Pull Request Report

Deterministic Preprocessing Verification Mode

OpenVerifiableLLM Project | March 2026

PR Title	`feat: Deterministic Preprocessing Verification Mode`
Files Added	`verify.py, test_verify.py`
Tests	**37 / 37 passing**
Framework Ref.	Balan et al. 2025 — Gap 5.1 (Raw → Processed data verifiability)

1. Overview

This PR introduces a dedicated verification module for the OpenVerifiableLLM data pipeline. Its job is simple: given the same raw Wikipedia dump, re-run the entire preprocessing step from scratch and prove that the output produced today is byte-for-byte identical to the output recorded in the manifest at first run.

This directly addresses Gap 5.1 from the framework paper — the missing cryptographic link between raw data and training-ready data. Before this PR, the pipeline could generate hashes and Merkle roots, but had no automated way to re-validate them.

2. Problem Being Solved

After the preprocessing pipeline runs once and produces wiki_clean.txt, there was no way to answer the question: "Is this output still trustworthy?" Any of the following could silently corrupt the dataset:

- **Silent file edits:** someone modifies wiki_clean.txt after generation
- **Dump substitution:** the raw .bz2 file is swapped for a different Wikipedia snapshot
- **Non-determinism:** a code change causes the same input to produce different output
- **Environment drift:** a different Python version producing subtly different cleaned text

Without catching these issues early, any downstream training run would silently use unverified data — undermining the entire goal of a cryptographically auditable AI pipeline.

3. What is Built

Two files were added to the project:

3a. verify.py — The Core Verification Module

Contains all the logic for running a full deterministic re-verification. The main public function is:

```
verify_preprocessing(input_dump, manifest_path, project_root)
```

When called, it works through 6 ordered stages. It returns a VerificationReport object containing a structured pass/fail result for every individual check.

3b. test_verify.py — The Test Suite

Contains 37 unit and integration tests covering every check, every failure mode, and backward compatibility with older manifests. Uses only Python's standard library unittest — no external test runner required.

4. How It Works — The 6 Verification Stages

Each stage is a gate. If a critical stage fails, the function returns immediately with a FAIL report rather than proceeding blindly.

#	Stage	What It Checks
1	**Manifest Exists**	Loads dataset_manifest.json — fails immediately if absent so no false positives occur downstream
2	**Raw File Integrity**	Checks the raw dump exists, then validates its SHA256 hash and Merkle root against the manifest before any re-processing begins
3	**Metadata Checks**	Validates dump_date, wikipedia_dump filename, and python_version — a Python mismatch is flagged as a FAIL because it can cause non-deterministic output
4	**Re-run Preprocessing**	Calls extract_text_from_xml() in a clean isolated temp directory — completely independent of the original output files, with any decompression or parse errors caught gracefully
5	**Processed File Check**	Computes SHA256 and Merkle root of the freshly reproduced wiki_clean.txt and compares both against the manifest values
6	**Manifest Fields**	Cross-checks preprocessing_version in the reproduced manifest against the original — catches cases where the pipeline code itself has changed

5. Key Design Decisions

Isolated Temp Directory

Re-processing always runs inside a fresh tempfile.mkdtemp() folder and never touches the original output files. This guarantees the verification is genuinely independent.

```
tmp_dir = Path(tempfile.mkdtemp(prefix='ovllm_verify_'))
```

Graceful Error Handling for Corrupted Files

If a tampered .bz2 file causes a decompression error during re-run, the exception is caught and recorded as a FAIL on reprocessing_succeeded rather than crashing the entire process.

```
except (OSError, EOFError, Exception) as exc:  # caught, recorded as FAIL
```

Backward Compatibility — Legacy Manifests

Old manifests that predate the Merkle root feature simply skip those checks with a SKIP status rather than failing. This means the verifier works on any existing project without requiring a re-run of preprocessing.

Structured Report Object

Every result is encapsulated in a VerificationReport with .passed, .failed, .skipped lists, .all_passed bool, .summary() for humans, and .to_dict() for machine consumption and JSON export.

6. Full Check Inventory

Below is every check the verifier runs, what it validates, and the expected status on a clean unmodified pipeline:

Check Name	Status	Description
`manifest_exists`	✓PASS	dataset_manifest.json found and loaded
`raw_file_exists`	✓PASS	Input dump file is present on disk
`raw_sha256`	✓PASS	SHA256 of raw dump matches manifest
`raw_merkle_root`	✓PASS	Merkle root of raw dump matches (SKIP if legacy manifest)
`dump_date`	✓PASS	Date parsed from filename matches manifest
`wikipedia_dump_name`	✓PASS	Filename matches what was originally recorded
`python_version`	✓PASS	Python version matches original run
`reprocessing_succeeded`	✓PASS	extract_text_from_xml() completed without error

Check Name	Status	Description
`processed_sha256`	✓PASS	SHA256 of reproduced wiki_clean.txt matches
`processed_merkle_root`	✓PASS	Merkle root of reproduced output matches (SKIP if legacy)
`manifest_preprocessing_ver`	✓PASS	preprocessing_version tag is consistent

7. Test Suite — 37 / 37 Passing

Tests are grouped into 6 test classes, each testing a distinct concern:

Test Class	Tests	What It Covers
`TestVerificationReport`	**11**	Report counts, summary verdicts, status icons, to_dict() serialisation
`TestCheckField`	**4**	Equal/unequal values, type coercion, detail field storage
`TestLoadManifest`	**3**	Missing file error, valid JSON loading, error message clarity
`TestHappyPath`	**8**	Full pass on clean unmodified pipeline — all check names, all hash comparisons, explicit manifest path
`TestFailureScenarios`	**8**	Missing manifest, missing dump, tampered file, corrupted SHA256/Merkle in manifest, wrong dump name
`TestLegacyManifest`	**2**	Manifests without Merkle fields produce SKIP not FAIL; non-Merkle checks still pass normally

8. How to Use

As a Python function

```
from openverifiablellm.verify import verify_preprocessing
report = verify_preprocessing('simplewiki-20260201.xml.bz2')
print(report.summary())          # human-readable
report.to_dict()                 # machine-readable / JSON export
```

From the command line

```
python -m openverifiablellm.verify simplewiki-20260201.xml.bz2
python -m openverifiablellm.verify dump.xml.bz2 --json report.json
```

The CLI exits with code 0 on full pass, code 1 on any failure — making it easy to plug directly into a CI pipeline or pre-training gate check.

9. Framework Alignment

This PR directly addresses the first gap identified in the paper: the missing verifiable link between raw data and training-ready data.

Framework Component	How This PR Contributes
Step 1 — Raw Data Verification	Validates SHA256 + Merkle root of the raw Wikipedia dump before re-processing begins
Step 2 — Extraction & Analysis	Re-runs the full preprocessing deterministically and cryptographically validates the output
Closure	Provides the first automated cryptographic bridge between raw dump and processed training data, closing the gap the paper identified as unresolved