Supplementary material: testing and code verification strategy

1. Scope and verification

This document describes the testing and code-verification strategy we implemented in `distill-abm`. It should be read as supplementary material to the codebase and, by extension, to the research workflow that the repository operationalizes. We do not rely on a single notion of "testing." Instead, we combine four complementary assurance layers: automated regression tests, static quality gates, pre-LLM smoke audits, and artifact-rich run contracts designed for post hoc inspection. The reason is straightforward: correctness in this project cannot be reduced to one function returning one expected value. The repository orchestrates multi-stage ABM-to-LLM workflows, resumable experiment runs, prompt/evidence materialization, statistical analyses, and publication-oriented reporting. Verification therefore needs to be distributed across the same boundaries, since checking one piece in isolation would miss how the pieces interact.

We approached code verification as a systems problem. At the lowest level, unit tests verify local transformations and contracts: prompt construction, evidence preparation, summarizer routing, metrics, Design of Experiment (DOE) helpers, ingestion parsers, adapter payloads, and repository invariants. At the middle level, integration tests verify that the benchmark pipeline composes these units correctly and writes reproducibility metadata that can be inspected later. At the highest level, end-to-end tests exercise the command-line interface, smoke orchestration, study entrypoints, and operational guardrails. Beyond conventional tests, the repository also implements dedicated smoke workflows that materialize the exact artifacts one would need to audit a run before, during, and after model execution.

The objective of this supplementary material is therefore twofold. First, we enumerate the automated and semi-automated mechanisms we implemented to detect regressions and prevent silent drift. Second, we document the code paths that exist specifically to preserve evidence: run-separated directories, latest-run pointers, structured reports, request-review CSVs, static review viewers, validation-state files, resumable manifests, and manual validation artifacts. Put simply, the testing strategy in this repository is inseparable from its reproducibility strategy.

2. Repository-Level Quality Gates

The repository defines a canonical local validation suite in `src/distill_abm/agent_validation.py`, exposed through the `quality-gate` CLI surface and profile-selection helpers in `src/distill_abm/cli_quality_gate.py`. The default validation checks are:

Check	Purpose	Implementation Surface
`pytest`	Behavioral regression testing across unit, integration, and CLI surfaces	`src/distill_abm/agent_validation.py`
`ruff check .`	Linting and import/style consistency	`src/distill_abm/agent_validation.py`
`mypy src tests`	Typed-interface consistency	`src/distill_abm/agent_validation.py`
`uv build`	Packaging/build integrity	`src/distill_abm/agent_validation.py`
`smoke-ingest-netlogo`	Pre-LLM ingest verification across configured ABMs	`src/distill_abm/agent_validation.py`

The `quality-gate` command supports three scopes. A `static` scope resolves to the quick profile with `ruff` and `mypy`; a `pre-llm` scope keeps the quick profile but delegates to the broader validation flow; and a `full` scope selects the default full validation set. These policies are tested at the CLI boundary in `tests/e2e/test_cli_quality_gate.py` and at the helper boundary in `tests/unit/cli/test_quality_gate.py`.

Why do these gates matter? Because this repository is not merely a library. It is an executable research pipeline, and its correctness depends simultaneously on code behavior, CLI ergonomics, buildability, and the integrity of ABM-specific preprocessing. If any one of these breaks silently, the experimental outputs may look plausible while being wrong.

3. Size and Shape of the Automated Test Suite

At the time of this audit, the `tests/` tree contained 87 test files. We explicitly organized the suite into layers.

Layer	Files	Collected Tests	Role
Unit	72	458	Verifies small contracts, helper logic, failure handling, and repository invariants
Integration	2	17	Verifies full in-process pipeline composition and metadata persistence
End-to-end	13	59	Verifies the Typer CLI as the public operational interface
Total	87	534	End-to-end behavioral regression surface

During this audit, `uv run pytest -q` completed successfully with `534 passed` and `48 warnings` on 2026-03-16. The warnings were concentrated in quantitative-study tests that intentionally exercise small-sample ANOVA paths.

One noteworthy pattern is that the unit suite is concentrated in the modules that matter most for research correctness. The `pipeline` subpackage alone accounts for 26 unit test files. This is not accidental: orchestration, run-state handling, reporting, and smoke infrastructure are first-class correctness concerns rather than incidental implementation details.

4. Unit-Tested Surfaces

4.1 Configuration and Policy

We treat configuration as executable input rather than passive metadata. YAML loading, defaults resolution, experiment settings, and model-policy enforcement are tested in `tests/unit/configs/test_loader.py`, `tests/unit/configs/test_runtime_defaults.py`, `tests/unit/configs/test_experiment_settings.py`, and the CLI policy tests under `tests/unit/cli/`. These tests matter because the research claims encoded by the repository — benchmark-model restrictions, reference-text selection, and so on — are expressed through configuration and policy layers before any LLM call is made. If configuration is wrong, the experiments are wrong, regardless of how well the rest of the code works.

4.2 Ingestion and NetLogo Preprocessing

ABM ingestion is verified through dedicated unit modules, notably `tests/unit/ingest/test_netlogo.py`, `tests/unit/ingest/test_netlogo_workflow.py`, `tests/unit/ingest/test_csv_ingest.py`, and `tests/unit/ingest/test_ingest_smoke.py`. They verify extraction of documentation and experiment parameters from NetLogo sources, normalization of CSV inputs, placeholder detection, and stage-granular ingest smoke behavior. This layer is particularly important because malformed parameters or documentation can propagate silently into prompts and downstream evaluations. If we do not catch them here, they may go unnoticed until they contaminate the final results.

4.3 LLM Adapter Contracts and Request Helpers

We test adapter-level correctness separately from pipeline orchestration. `tests/unit/llm/adapters/test_contract.py` verifies OpenRouter- and Mistral-facing request contracts, structured-output payload construction, runtime precision/provider metadata extraction, and pacing or failure handling. The shared helper layer in `tests/unit/pipeline/test_helpers.py` verifies prompt assembly, retry behavior, circuit-breaking, structured-output recovery, token/runtime tracing, and response normalization. In our experience, provider-specific transport details are a common source of drift in LLM-based systems — a payload field gets renamed, a default changes, or a new required parameter appears — so testing this layer in isolation has proven especially worthwhile.

4.4 Summarization

We verify summarization as a separate subsystem rather than as a side effect of the main pipeline. Model-runner behavior, batch handling, fallback/error paths, reference-text selection, and post-processing are covered in `tests/unit/summarize/test_model_runners.py`, `tests/unit/summarize/test_batch.py`, `tests/unit/summarize/test_summarizer_errors.py`, and `tests/unit/summarize/test_postprocess.py`. This matters because summary-only and full-text-only modes are experimental factors in this repository, not UI conveniences. If the summarizer silently mishandles a batch or drops a reference text, the downstream comparison between experimental conditions becomes invalid.

4.5 Evaluation and DOE Analysis

Lexical scoring and factorial analysis are verified in multiple modules: `tests/unit/eval/test_metrics.py`, `tests/unit/eval/test_metrics_full.py`, `tests/unit/eval/test_reference_scores.py`, `tests/unit/eval/test_reference_scores_batch.py`, `tests/unit/eval/test_doe.py`, and `tests/unit/eval/test_doe_full.py`. These tests verify BLEU, METEOR, ROUGE, Flesch, reference aggregation, and ANOVA-table generation under valid and invalid inputs. Their purpose is methodological: they help us guard against the risk that an apparent experimental effect is in fact an evaluation bug. In other words, if the scoring code has a subtle error, we might draw incorrect conclusions from the experiment even though the pipeline itself ran without issues.

4.6 Visualization and Statistical Evidence

We treat plots and tables as experimental evidence, so visualization is directly tested. `tests/unit/viz/test_plots.py`, `tests/unit/viz/test_viz_stats_table.py`, and `tests/unit/viz/test_viz_smoke.py` verify plot generation, statistics-table formatting, visualization-smoke stages, and fallback behavior. Related statistical-evidence logic is verified in `tests/unit/pipeline/test_statistical_evidence.py`. To understand why this matters, consider that the repository's prompt factors include `plot`, `table`, and `plot+table`. Evidence corruption would

therefore invalidate not just the software but the experiment itself — which runs counter to the goal of having reliable ablation conditions.

4.7 Pipeline and Run-State Infrastructure

The largest unit-test concentration appears in the orchestration layer. This is by design. Representative modules include:

Area	Representative Tests	Verification Focus
Smoke matrix orchestration	tests/unit/pipeline/test_smoke.py	Canonical case matrix, DOE/sweep optional steps, run-level reports
Smoke manifests and I/O	tests/unit/pipeline/test_smoke_manifests.py, tests/unit/pipeline/test_smoke_io.py	Resume-safe manifests, CSV contracts
Response bundles	tests/unit/pipeline/test_smoke_response_bundle.py	Stable row schemas, artifact-path contracts
Sampled real-inference smoke	tests/unit/pipeline/test_local_qwen_sample_smoke.py, tests/unit/pipeline/test_local_qwen_sample_response.py, tests/unit/pipeline/test_local_qwen_sample_artifacts.py	Structured-output recovery, prompt compression, resumability, review artifacts
Full-case smoke	tests/unit/pipeline/test_full_case_smoke.py, tests/unit/pipeline/test_full_case_suite_smoke.py, tests/unit/pipeline/test_full_case_matrix_run_review.py	Multi-trend case execution, validation states, suite progress, review CSVs
DOE smoke	tests/unit/pipeline/test_doe_smoke.py, tests/unit/pipeline/test_doe_smoke_models.py	Pre-LLM matrix generation, model-factor policies
Quantitative smoke and secondary studies	tests/unit/pipeline/test_quantitative_smoke.py, tests/unit/pipeline/test_exploitation_factor_study.py, tests/unit/pipeline/test_llm_same_settings_study.py	Aggregation, ANOVA summaries, pairwise and reference-family reporting
Monitoring and reporting	tests/unit/pipeline/test_local_qwen_monitor.py, tests/unit/pipeline/test_report_writers.py, tests/unit/test_run_viewer.py	TUI monitoring, JSON/Markdown report writers, static review viewer payloads

We concentrated tests in the orchestration layer because we regard the persistence of prompts, responses, logs, manifests, and review surfaces as part of correctness — not as an afterthought. Intuitively, if the

orchestration layer silently drops a manifest or writes a malformed CSV, then the experiment's auditability is compromised even if every individual function returned the right value.

4.8 Repository Guardrails

Finally, the suite includes repository-structure tests under `tests/unit/repo/`. These verify benchmark assets, archive layouts, developer assets, and legacy-surface constraints. Such tests exist because a refactor can silently remove a file or folder contract on which downstream smoke or reporting workflows depend. We have encountered cases where a renamed directory went unnoticed until a smoke workflow failed in a confusing way; these guardrails prevent that.

5. Integration Tests

Integration tests are centered on the executable benchmark pipeline rather than on isolated helpers. The main reference is `tests/integration/test_pipeline_run.py`, complemented by `tests/integration/test_pipeline_uses_abm_and_full_metrics.py`. These tests use fake adapters but real pipeline code and real filesystem writes. They verify that a run can proceed from structured inputs through context generation, trend generation, optional summarization, scoring, metadata persistence, debug-trace writing, runtime provider/precision capture, and resumable execution. They also verify that additional scoring references and summary-cleaning effects are persisted correctly.

Why is this layer essential? Because the principal scientific outputs of the repository are not individual function returns. They are compound artifacts: reports, metadata files, trace bundles, plots, statistics tables, and CSV summaries. A unit test can verify that a scoring function returns the right number, but only an integration test can verify that the number ends up in the right file with the right metadata alongside the right prompt. The integration suite tests precisely this compound behavior.

6. End-to-End CLI Tests

The repository's public operational surface is the Typer CLI defined in `src/distill_abm/cli.py`. The end-to-end tests therefore treat command invocation itself as a correctness boundary. The main modules are:

CLI Surface	Primary Test Module	What Is Verified
Core run and policy	`tests/e2e/test_cli.py`	Model-policy enforcement, ablation flags, invalid options, command wiring
Ingest smoke	`tests/e2e/test_cli_ingest_smoke.py`	Stage selection, JSON output, and ABM ingest audit entrypoints
Sampled smoke	`tests/e2e/test_cli_local_qwen_sample_smoke.py`	Sampled real-inference smoke command behavior and reporting

Full-case smoke and suite	tests/e2e/test_cli_full_case_smoke.py, tests/e2e/test_cli_full_case_matrix_smoke.py, tests/e2e/test_cli_full_case_suite_smoke.py	Multi-trend smoke orchestration, resumability, suite wiring
Summarizer smoke	tests/e2e/test_cli_summarizer_smoke.py	Summarizer-audit entrypoint behavior
Quantitative smoke and studies	tests/e2e/test_cli_quantitative_smoke.py, tests/e2e/test_cli_exploitation_factor_study.py, tests/e2e/test_cli_llm_same_settings_study.py	Post-generation quantitative workflows
Monitoring and health checks	tests/e2e/test_cli_monitor_local_qwen.py, tests/e2e/test_cli_quality_gate.py	Operational observability and validation-command defaults
Result distribution	tests/e2e/test_cli_results_bucket.py	Result-bucket synchronization surface

These tests matter because a reproducible research repository must be operable by command, not only by import. If the CLI does not work as expected, a researcher following the documented instructions will not be able to reproduce the results — even if the underlying library functions are correct.

7. Smoke Workflows as Verification Infrastructure

The repository implements multiple smoke workflows. Their purpose is not primarily to "run a small version" of the system, as one might initially assume. Their deeper purpose is to expose intermediate state in an auditable form, which makes them a major part of the verification design.

7.1 Ingest Smoke

The ingest smoke suite in `src/distill_abm/ingest/ingest_smoke.py` verifies NetLogo-derived preprocessing stages across ABMs. It writes run-separated reports (`ingest_smoke_report.json` and `.md`) and is explicitly included in the canonical validation suite. This stage is pre-LLM: it isolates failures in documentation extraction, parameter narratives, and intermediate ingest products before they contaminate prompting. One way to think of it is as a checkpoint that ensures the inputs to the LLM stage are well-formed, so that any issues observed later can be attributed to the LLM interaction rather than to upstream data problems.

7.2 Visualization Smoke

The visualization smoke suite in `src/distill_abm/viz/viz_smoke.py` materializes plot and statistics-table evidence under the same run-separated contract. This is important because `table` and `plot+table` evidence modes are central experimental factors in our design. The corresponding tests ensure that visualization failures show up as explicit stage failures rather than being buried inside later prompt-generation errors, where they would be much harder to diagnose.

7.3 DOE Smoke

The DOE smoke suite, implemented in `src/distill_abm/pipeline/doe_smoke.py` and reported through `src/distill_abm/pipeline/doe_smoke_reporting.py` , is one of the strongest verification surfaces in

the repository. It is strictly pre-LLM. It materializes the entire design matrix, request matrix, compact JSONL case indexes, request-review CSVs, exact context prompts, exact trend prompts, evidence paths, and unresolved context placeholders that would still exist prior to the first model call. The layout guide rendered by the reporting module describes a stable audit structure with:

DOE Artifact	Function
`design_matrix.csv`	Compact case-level DOE matrix
`request_matrix.csv`	Per-request plan across context and trend requests
`request_review.csv`	Reviewer-oriented prompt/evidence preview surface
`cases.jsonl` and `requests.jsonl`	Rich machine-readable indexes
`10_shared/global/` and `10_shared/<abm>/`	Shared factor and ABM-level prompt/evidence materialization
`doe_smoke_report.json` and `.md`	Human-readable and machine-readable run reports

This suite is particularly valuable from a methodological standpoint. It verifies the exact planned experimental treatments before any provider variability is introduced. As an analogy, consider the difference between checking your experimental design on paper before running it in the lab versus discovering a design flaw only after all the data has been collected. The DOE smoke is the former: it catches problems when they are still cheap to fix.

7.4 Sampled Real-Inference Smoke

The sampled smoke runner in `src/distill_abm/pipeline/local_qwen_sample_smoke.py` is a deliberately artifact-rich verification surface for real API-backed inference. It writes one self-contained folder per sampled case, including copied inputs, prompt text, request previews, hyperparameters, context and trend outputs, full traces, optional reasoning text, prompt-compression artifacts, and a run-level `request_review.csv`. It also renders a static `review.html` viewer through `src/distill_abm/run_viewer_payloads.py` and `src/distill_abm/run_viewer.py`.

This smoke path does more than detect crashes. It persists the exact prompts and outputs a reviewer would need to judge whether context generation, evidence routing, and trend narration behaved plausibly. In our experience, having these artifacts readily available has been the fastest way to diagnose unexpected LLM behavior — far more informative than logs or error messages alone.

7.5 Full-Case and Suite Smokes

The full-case matrix and all-ABM suite runners extend the same philosophy to larger experiments. `src/distill_abm/pipeline/full_case_smoke.py`, `src/distill_abm/pipeline/full_case_matrix_smoke.py`, and `src/distill_abm/pipeline/full_case_suite_smoke.py` preserve context outputs, ordered per-plot trend artifacts, validation states, review CSVs, suite-level progress, and stable current views. Suite progress is formalized in `src/distill_abm/pipeline/full_case_suite_progress.py`, while the stable per-case review CSV contract is centralized in `src/distill_abm/pipeline/full_case_review_csv.py`.

The full-case path also implements an explicit observability layer in `src/distill_abm/pipeline/full_case_run_observability.py`, which records request kind, prompt signatures, token usage, runtime provider and precision, retry settings, compression tiers, and whether requests were reused from prior runs or from a shared context cache. The point of these workflows is not

simply to run larger jobs. It is to ensure that large smoke runs remain resumable, attributable, and reviewable rather than becoming opaque batch jobs — which is easy to let happen when the number of cases grows.

7.6 Summarizer and Quantitative Smokes

The repository also exposes targeted smoke surfaces for summarizers and quantitative post-processing. The summarizer smoke path verifies the optional summarization subsystem independently of the full pipeline. The quantitative smoke path verifies the assembly of factorial inputs, ANOVA summaries, best-score tables, multi-LLM comparisons, and publication-oriented output tables. The sampled smoke also merges case-level response bundles into run-level `master_responses.csv` files and, when appropriate, into a repository-level `results/master_responses.csv` through the reporting helpers in `src/distill_abm/pipeline/smoke_reporting.py` and `src/distill_abm/pipeline/smoke_response_bundle.py`. Keeping these separate reduces diagnostic ambiguity. If we see a regression in summarizer handling or quantitative aggregation, we can localize it without conflating it with prompt generation or provider transport issues.

8. Reproducibility, Resume Semantics, and Artifact Contracts

The testing strategy is reinforced by explicit artifact contracts. These contracts are formalized in `src/distill_abm/pipeline/run_artifact_contracts.py`, which defines stable filenames such as `latest_run.txt`, `latest_report_path.txt`, `run.log.jsonl`, `review.html`, and standardized report names for ingest, visualization, DOE, and sampled smoke runs. The same module also defines an active-run lock to prevent concurrent corruption of a shared output root.

Reproducibility for benchmark pipeline runs is implemented in `src/distill_abm/pipeline/run_state.py`. This module computes deterministic run signatures from resolved input paths, file hashes, prompts, request defaults, selected summarizers, reference texts, and provider settings. It also supports resumable re-use when a cached `pipeline_run_metadata.json` matches the same signature and when required artifacts are still intact. The metadata written by this layer includes prompts, responses, scores, reference provenance, reproducibility signatures, and paths to debug-trace bundles.

Those debug-trace bundles deserve special attention because they are part of the verification design in their own right. They snapshot raw inputs, scoring references, request and response traces, summarization traces, artifact manifests, and input-validation warnings under a dedicated trace subtree. In effect, the repository preserves both the final outputs and the intermediate state needed to explain how those outputs were produced — which is essential for diagnosing unexpected results after the fact.

For smoke workflows, resumability is implemented through case manifests in `src/distill_abm/pipeline/smoke_manifests.py`. Successful cases persist a `case_manifest.json`; future runs can load only valid, successful manifests and skip redundant work. Response bundles are normalized in `src/distill_abm/pipeline/smoke_response_bundle.py`, which ensures that run-level and global CSV aggregations use a stable schema.

Why does this design matter? Reproducibility in LLM systems is inherently limited by provider nondeterminism: we cannot make providers deterministic. However, we can make the experimental inputs, requested treatments, stored outputs, and reuse decisions fully inspectable. That way, even when two runs produce different results, we can determine exactly what differed and why.

9. Monitoring and Reviewer Surfaces

The repository includes explicit operator-facing verification tools. `src/distill_abm/pipeline/local_qwen_monitor.py` renders a terminal dashboard for smoke and suite execution. It consumes filesystem snapshots rather than in-memory state, which makes monitoring compatible with resumable and nested runs. For suite execution, the stable root-level progress contract is `suite_progress.json`, refreshed through the full-case suite progress helpers. This means that a human reviewer can inspect run state while the workflow is still executing, rather than waiting for final reports — which is especially helpful for long-running suites where catching a problem early can save hours of wasted computation.

The static `review.html` viewer is the corresponding post hoc surface. Its payload builder in `src/distill_abm/run_viewer_payloads.py` resolves sampled-smoke and full-case runs into a typed JSON structure that includes prompt texts, evidence paths, outputs, errors, hyperparameters, validation states, and resume flags. This is a verification tool in the strict sense: it exists to make intermediate and final states visible to reviewers, so that they can assess the quality of a run without having to inspect raw files on disk.

10. Failure Modes Explicitly Exercised

The test suite and smoke workflows cover both nominal and failure behavior. We consider failure-path testing to be as important as happy-path testing, because experimental software often fails not because the happy path is impossible but because an intermediate artifact is malformed, missing, stale, or inconsistent with the experiment design. The verified failure modes include:

Failure Class	Verification Examples
Disallowed benchmark or debug-model use	`tests/e2e/test_cli.py`, `tests/unit/cli/test_policy.py`
Invalid configuration and malformed YAML	`tests/unit/configs/test_loader.py`
Missing or placeholder ingest artifacts	`tests/unit/ingest/test_ingest_smoke.py`, `src/distill_abm/pipeline/local_qwen_sample_smoke.py`
Structured-output failures and provider-specific retries	`tests/unit/pipeline/test_helpers.py`, `tests/unit/pipeline/test_local_qwen_sample_smoke.py`
Context-overflow and prompt-compression paths	`tests/unit/pipeline/test_local_qwen_sample_smoke.py`, `tests/unit/pipeline/test_full_case_smoke.py`
Resume-state corruption or incomplete metadata	`tests/integration/test_pipeline_run.py`, `tests/unit/pipeline/test_smoke_manifests.py`
Missing plots and invalid sweep prerequisites	`tests/unit/pipeline/test_smoke_optional_steps.py`
Invalid DOE inputs and zero-variance ANOVA cases	`tests/unit/eval/test_doe_full.py`
Generic unavailable model outputs	`tests/unit/pipeline/test_local_qwen_sample_smoke.py`
Repository-layout drift and missing benchmark assets	`tests/unit/repo/test_archive_layout.py`, `tests/unit/repo/test_benchmark_assets.py`

11. Interpretation

The testing and verification strategy we implemented in `distill-abm` should be understood as layered implementation assurance. It does not constitute formal proof, and it does not guarantee that every future provider response or every future ABM asset will behave identically. What it does provide is a strong, inspectable basis for confidence. We verify local logic, composed pipeline behavior, CLI behavior, pre-LLM artifact generation, resumability contracts, and reviewer-facing evidence surfaces. We preserve exact prompts, evidence paths, scores, signatures, and run reports in a way that makes failures diagnosable and successful runs auditable.

From a research perspective, this is the key point. The codebase was not only instrumented to produce outputs; it was instrumented to justify them. We designed the testing strategy to function as supplementary methodology: it documents how the software was constrained, observed, and checked so that experimental conclusions can be tied back to concrete executable evidence. In our experience, this is the most effective way to build confidence in an LLM-based research pipeline — not by assuming the software is correct, but by making it easy to verify that it is.