Finite Testability of Enterprise Software: A Quantitative Survey of Mutable State Across 200 Public Open-Source Codebases

Preprint. Paper 1 of the Open Honest empirical research program.

Adam Zachary Wasserman*
Open Honest

April 2026

Abstract

Enterprise software teams routinely report test-coverage figures in the 70–90 % range as evidence that their code is well-tested. We argue that test coverage, taken alone, is one signal among many — and not the most informative one — for what readers want to know about a codebase's overall health and verifiability. The Slop Audit instrument[2] combines twenty quantitative Layer 1 indicators with eighteen Layer 2 artifact inspections and eighteen Layer 3 marker assessments (across eighteen named compliance dimensions including security architecture, compliance engineering, operations, software architecture, and software development) into a single audit framework mapped to the published thresholds of named compliance regimes (SOC 2, NIST 800-53, OSFI B-13, OWASP ASVS, ISO/IEC 25010, and others). This paper reports the first preregistered cross-language measurement of one of those Layer 1 indicators.

The indicator measured here is the **mutable-state ratio** (L1.18 in the Slop Audit numbering): the proportion of functions in a codebase whose behavior depends on state outside their parameter list. Where this proportion is high, exhaustive behavioural testing is not possible regardless of test budget, because each mutable reference multiplies the effective state space. L1.18 belongs to the "Honest dimension" trio within the Layer 1 indicators (L1.18 mutable state, L1.19 decision-space coverage, L1.20 test determinism) that addresses finite testability specifically. We chose L1.18 as the first indicator to measure at corpus scale because it bears most directly on the gap between "executed" and "verified"; but it is one of twenty Layer 1 indicators, and the Layer 1 indicators are themselves only one of four layers in the audit instrument. Other Layer 1 indicators are mechanical computations from git history (delete/add ratio, net-negative commit frequency, fuzzy duplication, secret scan, type-escape density, god-file concentration, and twelve others). Layers 2-4 cover artifact inspection, qualitative marker assessment, and architectural synthesis. Companion preregistered studies measure additional indicators in turn.

We measured L1.18 across 200 public GitHub repositories (50 each in Python, Java, TypeScript, C#) using a tree-sitter-based analyzer validated by a 249-scenario behavioural specification. The median

*Open Honest is a FOSS project that publishes the Slop Audit (a 20-Layer-1-indicator, 18-dimension enterprise software quality audit instrument mapped to SOC 2, NIST 800-53, OSFI B-13, OWASP ASVS, and other compliance frameworks) and the Honest Framework (a correct-by-construction development methodology), governed by the Foundation under an immutable constitution. This paper measures one Layer-1 indicator from the Slop Audit. openhonest.org [PLACEHOLDER URL; confirm before submission].

[2]The Slop Audit is the quality-measurement instrument published by Open Honest. It is structured in four layers across eighteen named compliance dimensions. **Layer 1** comprises twenty quantitative indicators computed mechanically from git history and static analysis (rework patterns, code churn, secret scans, type-escape density, fuzzy duplication, god-file concentration, and the L1.18-L1.20 finite-testability trio). **Layer 2** comprises eighteen per-dimension artifact inspections with mechanical scoring (Present / Partial / Absent / Not Applicable). **Layer 3** comprises eighteen qualitative specified-marker assessments by a trained assessor (3-5 markers per dimension). **Layer 4** is architectural synthesis deferred to Phase 1 engagements. Compliance mappings cover SOC 2 Trust Services Criteria, NIST SP 800-53, OSFI B-13, NI 31-103, OWASP ASVS, ISO/IEC 25010, WCAG 2.2, Section 508, EN 301 549, AODA, and Quebec Law 25. This paper measures one Layer 1 indicator (L1.18) in isolation as a foundational empirical step; companion preregistered studies will address the other Layer 1 indicators and the higher layers.

ratio is 61.1 % for Python, 53.0 % for Java, 40.0 % for C#, and 14.8 % for TypeScript. TypeScript exhibits a bimodal distribution concentrated in a React-driven frontend subpopulation with ratios below 15 % and a NestJS / backend subpopulation with ratios in the 40–65 % range. Across all four languages only one repository (out of 200) falls below 15 % other than the 25 frontend-TypeScript repositories. We interpret this as evidence that **most enterprise-scale codebases are structurally incapable of exhaustive behavioural verification on this single indicator**, and that the picture revealed by L1.18 alone is sufficient motivation for the broader Slop Audit. These are initial results from the instrument's author; independent third-party validation by trained external assessors is the subject of Paper C in the same preregistered DBSYG block (OSF DOI 10.17605/OSF.IO/DBSYG; Paper C registered 2026-04-10) and is not addressed here. Results are reproducible in under 20 minutes on a standard development machine. The analyzer and its behavioural specification are released with the paper.

Keywords: mutable state, finite testability, Slop Audit, static analysis, cross-language, empirical software engineering, tree-sitter, behavioural specification.

1. Introduction

Enterprise software teams report test-coverage figures — typically in the 70–90 % range for well-run engineering organizations — as evidence that their code is meaningfully tested. These figures are produced by tools (JaCoCo, Coverage.py, Istanbul, dotnet coverlet) that measure which lines or branches of source code are executed when the test suite runs. High line coverage is treated as a proxy for test thoroughness and, transitively, for software quality. A substantial literature argues that line-coverage figures in this range are not strongly correlated with test effectiveness at detecting faults (Inozemtseva & Holmes 2014), but the critique has not meaningfully changed industry practice. Most coverage tools still measure execution. Most dashboards still report percentages. Most engineering leaders still take 80 % coverage to mean "mostly tested."

This paper measures a property of source code that line coverage does not capture: **the fraction of functions whose behaviour depends on state the function does not receive as a parameter**. Where this fraction is high, exhaustive behavioural testing is not possible regardless of the size of the test suite, because each mutable reference multiplies the effective state space. A function that reads `self.cache` does not just have a behaviour per input; it has a behaviour per (input $\times$ prior state of self.cache). Enumerating the resulting test cases to "cover" the behaviour space in a strong sense is generally infeasible.

We call this fraction the **mutable-state ratio** and denote it L1.18 (its position in a broader twenty-indicator Slop Audit framework). The ratio is intentionally simple: it counts functions, not statements or branches. For a function to avoid being flagged, it needs to express its dependencies through parameters and its effects through return values. No attribute accesses on `self` / `this`, no reads of module-level mutable names, no bare identifier references to class fields without `this.`. A function that meets this criterion has a finitely enumerable behaviour space (modulo its input domains), and its correctness can in principle be reasoned about from its signature and body alone.

We measure L1.18 across 200 public GitHub repositories in four widely-used languages: Python, Java, TypeScript, and C#. Per language we select the 50 highest-starred qualifying repositories that appear to be applications rather than libraries. We analyze 2.2 million functions in total.

The headline finding is that **most repositories in three of the four languages — Python, Java, and C# — exhibit mutable-state ratios above 40 %**. In Python the median is 61 %; in Java 53 %; in C# 40 %. TypeScript is the outlier: its median is 15 %, and its distribution is bimodal, with a large cluster in the Healthy band (< 15 %) concentrated in React-ecosystem frontend applications, and a smaller cluster in the Not-Healthy / Slop region containing backend TypeScript (NestJS), Angular services, and class-heavy desktop/game codebases like VSCode and PixiJS.

Across all 200 repositories only 26 fell below the 15 % threshold. 25 were TypeScript frontend applications. One was a Java authentication library (`justauth/JustAuth`, 12.3 %). No Python, no C#, no TypeScript backend, and only one Java project in the sample of 200 was structurally testable in the sense L1.18 defines.

We interpret this as evidence that **ecosystem-level paradigm choices, rather than individual engineering discipline, determine whether a codebase is structurally testable**. The TypeScript frontend cluster exists because React's hooks-era paradigm (pure components, immutable state, props-in / JSX-out) structurally prevents the ratio from climbing. The Python, Java, and C# equivalents do not exist in our corpus not because no individual team could achieve them, but because the dominant paradigms in those ecosystems (Django/Flask with class-based views; Spring with `@Component`-annotated bean graphs; .NET with `IOptions<T>` and DI containers) actively push code in the opposite direction.

1.1 Contributions

1. **A cross-language operationalization of mutable-state ratio (L1.18)** that measures the same underlying property — function-local versus function-external state access — in four languages with different syntax and paradigm conventions.
2. **A 200-repository corpus** of L1.18 measurements, released in full alongside this paper along with the analyzer source code and all intermediate and final result artifacts.
3. **A tree-sitter-based analyzer validated by a 249-scenario Gherkin behavioural specification**, reproducible end-to-end in approximately 20 minutes on a standard developer machine.
4. **A documented analyzer bug history**: three iterations of the analyzer, each producing different aggregate numbers, with the full validation trail included as Appendices A–D. This directly addresses a common concern about the reliability of static analysis tooling at scale.
5. **An exploratory observation of the TypeScript bimodal distribution** correlated with frontend/backend framework choice, forming the pre-registered hypothesis for a confirmatory follow-up study (Paper F in the Open Honest research program).

1.2 Paper organization

Section 2 surveys related work. Section 3 defines the mutable-state ratio, describes the corpus, documents the analyzer, and enumerates the validation layers applied to it. Section 4 presents the results. Section 5 interprets them. Section 6 acknowledges limitations. Section 7 situates this paper within a broader preregistered research program. Section 8 concludes. Appendices A–D extend the methodology disclosure.

2. Related work

This section is intentionally partial; expert review is expected to add, remove, and reorder citations. The organization reflects the categories of work the paper engages with.

2.1 Line coverage and its limits

The empirical literature on test-coverage effectiveness has consistently found that coverage percentages are weak predictors of test-suite fault-detection ability. Inozemtseva and Holmes (2014) measured the correlation between coverage and effectiveness directly on a corpus of Java projects and found that once coverage is reasonably high (in the 60-80 % range that most industrial teams target), marginal additions to coverage yield little improvement in test-suite quality. Our paper approaches the same problem from a different angle: rather than asking whether high coverage implies good tests, we ask whether the *codebase structure* of typical enterprise software permits behavioural testing at all. If a codebase is 60 % mutable-state-dependent, then no coverage figure — high or low — can claim behavioural exhaustiveness.

2.2 Mutable state, pure functions, and testability

The argument that pure functions are more testable than methods on mutable objects has a long pedigree in software engineering, tracing at least to Backus (1977) and formalized in the functional-programming tradition (Hughes 1989). Our contribution is to measure the prevalence of the property, not to re-argue its desirability. Where prior literature has argued functional code *should* be preferred, this paper asks: *how much of real-world enterprise code actually satisfies the pre-condition*? The answer at corpus scale: very little outside the React/TypeScript frontend cluster.

2.3 Compiler-enforced vs. framework-enforced purity

Three tiers of enforcement exist in practice for the property L1.18 measures.

Tier 1: compiler-enforced purity. Haskell's IO monad (Peyton Jones 2001) makes side effects visible at the type level; a function without `IO` in its return type is provably pure by construction. Elm (Czaplicki 2012) applies the same principle to frontend code. Dependent-type systems (Idris, Agda, Coq) go further, but remain academic. These languages achieve L1.18 ≈ 0 % by definition: the compiler rejects impure code that is not explicitly marked. The tradeoff is adoption cost. Haskell, Elm, and the dependently-typed languages collectively account for fewer than 3 % of developer usage; humans systematically prefer languages that permit informal reasoning about state (the Wason Selection Task literature — Wason 1966, Evans 2016 — predicts exactly this), and Tier 1 languages close those "escape hatches" entirely.

Tier 2: language-level immutability, convention-enforced purity. Elixir/Erlang (Armstrong 2007) makes data structures immutable, but GenServer processes carry mutable state across calls, and the BEAM actor model permits side effects (message passing, I/O) without type-level marking. Clojure (Hickey 2008) provides persistent immutable data structures as defaults, but atoms, refs, and agents are explicit mutation points. F# and OCaml are functional-first but allow mutable references. In all four, purity is a cultural norm, not a compiler guarantee. A GenServer-heavy Elixir codebase can exhibit high L1.18; a discipline-driven one can approach zero. These languages sit between Tier 1 and the enterprise mainstream: they make impurity inconvenient without making it impossible.

Tier 3: framework-enforced purity in imperative languages. The Honest Framework (Wasserman 2026) provides enforcement through bounded-vocabulary enumeration rather than compiler type-checking. Three mechanisms operate at different lifecycle stages: a pre-commit linter (`honest-check`) that flags functions accessing global state; an exhaustive test harness (`honest-test`) that calls every function twice with identical inputs, instruments for mutation and global reads, and fails on any impurity signal; and a type system (`honest-type`) whose Set-based recognizers make the input space finite and enumerable at definition time, enabling exhaustive behavioural verification via a `for` loop rather than a type proof. For bounded vocabularies — which constitute the majority of enterprise type spaces — the mathematical guarantee is identical to Tier 1: every input is tested, every output is verified, every side effect is detected. The enforcement mechanism differs (enumeration vs. type proof); the coverage does not. Honest Code achieves this in languages where the compiler offers no purity enforcement: Python, TypeScript, Java, C#, Ruby, and PHP.

This paper measures L1.18 without regard to which tier produced the observed ratio. The measurement is structural: it reports *how much* of a codebase is pure, not *how* the purity was achieved. The three-tier framework is relevant context because it explains why the enterprise languages in our corpus (Python, Java, TypeScript, C#) show high L1.18: they offer no Tier 1 or Tier 2 enforcement and, without a Tier 3 framework, purity is entirely optional. The React-frontend TypeScript cluster (§5.2) is the one subpopulation in our corpus where a Tier-2-like convention (React's functional-component pattern) drives L1.18 below 15 % without compiler enforcement.

2.4 Cross-language empirical studies

Ray et al. (2014) studied the relationship between programming language and software defect density on a 729-project GitHub corpus. Their methodology — filtering for actively-maintained repositories, controlling for project size and domain — is a direct influence on our corpus selection. Where Ray et al. found relatively modest language effects on defect density, we find substantial language/ecosystem effects on mutable-state ratio, suggesting that the two constructs are distinct and that mutable-state ratio may be the more informative variable for the subset of quality concerns related to testability.

2.5 Tree-sitter as an analysis platform

Tree-sitter (Brunsfeld et al.) has become the de-facto incremental parser for developer tooling; its grammars are maintained for most widely-used languages. Its use in empirical static analysis at the corpus scale described here is, to our knowledge, less common than its use in editors. Our analyzer's polymorphic design (`LANG_CFG` driving a single analysis pipeline across four grammars) is one contribution toward making tree-sitter more reusable for cross-language corpus work.

2.6 Behavioural specification for analysis-tool validation

Literature search in progress. Tests for static analysis tools are usually written as example-based unit tests (the input $\rightarrow$ expected-output style shown in the SpotBugs / PMD / SonarQube test suites). Pre-committed behavioural specifications in a Gherkin or property-based style are less common, though the practice is well-established for application-level testing (Cucumber, SpecFlow, pytest-bdd). We use Gherkin here primarily because the temporal discipline (specification before implementation) caught bugs that example-based unit tests had missed. Whether the Gherkin syntax itself confers additional benefits, particularly in AI-assisted code generation contexts where a natural-language-like specification may be better aligned with the assistant's training distribution, is an open empirical question we leave to future work. Appendix C and the companion methods note discuss the temporal-discipline finding at length.

2.7 AI-assisted empirical software engineering

Open area. The literature on using AI coding assistants (Copilot, Claude, Cursor) as research infrastructure is still forming. Our companion methods note is one case study in this emerging area. We are aware of active discussion in the SIGSOFT community (2024–2026) about disclosure norms and reliability of AI-produced analysis code but have not yet identified peer-reviewed work that directly addresses this topic for empirical software engineering research.

3. Methods

3.1 Definition of the L1.18 mutable-state ratio

For a given codebase, L1.18 is computed as:

> (number of functions that reference state outside their parameter list) divided by (total number of functions analyzed)

expressed as a percentage. The **total** count excludes dunder methods other than `__init__` (for Python), `constructor` (for TypeScript class constructors), and functions whose names match an I/O-boundary heuristic (substrings including `handler`, `endpoint`, `main`, `route`, `bootstrap`, etc., varying by language; full list in Appendix D). These exclusions reflect the paper's concern with interior business-logic code, not glue.

A function is considered to "reference state outside its parameter list" if it contains at least one of:

- An attribute access of the form `self.X` or `cls.X` (Python), `this.X` (Java/TypeScript/C#), where the receiver is the enclosing class or record.
- A bare identifier reference that resolves to a class field or property declared in the enclosing class and that is not shadowed by a local declaration or parameter (Java and C# only; the "implicit-member" case).
- A bare identifier reference to a name declared mutable at module scope, where "mutable at module scope" means:
  - Python: any top-level `X = ...` assignment, any augmented assignment (`X += ...`) anywhere, or any subscript assignment (`X[k] = ...`) anywhere.
  - TypeScript: any top-level `let X` or `var X` declaration.
  - Java and C#: the language-level module mutability semantics do not permit module-level variables in the same sense; these languages contribute no module-mutable cases.

Tests of this operationalization are documented in the behavioural specification (Appendix C).

3.2 Corpus selection

The corpus was constructed by `corpus_query.py` (included in the replication package, Section 6). For each target language (Python, Java, TypeScript, C#):

1. GitHub search was used to retrieve the top 500 repositories by star count matching the query `language:<X> stars:>=100 archived:false fork:false`.
2. Each candidate was qualified against two filters:
   - **Application indicator.** The repository must contain at least one file matching a language-specific application pattern (Dockerfile, `manage.py`, `Program.cs`, `application.properties`, `.github/workflows/*`, etc.). This excludes pure libraries.
   - **Test presence.** The repository must contain at least one test file matching a language-specific test pattern (`test_*.py`, `*Test.java`, `*.test.ts`, `*Test.cs`, etc.).
3. The first 50 qualifying repositories per language were accepted.

Total: 200 repositories, 50 per language. The full corpus list is in the replication package (`{python,java,typescript,csharp}_repos.txt` and the combined `corpus.json`).

3.3 Analysis implementation

A single unified analyzer (`l1_18.py`, ~900 lines of Python) handles all four languages. The analyzer uses `tree-sitter` (Brunsfeld) to produce an abstract syntax tree, then applies language-specific queries driven by a central configuration dictionary (`LANG_CFG`) that captures node-type names, field-access conventions, and exclusion rules per language. All language-specific differences are expressed as data in `LANG_CFG` or in three small dispatch tables (`MODULE_MUTABLE_DETECTORS`, `FUNCTION_NAME_EXTRACTORS`, `NAME_SKIPS`). No analysis code is duplicated across languages.

The analyzer was developed in three iterations, each with documented bugs corrected by the next (see Appendix A for the bug catalogue). The version whose results are reported here (v3) was validated against a 249-scenario Gherkin behavioural specification (Appendix C). On the full 200-repo corpus, v3 completes in approximately 20 minutes of wall-clock time on a single-node developer machine; a reviewer running the replication package can reproduce the analysis within one working hour.

3.4 Classification bands

For purposes of reporting, repositories are classified into three bands:

- **Healthy:** ratio < 15 %
- **Not Healthy:** 15 % $\leq$ ratio < 40 %
- **Slop:** ratio ≥ 40 %

We note that these band thresholds are **provisional**. They were chosen by author judgment before corpus analysis began and are used here for descriptive convenience, not as claims of empirical significance. A follow-up preregistered study (Paper E in this program) empirically calibrates the thresholds against clustering structure and rework-ratio outcomes; readers who require empirically justified thresholds should reference that study when it becomes available. Where this paper makes cross-language comparisons, it reports raw ratios alongside the band classifications so that readers can form their own judgments independent of the band choices.

3.5 Validation

Four validation layers were applied before v3 results were accepted:

1. **Differential comparison** against two prior analyzer implementations (v1 regex-based, v2 first-pass tree-sitter). Aggregate disagreements exceeding 5 percentage points per repo were individually audited; four bugs in v1 and two in v2 were identified and corrected.
2. **Behavioural specification** via Gherkin (249 scenarios; Appendix C). Every function in the analyzer is exercised by at least one scenario. Scenarios were written with adversarial intent, enumerating edge cases (varargs, expression-bodied methods, deeply nested expressions, directories with file-suffix names) before the underlying code was examined.
3. **Full-corpus re-run** with the validated analyzer to surface environment-level issues (stack overflow on deeply-nested expressions, `IsADirectoryError` on directories with file-suffix names in C# repos) that the Gherkin scenarios had not covered.
4. **Preserved prior artifacts.** All three result sets (`results_v1_regex_buggy/`, `results_v2_tree_sitter_buggy/`, `results/`) are included in the replication package. Readers can independently verify the corrections reported here.

4. Results

4.1 Per-language descriptive statistics

Table 1 summarises the per-language distributions across the 200-repo corpus.

Table 1. Per-language summary statistics (v3 validated analyzer).

Language	n	Mean ratio	Median ratio	Range	Total functions analyzed	Healthy	Not Healthy	Slop
Python	50	60.0 %	61.1 %	20.8 – 90.9 %	239,452	0	3	47
Java	50	53.1 %	53.0 %	12.3 – 80.3 %	1,028,640	1	6	43
TypeScript	50	21.6 %	14.8 %	0.0 – 62.8 %	323,127	25	18	7
C#	50	40.0 %	40.0 %	20.1 – 67.9 %	610,424	0	25	25

Aggregate function count across the corpus: 2.20 million functions.

Figure 1 (violin plot of per-language distributions) shows the overall shape. Python, Java, and C# all have median ratios in the Slop band ($\geq$ 40 %) and no or near-zero repositories in the Healthy band. TypeScript's distribution is qualitatively different: half of the TypeScript corpus falls in the Healthy band, and the distribution is bimodal.

4.2 TypeScript's bimodal distribution

Figure 2 plots the TypeScript distribution as a histogram. Two modes are visible: a large Healthy-band peak concentrated in the 0 – 15 % range (15 repositories with ratio < 5 %, 25 repositories total below 15 %), and a second mode in the 25 – 40 % band with a Slop-band tail extending to 65 %.

Inspection of the Healthy-band TypeScript repositories shows they are overwhelmingly React-ecosystem frontend applications (Next.js, shadcn-ui, Chakra UI, Ant Design, Supabase dashboard, Vue, Nuxt, Astro, etc.). The second mode contains backend TypeScript applications (NestJS-based servers, Socket.IO, Angular non-frontend services), electron-style desktop applications (VSCode), and class-heavy infrastructure (Microsoft Playwright, the PixiJS game engine).

We flag this bimodal pattern as an **exploratory observation**. Paper F in the Open Honest research program preregisters a confirmatory test of the frontend/backend split as a predictor of TypeScript mutable-state ratio.

4.3 Cross-language comparison

Figure 3 (cumulative distribution by language) makes the cross-language comparison explicit. TypeScript dominates the left portion of the graph (low ratios); Python, Java, and C# dominate the right. The ordering by median is TypeScript

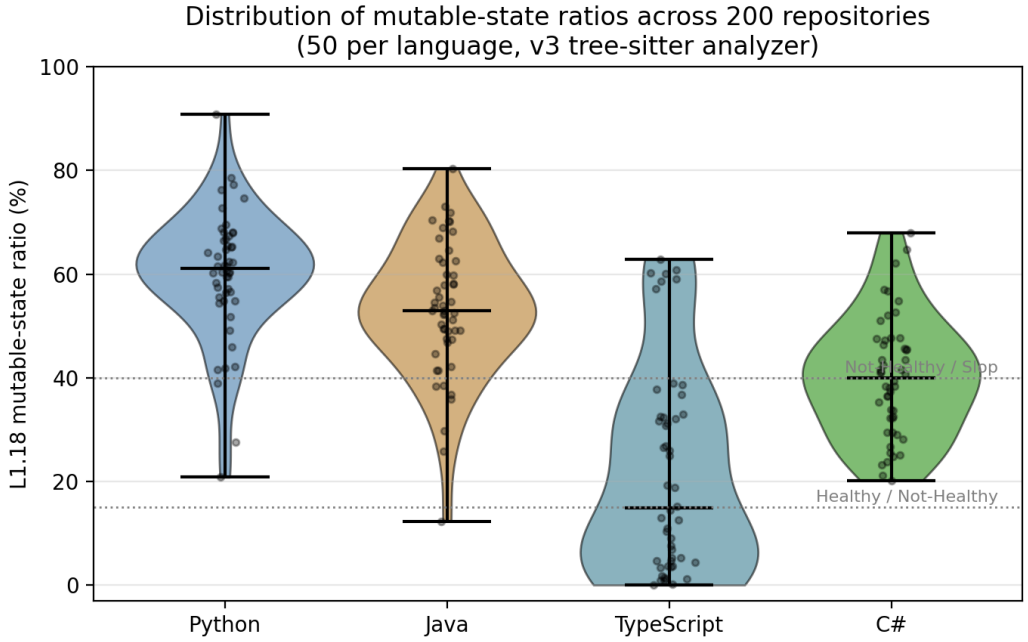

Figure 1: **Figure 1.** Distribution of L1.18 mutable-state ratios across 200 repositories. Violin plots show per-language density; each dot is one repository. Dotted lines at 15 % and 40 % mark the Healthy / Not-Healthy / Slop band thresholds.

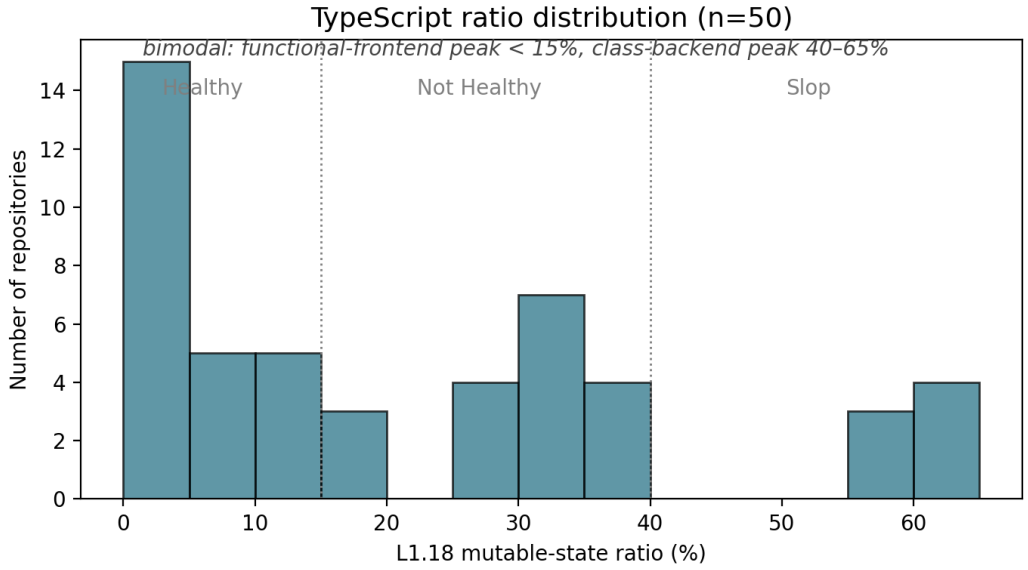

Figure 2: **Figure 2.** TypeScript ratio distribution (n=50). Bimodal: a large Healthy-band peak below 15 % (React-ecosystem frontends) and a second mode in the 25-40 % band with a Slop-band tail extending to 65 % (NestJS/backend and class-heavy codebases).

≪ C# < Java < Python. Python's higher median than Java is a finding that emerged only in v3 (see Appendix A, bug V2-02, and Appendix B): prior analyzer versions systematically underestimated Python's mutable-state ratio by omitting subscript-assignment patterns (`CACHE[key] = value`) from module-mutable detection.

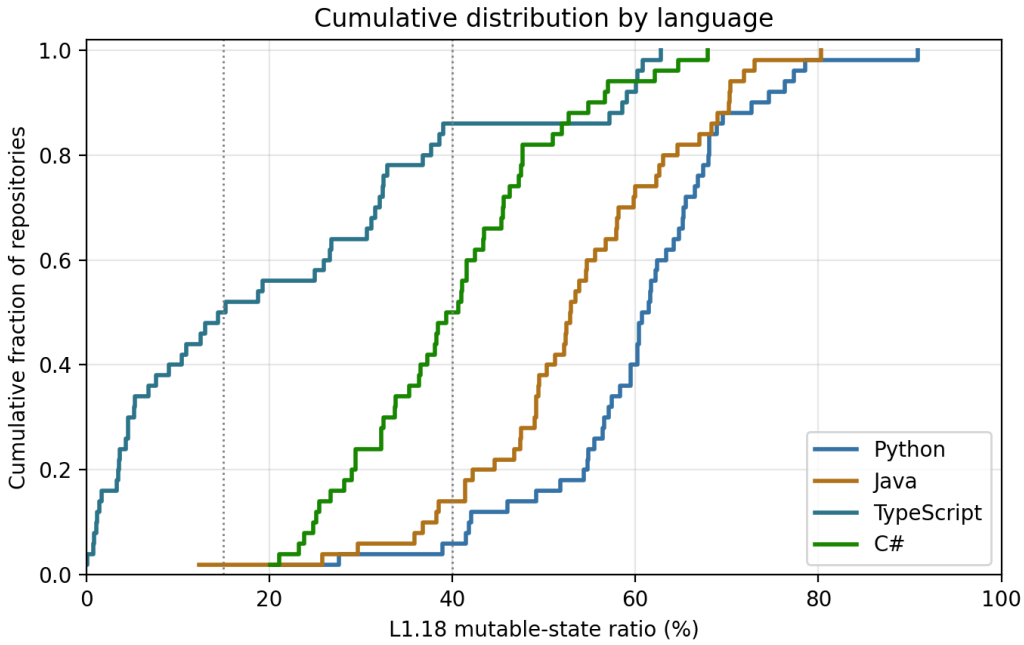

Figure 3: **Figure 3.** Cumulative distribution function by language. TypeScript dominates the left (low-ratio) portion of the graph; Python, Java, and C# dominate the right.

4.4 Size independence

A common concern in corpus-scale studies is that findings may be confounded by codebase size (e.g., larger codebases might accumulate more mutable state). Figure 4 plots L1.18 ratio against total function count per repository (log x-axis). No strong size dependence is visible. The languages remain clearly separated across three orders of magnitude of codebase size, from ~100-function tutorial repositories to 270,000-function monorepos (openjdk/jdk, oracle/graal, dotnet/runtime, mono/mono).

Spearman rank correlation between log(total_functions) and ratio:

Language	rho	p-value	Direction
Python	+0.358	0.011	Larger → higher ratio
Java	−0.283	0.046	Larger → lower ratio
TypeScript	+0.190	0.186	Not significant
C#	+0.140	0.332	Not significant

Two languages show statistically detectable size effects at $\alpha = 0.05$ but in opposite directions. Python's positive effect suggests larger Python codebases accumulate more mutable state, consistent with a "framework- driven" interpretation: big Python repositories in this corpus are overwhelmingly ML/infrastructure projects (tensorflow/models, ray, huggingface/transformers, sentry, airflow) whose scale comes in part from extensive class-based configuration machinery. Java's *negative* effect is more surprising: the largest Java repositories in the corpus (openjdk/jdk, oracle/graal) are compilers and runtimes whose internal structure includes substantial compile-time utility code that is less mutation-heavy than typical enterprise Java. We do not treat these correlations as strong evidence of size-ratio causality; they are reported for completeness. The figure confirms that the four language distributions remain clearly separated across three orders of magnitude of codebase size, which is the headline result.

4.5 The full-corpus verdict

Across 200 repositories:

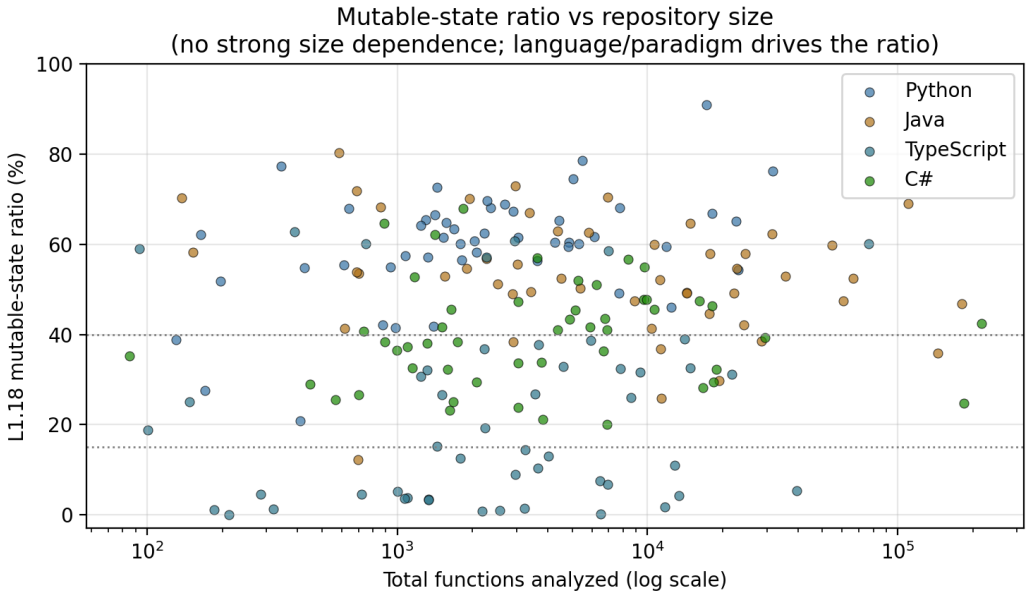

Figure 4: **Figure 4.** L1.18 mutable-state ratio vs repository size (total functions analyzed, log x-axis). The languages remain clearly separated across three orders of magnitude of codebase size.

- 26 repositories (13 %) classify as Healthy. 25 of these are TypeScript (specifically, TypeScript repositories that appear to be React-ecosystem frontends). The 26th is `justauth/JustAuth`, a small Java authentication library.
- 40 repositories (20 %) classify as Not Healthy.
- 134 repositories (67 %) classify as Slop.

Put differently: for any public codebase in the 100+ star range in Python, Java, or C#, the odds of finding one with mutable-state ratio below 15 % are approximately zero. Within TypeScript, the odds depend sharply on whether the project is frontend or backend.

5. Discussion

5.1 What a 60 % mutable-state ratio means for testability

A function that takes `(a, b)` as parameters and returns `a + b` has a behaviour space that is exactly the Cartesian product of its input domains. It can, in principle, be exhaustively tested by covering that product; and for most input types the product is at least enumerable in a useful sense. A function that takes `(a, b)` but also reads `self.cache` has a behaviour space that is the product of its input domains *and* the current state of `self.cache` at the moment of invocation. If `self.cache` is a dictionary that accumulates during program execution, the behaviour space is effectively unbounded; any test case must specify not just `(a, b)` but the entire prior execution trace that produced the relevant `self.cache` state. For complex programs with many such functions, the number of distinct behavioural situations becomes combinatorially intractable. Line coverage captures none of this: executing such a function once, under a particular `self.cache` state, registers as 100 % coverage of its lines without exercising any meaningful fraction of its behaviour.

When we report that the median Python repository in our corpus has a mutable-state ratio of 61 %, we are reporting that in the median case, roughly three out of every five functions in the codebase have this property. Line coverage on such a codebase measures which lines execute. It does not measure whether the behaviour space around the execution is tested. A test suite that achieves 80 % line coverage on a 61 %-mutable- state codebase is testing ~80 % of executions of ~39 % of the behaviour- enumerable functions, plus ~80 % of executions of ~61 % of the behaviour- non-enumerable functions where "executed" has a weaker epistemic meaning. Reporting the first figure without the context of the second is, we argue, systematically misleading about what has been verified.

5.2 Paradigm, not language, drives the ratio

The cleanest interpretation of our cross-language results is that **the paradigm predominantly used by a language's community**, not the language itself, drives the observed ratio. Three lines of evidence support this:

First, **TypeScript bimodality**. TypeScript supports both class-based OOP and pure-function patterns. Its corpus is split almost cleanly into a functional-React cluster below 15 % and a class-based backend cluster at 40 %+. Language affordance is held constant; paradigm choice varies; the ratio follows the paradigm. (Appendix C documents the operationalization well enough that readers skeptical of this interpretation can replicate.)

Second, **Python's high ratio is a framework story**. Python supports pure functions excellently; no language feature forces `self.cache` patterns. But the dominant Python frameworks in our corpus — Django's class-based views, Flask's `g` globals, the class-heavy transformers library, Ray actors — make mutable class state the path of least resistance. The language permits functional code; the ecosystem does not reward it.

Third, **the single non-TypeScript Healthy repository in our corpus, `justauth/JustAuth`, is also an ecosystem outlier**: a small, single-purpose Java utility library that adopts a builder-and-static- method style rather than the typical Spring-dependency-injection graph. A single outlier within an otherwise uniform 150-repo distribution is consistent with the ecosystem-paradigm reading rather than a language-capability reading.

If this interpretation is correct, the prescription for enterprise software is not "switch languages" but "adopt a paradigm." We return to this theme in §5.4.

5.3 What industry coverage reports actually measure

A Python codebase with 61 % mutable-state ratio and 80 % line coverage is executing most of its lines during the test suite. It is not exhaustively testing the behaviour that depends on `self.cache`-style state. For the 39 % of functions that are purely parametric, 80 % line coverage is a reasonable proxy for 80 % behaviour coverage, within known limits. For the 61 % that are not, 80 % line coverage is a statement about execution under the particular pre-states the test suite happened to construct; which may or may not resemble production states.

We do not claim coverage figures are useless. We do claim that reporting coverage without qualifying it by mutable-state ratio (or an equivalent measure of behavioural enumerability) produces systematically misleading impressions of what has been verified. The industry practice of reporting coverage alone as evidence of testing thoroughness should, on the evidence here, be qualified.

5.4 Honest Code principles as structural enabler

A separate literature (the Honest Code principles; the Honest Framework specification published by Open Honest) advocates a set of architectural rules — pure functions, dispatch tables, I/O-at-the-boundary, no classes except for TypedDicts — that structurally prevent the ratio from climbing. The React-frontend TypeScript cluster in our corpus shows that codebases built on roughly these principles, even without Honest Code language *per se*, achieve ratios structurally bounded in the Healthy band. This is not evidence that Honest Code principles *cause* low ratios; it is evidence that the principles and the observed outcome are consistent.

A sharper test would be a before/after measurement of a single codebase adopting the principles. That test is pre-registered as Paper D in this program (Wasserman & Staff, in preparation), which measures rework metrics on two production codebases before and after their authors adopted the principles.

5.5 The inverse argument: what low L1.18 structurally eliminates

The preceding sections argue that high L1.18 creates specific problems: unbounded behaviour spaces, misleading coverage figures, cognitive escape hatches that enable informal reasoning about state. The inverse is equally important and often overlooked: **when L1.18 approaches zero, entire categories of defect become structurally impossible, not merely unlikely**.

This is not a statistical claim about defect rates. It is a mathematical consequence of the constraint "no external mutable state":

- **Race conditions** require shared mutable state. Pure functions share nothing; concurrent invocations cannot interfere.
- **Order-dependent test failures** require evaluation-order-sensitive state. Pure functions produce the same output regardless of the order in which they or other functions are called; L1.20 (test determinism) approaches 5/5 by construction.
- **Stale-cache and stale-reference bugs** require a mutable cache or reference that drifts from its source of truth. No mutable state means nothing to go stale.

- **Side-effect interference** — one function's invisible write corrupting another function's read — requires implicit shared state. All dependencies are parameters; nothing is implicit.
- **Null-from-uninitialized-state** bugs require a variable whose value depends on whether prior code has executed. Pure functions do not depend on initialization order.
- **State-space explosion in testing** — the combinatorial intractability described in §5.1 — requires the `State(S)` term in the behaviour domain. When `State(S)` is empty, the behaviour domain is the finite Cartesian product of parameter domains, and exhaustive testing becomes tractable.

Each elimination is a logical entailment of the definition of "pure function" (output determined solely by parameters; no reads or writes to external state). L1.18 measures how large the population of functions carrying these exposures is. A codebase at L1.18 = 0 % is not merely "well-structured" or "easy to test." It is structurally immune to the above defect categories. No amount of testing, review, or debugging discipline can achieve the same guarantee for a codebase at L1.18 = 60 %; the guarantee is architectural, not procedural.

The cognitive dimension reinforces the structural one. Humans systematically fail at the conditional logic that mutable-state code demands (the Wason Selection Task; Wason 1966, Evans 2016; fewer than 10 % correct on first attempt). Mutable state provides what Wasserman (2025) calls "escape hatches from formal reasoning": mutation replaces provable transformation with informal assertion, and side effects replace explicit dependency tracking with implicit coupling. When L1.18 approaches zero, these escape hatches close: every dependency is a parameter, every transformation is a return value, and the formal-logic structure of the code is visible on inspection. The programmer cannot avoid reasoning formally about the code because the code leaves no room for informal shortcuts. This is harder for humans (Haskell's learning curve is the canonical example), but it eliminates the cognitive failure modes that produce the defects listed above.

The practical implication for enterprise auditing is that L1.18 is not merely a structural metric; it is a **defect-category predictor**. An auditor who observes L1.18 = 5 % can report, with mathematical certainty, that the six defect categories above are confined to at most 5 % of the codebase's functions. An auditor who observes L1.18 = 60 % cannot make any such claim. The auditor's report is therefore not "this codebase has a high/low number" but "this codebase is structurally exposed to / structurally immune from the following defect categories, to the following degree."

5.6 The "Slop" terminology

We use the terms **Healthy**, **Not Healthy**, and **Slop** as technical labels for ratio bands, not as aesthetic judgments. They derive from a broader Slop Audit framework in which 19 other structural indicators are defined alongside L1.18. The terms are deliberately provocative; we think the provocation is warranted given the gap between industry testing claims and the structural facts reported here. A repository labelled Slop is not "bad code" in any global sense; many of the Slop repositories in our corpus are widely used, heavily tested, and solve real problems well. The label reports a score on one of twenty Layer 1 indicators, not a comprehensive judgment: although a codebase that scores Slop on even a quarter of the twenty indicators is, by any practitioner reading, in structurally rough shape.

5.7 A note on the measurement process

Developing the analyzer that produced these numbers required three iterations (Appendix A). The first was regex-based and contained bugs that silently under-counted Python's ratio by approximately 11 percentage points. The second was tree-sitter-based but inherited related errors in field-name conventions; those added another 3-point correction in Python. The third was validated against a 249-scenario behavioural specification (Appendix C) before being accepted. At each stage, the authors' manual sanity checks passed. At each stage, the bugs were invisible without systematic adversarial testing.

This matters for two reasons. First, the preserved v1/v2/v3 result artifacts let reviewers verify the corrections rather than take them on faith (Appendix B). Second, the experience itself reinforces the paper's thesis. If three iterations of a purpose-built measurement instrument, developed with care by authors who knew the property they were trying to measure, each produced systematically wrong numbers that passed their authors' casual checks, then the coverage-report numbers produced continuously by uninstrumented tools across tens of thousands of enterprise codebases are presumably at least as unreliable. The methodology demonstrated here (preregistration, exhaustive behavioural testing, public diff against prior implementations, all intermediate artifacts preserved) should be treated as a minimum bar, not a gold standard, for any numerical claim about software quality at scale.

6. Limitations

6.1 Threshold provisionality

The Healthy / Not Healthy / Slop bands at 15 % and 40 % are author-chosen and not empirically calibrated. Readers who disagree with these thresholds should focus on the raw ratios reported in Table 1 and the figures. Paper E in this program empirically calibrates the thresholds using cluster analysis and rework correlation.

6.2 Corpus selection

The corpus is limited to public GitHub repositories with $\geq$ 100 stars that pass a coarse "is an application" heuristic. The sample is biased toward popular, publicly visible, well-maintained code; closed-source enterprise codebases may exhibit different distributions. Within each language, the sample is a convenience sample of the 50 highest-starred qualifying repositories rather than a stratified random sample across domains.

6.3 Ecosystem vs language confound

The cross-language comparison cannot disentangle the effect of the language itself (supported syntax, idioms) from the effect of the ecosystem (dominant frameworks, community norms). The TypeScript finding in particular is driven by React's dominance in the frontend corpus and NestJS's presence in the backend corpus. A carefully matched cross-language comparison (e.g., Java Spring Boot vs Python Flask vs NestJS) would better isolate the language effect.

6.4 L1.18 operationalization

Our operationalization of "references mutable state" depends on several judgment calls: the I/O-boundary exclusion list, the dunder-method exclusions, and the treatment of `self.X` / `this.X` as external state access. Each of these decisions is documented in Appendix D and externalized in `LANG_CFG`; readers who prefer different choices can modify the analyzer and re-run.

6.5 Prior-implementation bugs

Earlier versions of the analyzer produced numerically different aggregate numbers. Bugs are enumerated in Appendix A; deltas are documented in Appendix B. Readers should use v3 results (reported here); prior artifacts are preserved in the replication package for verifying corrections.

6.6 Generalization to proprietary code

We make no claim that the 60 % / 53 % / 41 % / 22 % medians measured here generalize to the private repositories of any particular enterprise. We claim only that the 200 public repositories measured here — selected by criteria intended to exclude libraries and small tutorials — predominantly exhibit mutable-state ratios above what would permit exhaustive behavioural testing.

7. Future work

The following preregistered studies in the Open Honest research program extend the work reported here:

- **Paper B.** Paradigm-AI compatibility: whether AI code generation is affected by the paradigm (pure functions, dispatch tables, class hierarchies).
- **Paper C.** Instrument validation: whether independent assessors using the same methodology produce the same classifications.
- **Paper D.** Rework comparison on specific codebases before and after adoption of Honest Code practices.
- **Paper E.** Empirical threshold calibration via cluster analysis and rework correlation on the present corpus.
- **Paper F.** Confirmatory test of the TypeScript frontend/backend bimodal split observed here as an exploratory finding.
- **Paper G.** Multidimensional calibration across all 19 Slop Audit indicators (of which L1.18 is one).
- **Paper H.** Architectural coherence under multi-turn AI development on a bounded enterprise application, testing whether the paradigm effect on correctness compounds over development turns.

8. Conclusion

We measured the mutable-state ratio across 200 public repositories in four languages. The median ratio was above 40 % in Python, Java, and C# and bimodally distributed in TypeScript. Across all four languages, only one non-TypeScript repository in the corpus fell below 15 %. These ratios are inconsistent with the assumption of exhaustive behavioural testability at scale, regardless of reported line-coverage figures. The TypeScript frontend subpopulation shows that structurally-testable code at scale is achievable when the dominant paradigm supports it; the sparsity of comparable examples in Python, Java, and C# shows that such achievements are rare in practice. We interpret this as evidence that ecosystem-level paradigm choices, rather than individual engineering discipline, determine whether a codebase is structurally testable.

References

This list is preliminary. Several sections (particularly Related Work, §2) will require additions after expert review. Citations marked [VERIFY] should be checked by the author against the cited source; citations marked [TBD] are placeholders for categories of work we intend to cite but have not yet identified the specific reference.

Static analysis and coverage critique.

[VERIFY] Inozemtseva, L. & Holmes, R. (2014). Coverage Is Not Strongly Correlated with Test Suite Effectiveness. *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 435–445.

[TBD] Mutation-testing effectiveness literature (Andrews et al., Just et al.).

[TBD] Software testability metrics (Binder; Freedman).

Cross-language empirical software engineering.

[VERIFY] Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014). A Large-Scale Study of Programming Languages and Code Quality in GitHub. *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 155–165.

[TBD] Casalnuovo et al. on cross-language naturalness.

[TBD] GitHub-corpus methodology papers (Kalliamvakou et al. "The promises and perils of mining GitHub").

Static analysis tooling.

Brunsfeld, M. et al. tree-sitter: An incremental parsing system for programming tools. https://tree-sitter.github.io/tree-sitter/ (retrieved 2026-04).

Mutable state and pure functions.

[TBD] Backus (1977) "Can Programming Be Liberated from the von Neumann Style?" (Turing Award lecture; the classical argument for separating data from mutation).

[TBD] Hughes (1989) "Why Functional Programming Matters", for the composition-via-pure-functions argument.

[TBD] References from the Honest Framework specification, to be cross-referenced when that document is published.

Behavioural specification and testing.

[TBD] pytest-bdd or behave documentation, for Gherkin methodology.

[TBD] Hypothesis / QuickCheck property-based testing literature.

Industry data on AI-assisted development.

[VERIFY] GitClear (2025). AI Copilot Code Quality Report. https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_

This paper's replication package.

Wasserman, A. Z. (2026). Replication package for "Finite Testability of Enterprise Software." Zenodo. [DOI TO BE MINTED]

Open Honest research program.

[TBD] Preregistration: OSF Registry. [OSF ID + DOI once approval is finalised]

[TBD] Honest Framework specification (Wasserman, forthcoming).

Appendices

- **Appendix A.** Bug History of the L1.18 Analyzer (`appendices/appendix-a-bug-history.md`).
- **Appendix B.** Version Deltas (`appendices/appendix-b-version-deltas.md`).
- **Appendix C.** Behavioural Specification for the L1.18 Analyzer (`appendices/appendix-c-gherkin-specification.md`).
- **Appendix D.** Development Methodology (`appendices/appendix-d-development-methodology.md`).