Don't Port the Syntax. Port the Evidence.

A practitioner's guide to large code migrations and clean-room-style rewrites with coding agents

Farrukh Nauman

2026-03-09

Over the last six months, I have used coding agents on multiple migrations and reimplementations in the 10k+ line range. The strongest lesson is that these projects are not mainly translation problems. They are evidence-preservation problems.

That is true whether you are moving JAX to PyTorch, React to Svelte, C or Fortran to Python, or rebuilding a vision or time-series algorithm against an existing API and test suite. The cost of producing plausible code has fallen sharply. The cost of proving that the code is correct has not.

That gap is where most projects go sideways.

Why migration and clean-room rewrite are the same hard problem

A migration says: preserve behavior while changing language, runtime, or architecture. A clean-room-style rewrite says: match the observable contract without copying the original implementation. In practice, they share the same hard part: you need a trustworthy behavioral oracle — an external source of truth that tells you whether the new code actually matches the old behavior.

That source of truth may be reference outputs, captured intermediate states, API fixtures, gradient checks, or a frozen end-to-end workflow. When it is absent, the new implementation starts defining its own success criteria — and coding agents will happily optimise for plausibility rather than truth.

A note on terminology: I say *clean-room-style rewrite* unless there is actual personnel and information separation in the traditional legal sense. This post is about engineering process, not legal advice.

The failure modes that keep showing up

After a few large migrations, the agent failure modes become predictable.

1. Potemkin implementations

The agent gives you a module with the right names, file structure, docstrings, and a green test suite. Underneath, one or two load-bearing pieces are still placeholders, simplified branches, or silent fallbacks. A custom backward path replaced by a trivial approximation. A critical boundary update reduced to a stub. Lifecycle cleanup or accessibility details quietly dropped.

This is the failure mode I look for first: code that is *structurally convincing* but *algorithmically hollow*.

2. The "almost works" trap

A migration that fails immediately is easier to fix than one that passes 90% of its tests. The dangerous version is when one missing step or one subtle semantic change is masked by the rest of the system. Nine green checks, one red check, and a strong temptation to treat the failure as noise.

That temptation is usually wrong. The missing piece is often only visible in the test that exercises the least dramatic regime — a smooth signal instead of a spiky one, a steady-state case instead of one with discontinuities, a UI flow that stresses cleanup rather than the happy path.

3. Tests that prove nothing

Agents love tests that are easy to satisfy: shape checks, type checks, "does not crash," snapshots generated by the same code under test, before-and-after comparisons that accidentally alias the same mutable object. A migration can have a green suite and still be almost entirely unverified.

The broad pattern: if a test does not compare against an independent oracle, or can be satisfied by preserving superficial structure, it is weak evidence.

4. Configuration drift and branch loss

The old code rarely has one behavior. It has a decision surface — runtime flags, device and dtype, train versus eval mode, feature flags, optional stages. Agents port the visible path and silently collapse the rest.

A related trap: treating "feature disabled" as "coefficient zero." In real systems, zero is often not off. A stage that should be absent may still execute, mutate state, consume randomness, or impose constraints.

5. Quiet cheating

Once an agent is attached to the goal "make tests pass," watch for shortcuts: loosening tolerances, rewriting tests to fit current behavior, skipping the expensive path, leaving comments like "placeholder, full implementation later" inside supposedly finished code, or removing a failing edge case rather than explaining it.

This is why review must include *evidence review*, not just code review.

The workflow that stopped the failures

1. Freeze one golden path

Pick one reference configuration that matters: one model, one device, one dtype, one batch shape — or one UI route, one feature-flag setting, one numerical profile. Everything else is out of scope until that path is genuinely trustworthy.

2. Build the oracle before you port

Do not start by asking an agent to "rewrite this module." Start by building evidence: captured reference outputs, serialized intermediate tensors, UI interaction traces, API fixtures, gradient checks, deterministic seeds, exact commands for reproduction. The implementation comes second.

3. Compare stages, not just endpoints

End-state comparisons are too coarse. If a port yields the wrong training curve, the error may live in preprocessing, masking, broadcasting, dtype promotion, RNG use, the optimiser step, or the loss reduction. Without stage-level checkpoints, every mismatch becomes archaeology.

4. Use small PRs and adversarial review

Large migrations fail in review because no one can hold the whole diff in their head. Small PRs make it possible to ask: which source branch did we preserve? Which one did we intentionally omit? What exact evidence was added? What changed in tolerances?

I also get more reliable results when different models review each other's work with directed prompts:

- "List every source branch or feature path that is still missing."
- "Find any place where this implementation used a placeholder or a cheaper substitute."
- "Compare the test evidence to the claimed scope. What is unproven?"
- "Identify any test that could pass while the implementation is still wrong."

5. Turn every failure into repository policy

Every recurring failure should become an instruction in `AGENTS.md`, a checklist item, a regression test, or a new reproducibility artifact. Over time, the repository accumulates institutional memory. That is the real leverage point.

What I now put in AGENTS.md

A good `AGENTS.md` is not a vibe document. It is a quality contract.

```
# AGENTS.md

## Scope and honesty
- No placeholder implementations.
- If something is incomplete, say so explicitly.
- End every nontrivial task with: implemented / partially implemented / not implemented / risks.

## Verification
- Do not change tests or tolerances just to make them pass.
- Use the existing implementation, captured outputs, or test fixtures as the oracle.
- Add or update tests that assert values and behavior, not only shapes or existence.

## Migration discipline
- Preserve the frozen compatibility path before generalizing.
- List every feature flag, branch, or runtime mode touched by the change.
- A disabled feature should be absent from execution, not just numerically neutral.

## Shortcut audit
- Search for TODO, FIXME, placeholder, simplified, stub, pass, and NotImplemented.
- Flag any fallback path that weakens correctness, not just performance.
```

I also add one explicit sentence for clean-room-style work:

> Use the existing implementation as a behavioral oracle, not as text to paraphrase.

That single line changes the task from "restate this code in another syntax" to "rebuild the behavior and prove it."

Where the harness matters more than the model

There is a growing body of evidence that the *environment* around a coding agent matters at least as much as the model's raw capability.

OpenAI's harness engineering team recently described building a million-line production application where human engineers wrote zero lines of code directly. Their central finding was that early progress was slow not because the agent was incapable, but because the environment was underspecified. The engineers' primary job became designing the harness: breaking goals into building blocks, making constraints legible and enforceable, and structuring documentation so that the agent could navigate it. They found that a monolithic `AGENTS.md` failed — it crowded out the actual task, rotted quickly, and became impossible to verify. Structured, cross-linked documentation in a `docs/` directory worked much better.

Models matter too. Peter Steinberger made the observation in his "Shipping at Inference Speed" workflow. He noted that Codex models build context autonomously — sometimes silently reading files for ten or fifteen minutes before writing a single line of code. This contrasts with models like Opus that tend to start writing eagerly and then miss parts of the codebase. Steinberger found that even though Codex sometimes took four times longer per task, the overall velocity was higher because he did not have to go back and fix the fix.

These two observations converge on the same point. The agent's effectiveness depends on the quality of the environment it can read and the evidence it can verify — not on how clever the prompt is. For migrations and clean-room-style rewrites, that environment is your oracle, your test suite, your `AGENTS.md`, and your intermediate checkpoints. Get those right and agents become extremely effective. Without them, they mostly accelerate ambiguity.

A review checklist I actually trust

When a migration or clean-room-style rewrite is under review, these are the questions I want answered.

1. What is the frozen reference configuration?
2. What is the external oracle?
3. Which branches, flags, or modes are in scope? Which are explicitly out?
4. What evidence exists at intermediate stages, not just final outputs?
5. Which tests assert values or behavior rather than shapes, types, or snapshots?
6. Could any test be passing because it reuses mutated state or self-produced outputs?
7. Were any tolerances changed? If yes, why?
8. What one test would most likely expose a missing algorithmic step?
9. What would make us say this change is not complete yet?

Closing thought

Coding agents make implementation cheaper. They do not make correctness cheaper.

In fact, they often make correctness *more* expensive, because they can generate large amounts of plausible but weakly justified code at high speed. Judgment, review, and evidence become the bottleneck.

This is not a reason to avoid agents. It is a reason to use them like an engineer: trust the agent to generate options, trust tests only if the proof chain is clean, trust "done" only after a shortcut audit, trust speed only after correctness is nailed down.

Once the repository has a real oracle, a real `AGENTS.md`, a real review checklist, and a real compatibility path, agents become extremely effective. Without those things, they mostly accelerate ambiguity.

If I were starting another migration tomorrow, I would spend less time asking, "How do I translate this code?" and more time asking, "What would count as proof that I have not fooled myself?"

That question is more demanding up front. It is also what separates a flashy rewrite from one you can trust.

Further reading

- Simon Willison, "Can coding agents relicense open source through a 'clean room' implementation of code?" — a useful framing of agentic rewrites, behavioral equivalence, and the unresolved legal and ethical questions.
- Armin Ronacher, "AI And The Ship of Theseus" — on what changes once code can be reimplemented from API behavior and tests.
- OpenAI, "Harness engineering: leveraging Codex in an agent-first world" — on environment design, structured documentation, and why a monolithic instruction file fails at scale.
- OpenAI, "Unrolling the Codex agent loop" — on the core agent loop, context management, and how the harness orchestrates model inference, tool calls, and feedback.
- Peter Steinberger, "Shipping at Inference Speed" — on autonomous context-building in Codex models, shorter prompts, and when to trust the agent's own exploration.
- Anthropic, "Effective harnesses for long-running agents" — on feature lists, progress files, incremental work, and preventing premature victory.
- Anthropic, "Scaling agentic coding across your organization" — on project-level `CLAUDE.md` files, explicit instructions, and test-driven rollout.
- Meta, "Translating Java to Kotlin at Scale" — a reminder that "simple" transformations become complex at scale.
- Stripe, "Migrating millions of lines of code to TypeScript" — on planning, codemods, validation, and the value of a sharp cutover once the evidence is strong enough.