

Research to Runtime Correctness Under Adversarial Numerics

MorphIQ Labs Research

Authored by Stephen Romano for MorphIQ Labs

August 2026

Version 0.2 — Draft for review

A research result and a production system are two different artifacts, and the distance between them is where correctness quietly fails. A pricing model is validated in a notebook and re-implemented in a fast language; a signal is proven on clean historical data and then run on a live feed; a strategy is backtested against one ordering of events and deployed into another. Each translation is a place where the mathematics that was checked stops being the mathematics that runs. In quantitative trading, that gap is a dominant source of model risk, and it is largely unmeasured.

This paper describes how MorphIQ Labs closes that gap. Its subject is not any single product but the *program*: a standard productized as gate scripts, validators, proof audits, and evidence ledgers that every engine we build inherits. It is written for the technical decision-maker who has to rely on systems like these, and for whom “rigorous” means nothing until someone shows the machinery. The thesis is deliberately narrow and falsifiable: correctness, integrity, and performance are properties of the *same* artifact, not trade-offs among them, and the disciplines that keep them together are engineering contracts we can show, not adjectives we assert: reference oracles, proof gates on stable laws, property-based and mutation testing, deterministic replay, bounded arithmetic, cross-implementation conformance, and reproducible release artifacts. We describe the standard, the machinery that makes it a program rather than a set of good intentions, and the evidence that the program is real across its instances: a Rust wavelet engine, a commercially licensed Java implementation of the same mathematics, an options-risk engine, and a deterministic replay substrate. As important, we state precisely what we do *not* claim. The companion whitepapers and repository evidence ledgers cited throughout are the receipts; this paper is the map to them.

Contents

1	The Seam	4
2	The Claim	4
3	The Standard	5
3.1	Correctness	5
3.2	Integrity	6
3.3	Performance	6
4	The Program	7
5	The Evidence	7
5.1	Reference oracles: FerroWave and FerroRisk	8
5.2	Cross-implementation conformance: VectorWave and FerroWave	8
5.3	Proof gates on stable laws: FerroReplay, FerroWave, VectorWave, FerroRisk	8
5.4	Mutation-tested transforms: FerroWave and VectorWave	9
5.5	Determinism as a contract: FerroReplay	9
5.6	Performance as a contract: FerroWave and VectorWave	9
5.7	Reproducible artifacts: FerroWave and VectorWave	10
5.8	Release gates: FerroRisk	10
6	What This Buys a User	10

7 The Questions This Standard Teaches You to Ask	11
8 What We Do Not Claim	11
9 Closing	12

1 The Seam

Most software-correctness effort is spent inside a single artifact: this function, given this input, returns this output. Quantitative systems fail somewhere else: at the *seam* between the research that established a result and the runtime that is supposed to preserve it.

The seam has a characteristic shape. A model is derived and checked in a research environment optimized for expressiveness: arbitrary precision, rich libraries, no latency budget, a fixed and friendly dataset. It is then re-expressed in a runtime environment optimized for the opposite: fixed-width arithmetic, a hard latency budget, adversarial and incomplete inputs, and event orderings no one chose. Nothing forces the second artifact to agree with the first. The tests that pass are usually the tests someone thought to write, over the inputs that occur most often, which is exactly the region where two implementations are *most* likely to agree and least likely to reveal a divergence.

The failures that result are not loud. They are a Greek that is correct at the money and wrong in the deep wings; an implied-volatility solver that converges to the wrong root in an admissible-but-rare region; a transform that is exact on the sample and biased on the tail; a backtest that is reproducible only because yesterday a thread won a race it will lose tomorrow. None of these throw. They produce a number, and the number is used to rank, hedge, size, or explain a decision. This is what *model risk* means operationally: not that the model is wrong on paper, but that the running system silently stopped being the model.

The industry's usual answer is process *around* the seam: model-validation groups, sign-off gates, reconciliation reports. Those are real controls, but they treat the seam as a handoff to be inspected. Our position is that the seam should not be a handoff at all.

2 The Claim

The claim is deliberately small enough to be wrong:

Correctness, integrity, and performance are properties of the same artifact. The rigor that validates the research is carried into the artifact that runs at speed, provably where the law is stable enough to specify. And the artifact, its evidence, and its provenance cannot silently diverge.

Each word in that claim is doing work. *Correctness* means the mathematics that runs is the mathematics that was validated. *Integrity* means the artifact, its evidence, and its provenance cannot silently diverge: a gate cannot pass by not running, a baseline cannot drift from its recorded history, and the package a user installs is the package that was gated. *Performance* means latency and allocation budgets are enforced with the same machinery as the other two.

Two things this claim is careful *not* to say. It does not say the artifact is “verified” in the sense that its outputs are guaranteed correct; only specific, stable primitive laws carry maintained proofs, and we are precise about which (§8). And it does not say any pillar was bought with another. The load-bearing engineering observation is that the three do not trade against one another when the disciplines are in place from the start rather than retrofitted: a reference oracle is a test-time dependency, not a runtime one; a proof gate runs in CI, not in the hot path; determinism is a structural property, not an overhead; and provenance is carried with the artifact, not a runtime cost.

The remainder of this paper is the evidence for that claim: the standard (§3), the machinery that productizes it (§4), the receipts (§5), what it buys a user (§6), the questions it teaches a reader to ask of any quantitative system (§7), and its limits (§8).

3 The Standard

We hold a layered standard. No single discipline is sufficient; each is chosen to catch a failure mode the others structurally cannot. And the standard is held as machinery, not as culture: each layer below exists as an executable, blocking check; §4 is about that machinery. This section describes the *shape* of the practice: what each layer is and why it exists. The mechanics live in the per-engine companion papers cited in §5. The disciplines are organized by the pillar each defends; the imbalance among them is honest, because correctness fails in the most ways and so takes the most layers.

3.1 Correctness

Reference oracles. A test that compares an implementation to itself proves only that it is self-consistent. Numerical kernels are instead checked against an *independent* source of truth computed at far higher precision (an arbitrary-precision oracle, or a canonical reference implementation) whose answer is trusted because it is derived differently and to more digits than the runtime path. This is the discipline that catches the deep-wing Greek and the wrong-root solver, because it evaluates agreement *where the fast path is under stress*, not where it is comfortable. The oracle is a test dependency, never the runtime path; the runtime stays fast.

Cross-implementation conformance. The same mathematics sometimes ships in more than one implementation. When it does, one implementation is declared the behavioral reference, and every other must prove agreement against fixtures generated from a pinned revision of that reference, with the provenance recorded and self-generated fixtures explicitly rejected as oracle evidence. Round-trip reconstruction is not accepted as proof of agreement, because a mutually wrong forward/inverse pair reconstructs perfectly; the inverse must also consume coefficients it did not produce.

Proof gates on stable laws. Where a property is a stable mathematical law (monotonicity, put-call parity, no-arbitrage bounds, the clock and interval laws of a deterministic scheduler), it is stated as a law and machine-checked in a proof assistant; where it is a required check, it gates the build. Proof gates do not replace tests; they cover the cases tests can only sample. They apply only where the law is stable enough to specify, which is a minority of any system and precisely the minority whose violation is catastrophic.

Property-based testing. For behavior that is lawful but not provable in closed form, we assert the *invariant* and let a generator attack it across a large, randomized input space, including the pathological corners a hand-written suite omits. Properties test the region between “specific example” and “formal proof.”

Mutation testing. A passing test suite says the tests pass; it does not say the tests would *notice* a defect. Mutation testing deliberately injects faults and measures how many the suite kills. It is the discipline that tests the tests, and it is why a high test count is not, by itself, a correctness claim we make.

Bounded arithmetic. Where correctness depends on it, we do not use floating point for value-bearing quantities; kernels that must use floating point carry known error envelopes rather than hope. The point is not purity. It is that rounding behavior is a specified property, not an accident of the hardware.

3.2 Integrity

Deterministic replay. Wall-clock time, iteration order, and thread races are untested inputs in most systems. We remove them as a class: the runtime has no direct access to time or nondeterministic ordering, a constraint enforced mechanically rather than by convention, so that a backtest and a production run can be the *same program* over the same event tape. Determinism is what makes every other guarantee reproducible; without it, a result you cannot re-run is a result you cannot trust.

Fail-closed gates. A gate that cannot tell “checked and passed” from “did not run” is not a gate. Every required check is built to fail on missing evidence, not only on bad evidence: an incomplete mutation report fails the build, a benchmark that disappears from the corpus fails the comparison, and a committed baseline is rejected unless its checksum, execution policy, and source-revision ancestry all validate.

Reproducible artifacts. The artifact a user installs must be the artifact that passed the gates. Release packages are built twice and required to be byte-identical; the published archive’s own test suite is executed from the archive; toolchains, CI actions, and container images are pinned by digest; and dependency advisories are a blocking scan, not a report.

3.3 Performance

Benchmark gates. Performance is a contract too. Perf-sensitive paths carry benchmarks whose regressions gate the build, because an artifact that is correct but misses its latency budget has failed the *same* claim as one that is fast but wrong. And because latency alone can hide a defect, per-operation allocation is gated on the same terms.

The layers compose into defense in depth: the oracle catches numerical divergence, conformance keeps sibling implementations from drifting, proofs cover the stable laws, properties attack the lawful-but-unproven middle, mutation testing validates that the whole suite has teeth, replay makes all of it reproducible, the benchmark gate keeps correctness from being bought with speed, fail-closed gating keeps missing evidence from passing as success, and reproducible packaging carries the guarantee into the artifact that ships.

4 The Program

A standard becomes a program when it stops depending on the people who first held it. The productization has four parts.

Gates as code. Every discipline in §3 exists as an executable, blocking check: a mutation gate, a same-host performance comparator, a proof audit, a packaging verifier. The gate logic is itself under test: the validators that enforce the floors carry their own unit suites, because a gate with a bug is a hole shaped exactly like the failures it was built to catch.

Evidence as a ledger, not a claim. Findings receive stable identifiers in an append-only register; the commit that fixes one must reference it, and closure is recorded with the measurement that proves it. Verification status is a machine-readable matrix, not a paragraph. Performance baselines are immutable records carrying their own checksums, host description, pinned execution policy, and source-revision ancestry, and a validator rejects any snapshot whose provenance does not check out.

One program, many stacks. The clearest evidence that the rigor is productized rather than heroic is that the same gate architecture is instantiated in two unrelated toolchains. The platform's wavelet mathematics ships as FerroWave, a Rust engine, and as VectorWave, a commercially licensed Java library: different build systems, different test frameworks, different profilers. Both carry the same shapes: a changed-code mutation floor that fails closed, a same-host latency-and-allocation gate, byte-reproducible packaging verified by building twice, a mirrored machine-checked proof tree, and provenance-validated baselines. The program is the invariant; the language is a parameter.

Inheritance, not adoption. A new engine does not negotiate its own standard. It starts inside the program, with the gate scripts, the audit tooling, and the ledger conventions already in place, and its correctness surface grows under the same contracts from the first commit. That is what taking research to runtime means as a productized capability rather than a one-time achievement.

5 The Evidence

A program is only a claim until it produces artifacts. The following are representative receipts from the program's instances, each checkable against a repository or a published paper. Each subsection stands alone; skim for the discipline you distrust most.

5.1 Reference oracles: FerroWave and FerroRisk

FerroWave anchors every numerical value in the library to a citable external authority and treats a failing reference test as a finding, not a tolerance to relax. The discipline earns its keep: it has surfaced real divergences, including long-latent ones that every self-consistency test had passed, because an external reference is a standard a buggy implementation cannot satisfy from the inside.¹ The definitional oracles have since been widened and hardened: transform families are cross-validated against independent NumPy/SciPy reference implementations with pinned versions, and the committed fixtures that carry those oracles are consumed by mandatory tests in the default gate, no longer opt-in runs. FerroRisk applies the same posture with an arbitrary-precision oracle: its Greeks (all ten, for both the Black–Scholes and Black-76 models), the normalized-Black price and vega, and the underlying Gaussian numerics are validated against a 512-bit MPFR reference, gated behind a test-only feature and run in CI.² Its implied-volatility solver is validated against a different independent authority, Peter Jäckel’s published “Let’s Be Rational” reference, rather than against MPFR, precisely because the right oracle for a solver is another trusted solver.³

5.2 Cross-implementation conformance: VectorWave and FerroWave

Because the wavelet mathematics ships in two implementations (§4), drift between them is a correctness risk in its own right. FerroWave is declared the behavioral reference, and VectorWave’s conformance program holds every production algorithm path to six gates before it may dispatch: reference coefficients from a pinned independent oracle; a foreign-inverse check, in which the Java inverse must reconstruct a pinned signal from coefficients it did not produce; property tests including adversarial values; path parity, in which every scalar, SIMD, batch, parallel, and streaming variant must agree with the verified scalar definition; formal conformance against the mirrored theorem set; and a performance gate. Forty-four production algorithm paths currently hold verified status, backed by a fixture corpus generated from pinned FerroWave revisions whose generator scripts abort on any revision mismatch.⁴ Cross-platform determinism is held to the same posture: the same computation is fingerprinted on three operating-system/architecture pairs, and the fingerprints must be identical for the build to pass.

5.3 Proof gates on stable laws: FerroReplay, FerroWave, VectorWave, FerroRisk

The proofs are written in Lean 4 and carry no sorry and no custom axioms; the trusted base is audited. FerroReplay carries machine-checked clock, wake, interval, and realignment laws under a *blocking* CI gate; stated precisely, these prove a specified model that mirrors the runtime surface, not the Rust code directly.⁵ FerroWave proves eighteen structural primitive-contract laws (index, length, and boundary logic) on the extracted Rust via Aeneas, and carries 268 theorems over $\mathbb{R}$ in Lean/mathlib: perfect reconstruction $\text{IDWT}(\text{DWT}(x)) = x$, Parseval energy preservation, filter orthonormality, and the laws of its newer decomposition and estimator surfaces. Every one of these now gates CI: the axiom audit mechanically derives the complete theorem inventory, pins

¹Reference-Based Numerical Validation in FerroWave, MorphIQ Labs Research.

²Pricing Models and companion FerroRisk notes, MorphIQ Labs Research.

³Implied-Volatility Solver, MorphIQ Labs Research.

⁴VectorWave conformance ledgers (parity program, findings register, and verification matrix), maintained in the repository.

⁵FerroReplay, Sequence-not-Timestamp, and Crash-Tolerant Replay, MorphIQ Labs Research.

its count, and fails if any declaration rests on sorry or a non-standard axiom.⁶ Its floating-point step bounds (Haar DWT ≤ 1 ulp, db2 ≤ 1.6 ulp, CWT frequency-domain complex multiply ≤ 1 ulp) are proven in Gappa and independently re-verified by the Coq kernel against the Flocq floating-point formalization, under a blocking audit job whose toolchain is pinned by digest.

VectorWave mirrors the same theorem inventory—the full set over $\mathbb{R}$ plus the structural laws—into its own audited proof tree, with the executable assumptions pinned by Java tests. FerroRisk states put–call parity, no-arbitrage price bounds, monotonicity in spot, convexity, and the Black–Scholes PDE as machine-checked Lean theorems over $\mathbb{R}$.⁷ The honest scope is stated in §8: the extraction proofs cover structural logic (Aeneas has no floating-point model), the $\mathbb{R}$ proofs cover the idealized mathematics, and the floating-point gap is bounded separately by kernel-checked Gappa certificates or measured in conformance tests.

5.4 Mutation-tested transforms: FerroWave and VectorWave

Both wavelet implementations enforce mutation floors as blocking gates, and both fail closed. FerroWave’s merge gate mutates every changed production line and requires a 75% kill rate; any timed-out mutant fails the gate regardless of the rate, an incomplete mutation report fails the gate, and an inventory floor rejects any change that silently shrinks the mutable surface. VectorWave holds changed production code to 75% mutation kill, 90% line coverage, and 75% test strength, with additional per-module floors, and rejects any run containing an uncovered, timed-out, or abnormally terminated mutant. The floors are honest in both directions: where a first whole-module measurement came in below the changed-code floor, the module’s floor was set at the measured value and documented as exactly that—a measured result, not a claim of parity with the narrower algorithm packages. A floor you cannot currently meet is a wish, not a gate. This is the discipline behind the claim that FerroWave’s more than 2,000 tests and VectorWave’s more than 7,900 tests have teeth, rather than the counts being the claim itself.⁸

5.5 Determinism as a contract: FerroReplay

The deterministic-replay constraint is enforced mechanically, not by reviewer discipline, through two independent, blocking gates: a Clippy disallowed-methods lint that bans direct wall-clock and timer APIs (`SystemTime::now`, `Instant::now`, the `tokio::time` family), and a separate textual CI check that greps the source for the same tokens (“belt and suspenders”). Exactly one sanctioned clock module is exempt, so a backtest and a production run are the same program over the same event tape.⁹

5.6 Performance as a contract: FerroWave and VectorWave

Both wavelet libraries gate performance on the same terms as correctness. A change is compared against its base on the same host, back to back: tail latency may not regress by more than 10% on any benchmark in the corpus; and because allocation churn is a defect that latency alone can hide,

⁶*Reference-Based Numerical Validation in FerroWave*, MorphIQ Labs Research, and the repository’s formal-verification documentation.

⁷*Put–Call Parity and the Forward and Pricing Models*, MorphIQ Labs Research.

⁸FerroWave mathematical-validation documentation and the VectorWave conformance ledgers; per-run figures and scope are recorded there.

⁹*Determinism as a Correctness Contract*, MorphIQ Labs Research.

per-operation allocation counts and bytes are gated at the same threshold. A benchmark that disappears from the corpus fails the comparison rather than escaping it, and VectorWave runs the gate on two host architectures. Committed baseline snapshots are treated as evidence, not thresholds: each records its checksum, host, pinned execution policy, and source revision, and a validator rejects any snapshot whose provenance, benchmark inventory, or revision ancestry does not check out.

5.7 Reproducible artifacts: FerroWave and VectorWave

A gate proves nothing about an artifact it never saw, so packaging is under the same contract. FerroWave’s release crate is packaged twice and the archives must be byte-identical; the published archive ships its test suite, and CI executes that suite *from the packaged archive*: evidence that the artifact a user downloads passes the same tests as the checkout it came from. VectorWave builds all nine of its distributables twice from clean and requires identical checksums, pins its build toolchain by checksum, and smoke-tests the packaged modules, including the obfuscated release JARs, through real module-path consumers on both supported JDKs. In both repositories, CI actions, container images, and analysis tools are pinned by digest or checksummed at install, and dependency advisories are a blocking scan, not a report.

5.8 Release gates: FerroRisk

FerroRisk’s CI enforces formatting, Clippy warnings-as-errors, the full regression suite (which runs the MPFR oracle tests), documentation warnings-as-errors, a fuzz build check plus fuzz smoke runs, SIMD correctness under aarch64 emulation, and a required Lean model-proof gate; performance thresholds are enforced as a release gate. A release cannot silently drop a guarantee because each of these is a gate, not a guideline.

Each item above is a claim a reader can check: an artifact exists, and it says what we say it says. That is the standard this paper is itself held to.

6 What This Buys a User

Rigor is not held for its own sake; it changes what a user can *rely on*.

- **Numbers you can stand behind.** A risk surface whose contracts are proven or oracle-checked can be *explained* (parity holds, monotonicity holds, the wings are specified) rather than merely produced.
- **Backtests that mean something.** When production is the same program as the backtest, a result that reproduces is evidence of determinism, not of profit, and not coincidence.
- **Failure where it is visible.** Bounded arithmetic and stated error envelopes turn silent numerical drift into a specified, inspectable property.
- **The artifact you install is the artifact that passed.** Byte-reproducible packaging and packaged-artifact testing close the last translation—the one between the repository that was gated and the artifact that ships.

- **Diligence that moves fast.** The receipts are artifacts, not interviews. A counterparty evaluating the work can check a gate, a ledger, or a proof audit directly, rather than sampling our confidence.
- **A standard that does not decay.** Gates run on every commit, and a new engine inherits the program rather than negotiating its own standard. The correctness is a maintained contract, not a launch-day snapshot.

7 The Questions This Standard Teaches You to Ask

The fastest way to evaluate a claim of rigor, ours included, is not to read the claim. It is to ask what the machinery does when something goes wrong. Six questions transfer to any quantitative system:

1. **Is the oracle independent?** If the reference values were generated by the implementation they test, the tests prove self-consistency and nothing else.
2. **What happens when a required check cannot run?** The only honest answer is “the build fails.” Anything softer means missing evidence can pass as success.
3. **Do the proofs block anything?** A theorem that gates merges is an engineering control. A theorem in a slide deck is marketing.
4. **Would the tests notice?** Ask for the mutation figures, not the test count. A suite that has never been attacked is a suite of unknown strength.
5. **Is the shipped artifact the tested artifact?** Ask how the package a user installs is tied, byte for byte, to the build that passed the gates.
6. **Can you replay it?** Ask whether last month’s run reproduces exactly, with the same inputs, ordering, and outputs, and what mechanically enforces that.

None of these questions require access to the code; they require only that the answers be specific. That is what a program buys: the answers stop depending on who you ask.

8 What We Do Not Claim

A paper about rigor earns trust in proportion to what it refuses to overstate. The limits are therefore part of the standard, not a disclaimer appended to it.

- We do **not** call any product, platform, application, or trading strategy “formally verified.” The claim is specific, stable *primitive laws* carrying maintained proofs: never investment outcomes, model certainty, execution quality, or risk-free trading.
- A proof boundary covers the law it specifies and nothing outside it. Where the underlying law is not stable enough to specify, we say so, and rely on oracles, properties, and mutation testing instead.

- Machine-checked proofs prove a model. The formal surfaces (structural contracts, deterministic-clock laws, and the mathematical-law inventories over $\mathbb{R}$: reconstruction, Parseval, put-call parity, the Black–Scholes PDE) now gate CI as blocking checks. But the $\mathbb{R}$ proofs cover the idealized mathematics; the Rust-extraction proofs (Aeneas) cover structural and index logic, not floating-point values; and the floating-point gap is bounded separately, by Gappa with Coq-kernel-verified certificates for specified kernels and by measured conformance elsewhere. We do not claim the compiled floating-point binary is end-to-end proven.
- Cross-implementation agreement is inherited evidence, not independent truth. VectorWave’s parity to FerroWave means what it means because the reference is itself anchored to external authorities; the parity gates transfer that anchor, they do not replace it.
- Evidence is verifiable but bounded: a caught-bug narrative is one data point, a mutation-kill rate is a property of a suite over a scope at a point in time, a benchmark is a measurement under stated conditions. We publish the conditions, and the enforced mutation contract is the changed-code floor held on every merge; we no longer quote a historical whole-repository score as if it were one.
- Rigor reduces model risk; it does not eliminate it. The residual is the part we are honest about, because a rigor practice that oversells a single claim spends more credibility than any one result can return.

9 Closing

The seam between research and runtime is where quantitative systems actually fail, and it is usually managed by inspecting a handoff. We do not treat it as a handoff. The same rigor that validates the research is carried into the artifact that runs: oracle-checked, proof-gated where the law is stable, mutation-tested, deterministic, benchmarked, and packaged reproducibly. The mathematics that ships is the mathematics that was proven. And because that rigor is a program rather than a project, the next engine starts with the same guarantee. The receipts are in the repositories and in the companion whitepapers, and they are the standard this paper holds itself to.

Companion Whitepapers

The following are the primary evidentiary sources for this thesis. All are MorphIQ Labs Research whitepapers unless noted.

- *Reference-Based Numerical Validation in FerroWave*: the reference oracle and test-triangle methodology, with case studies.
- *Implied-Volatility Solver; Pricing Models; Volatility Surface; Put–Call Parity and the Forward*: FerroRisk numerical contracts and oracle validation.
- *FerroReplay; Sequence-not-Timestamp; Crash-Tolerant Replay; Determinism as a Correctness Contract*: deterministic replay primitives and their proven laws.
- *Portable SIMD Numerical Kernels*: performance under the same correctness contract (note: NDA-tier; cite existence, not internals).

- VectorWave conformance and assurance ledgers: repository-maintained evidence (the parity program, findings register, verification matrix, and baseline provenance records) rather than a standalone whitepaper.

Colophon

Version 0.2 — draft for review. August 2026.

Revision history: 0.1 (July 2026), initial draft. 0.2 (August 2026), reframed around the productized program (§4) and elevated integrity to a first-class pillar of the claim; restructured for the technical decision-maker, with the standard organized by pillar and a section of transferable diligence questions (§7); evidence re-verified against the current assurance programs; added cross-implementation conformance, fail-closed gating, and reproducible artifacts to the standard; formal-proof surfaces are now blocking gates throughout, and the figures in §5 were updated accordingly.

© 2026 Prophetizo LLC. All rights reserved. MorphIQ Labs, the Ferro suite, and VectorWave are trade names and product brands of Prophetizo LLC.

Third-party names, including Lean, mathlib, Aeneas, Gappa, Coq, Flocq, MPFR, Clippy, tokio, NumPy, SciPy, PIT, JMH, and “Let’s Be Rational”, are trademarks or marks of their respective owners and are used here for identification only; their use does not imply endorsement.

This paper describes engineering correctness practices. It is not investment advice, not an offer or solicitation, and makes no representation of trading performance or outcomes.