with Gradle stubs not evaluated. A clear path forward is to integrate execution-based oracles, enrich prompts with project-wide context, and broaden subjects and baselines; detailed proposals appear in Sections 5.3–5.4.

Closing remark. Within these boundaries, the thesis establishes a feasible and effective path toward automated test-case adaptation in fork ecosystems: LLM guidance can meaningfully increase compilable reuse beyond naive copy–paste, while highlighting a clear roadmap to soundness-aware, cost-efficient, and more general solutions.

List of Figures

List of Tables

Bibliography

[1] I. Bouzenia and M. Pradel. "You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects". In: *arXiv preprint* (2024). arXiv: 2412.10133 [cs.SE].

[2] C. Brindescu, M. Codoban, and C. Bird. "An empirical investigation into merge conflicts and their effect on software quality". In: *Empirical Software Engineering* 25 (2020), pp. 3915–3951.

[3] J. Dong, Y. Lou, H. Fu, P. Zhou, L. Zhang, and H. Mei. "MergeGen: A Generative Model for Merge Conflict Resolution". In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023.

[4] A. Al-Kaswan, M. Ghafari, and O. Nierstrasz. "Reuse and maintenance practices among divergent forks in three software ecosystems". In: *Empirical Software Engineering* 27.54 (2022), p. 54.

[5] Luca Kramer. *IntelliJ IDEA Plugin for Test Case Recommendation for Integration in Forked Java Projects Hosted on GitHub*. Bachelor's thesis. Software Engineering Faculty; Erstprüfer: Prof. Dr. Thorsten Berger; Zweitprüfer: Jan Sollmann, M. Sc. Bochum, Germany, Sept. 2024. URL: https://github.com/isselab/TestCasePropagation.

[6] S. Larsén. "Spork: Move-enabled structured merge for Java with GumTree and 3DM". Student thesis. KTH, School of Electrical Engineering and Computer Science (EECS), 2020.

[7] A. Mastropaolo, M. Ciniselli, L. Pascarella, and G. Bavota. "An empirical study on the code refactoring capability of large language models". In: *arXiv preprint* (2024). arXiv: 2401.02320 [cs.SE].

[8] M. Mukelabai, P. Borba, and T. Berger. "Semi-automated test-case propagation in fork ecosystems". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 2021, pp. 16–20.

[9] M. Mukelabai, C. Derks, J. Krüger, and T. Berger. "To share, or not to share: Exploring test-case reusability in fork ecosystems". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering(ASE)*. 2023.

[10] S. Rahman, S. Kuhar, B. Cirisci, P. Garg, S. Wang, X. Ma, A. Deoras, and B. Ray. "UTFix: Change Aware Unit Test Repairing using LLM". In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA1 (2025), Article 85.

[11] Chanchal Kumar Roy and James R. Cordy. *A Survey on Software Clone Detection Research*. Tech. rep. 2007-541. Technical Report. School of Computing, Queen's University at Kingston, Sept. 2007. URL: https://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf.

[12] A. Shirafuji, Y. Hoshikawa, Y. Watanobe, and K. Inoue. "Refactoring programs using large language models with few-shot examples". In: *arXiv preprint* (2023). arXiv: 2311.11690 [cs.SE].

[13] A. Svyatkovskiy, Y. Zhao, C. Fu, N. Sundaresan, and S. Chandra. "MergeBERT: Program Merge Conflict Resolution via Neural Transformers". In: *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Also arXiv:2109.00084. 2021. arXiv: 2109.00084.

[14] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. "Software testing with large language models: Survey, landscape, and vision". In: *arXiv preprint* (2023). arXiv: 2307.07221 [cs.SE].

[15] J. Zhang, M. Kaufman, T. Mytkowicz, R. Piskac, and S. Lahiri. "Can Pretrained Language Models be Used to Resolve Textual and Semantic Merge Conflicts? (Experience Paper)". In: *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2021.

.1 Soundness Worksheets

Template per pair:

- **Project / Pair:** *<name>*
- **Attempt index:** *2 or 3*
- **Scenario/Inputs (src→tgt):** *<one line>*
- **Focal calls (src→tgt):** *<class.method list>*
- **Oracle (src→tgt):** *<value/exception/relation>*
- **Key adaptations:** *<JUnit version, missing classes, mocks, etc.>*
- **Verdict:** *Sound / Probably-sound / Unsound*
- **Justification:** *<one line>*

Filled examples (short):

- **IronMQ:** Sound. Same CRUD/queue lifecycle scenarios and error handling; framework/package adjustments only.
- **shiroredis:** Sound. Same scenarios and oracles; API differences mocked.
- **JSQLParser:** Probably-sound. Core parsing cases identical; removed advanced/absent features.
- **lightcouch:** Sound. Same view/query semantics; assertion syntax changes only; JSON encoding preserves meaning.
- **geometry-api:** Unsound. `equals` vs. `Equals` semantics; one vacuous conversion test.
- **javaewah:** Probably-sound. Materialization differs, tested properties align.
- **obd-java-api:** Unsound. Different exception types and class under test.