



AI Test Code & the Last Mile

Why faster code generation does not shorten physical validation

Eric Beaty

Dr Volt Inc

AI can produce an initial version of test software in hours, but engineering still has to prove that the system measures the right thing, responds safely, and produces decisions the organization can trust.

AI Has Changed the Start of the Project

AI can now translate a test description into instrument communications, data logging, calculations, limit checks, operator prompts, and a functional user interface with remarkable speed. In many cases, the result will run and may even perform most of the intended procedure correctly on the first attempt.

That is a meaningful change. The first draft used to consume a large share of the schedule. Now it can appear before the hardware is fully assembled.

But the purpose of test software is not to generate code quickly. It is to create measurements and decisions that engineering, manufacturing, and quality teams can rely on.

The First Version Is No Longer the Hard Part

A test application can look complete while containing a wrong channel assignment, an incorrect scale factor, a timeout shorter than the instrument settling time, a sign error in a limit comparison, or an abort path that records a fault without putting the hardware into a safe state.

Human developers make these mistakes too. What AI changes is the speed and scale at which plausible implementations can be created. Code that is clean, organized, and syntactically correct can now arrive faster than the team can build confidence in its behavior.

AI has not removed the last mile of test development.

It moved more of the project into it.



The Last Mile Is Where Trust Is Earned

The last mile begins after the application appears to work. It includes checking the intended procedure against the implemented logic, confirming instrument behavior, challenging assumptions, validating failure handling, and exercising the complete system with real hardware.

This work is slower because it depends on evidence. A battery may need to charge, discharge, balance, heat, cool, or rest. A fixture may need to be rewired. A deliberate fault may need to be created. The engineer must observe what the system actually does, not what the code appears to say it will do.

AI can make that work more focused. It cannot make the evidence optional.

Physical Verification Still Sets the Pace

Software can often be exercised through automated test suites, simulated environments, and thousands of parallel cases. Physical test systems benefit from those same practices, along with mocked instruments, recorded-data replay, and hardware-in-the-loop methods.

Those tools are valuable, but they cannot reproduce every interaction among programmable supplies, loads, meters, relays, fixtures, communication networks, safety circuits, thermal equipment, and the unit under test.

Real hardware introduces behaviors that are easy to omit from a prompt, model, or simulation:

- startup and settling delays that vary with operating state
- wiring errors, noise, ground paths, and sensor scaling
- instrument firmware differences and communication timing
- contactors, interlocks, and safe-state behavior during faults
- operator actions, fixture wear, and interrupted test recovery

The practical goal is not to eliminate physical validation. It is to use AI and automation to remove avoidable mistakes before the run, capture better evidence during the run, and shorten the path from a failure to a focused correction.



The Bottleneck Has Moved

AI has dramatically reduced the effort required to produce a first version of test software.

The bottleneck is increasingly the engineering work that follows generation: review, integration, fault handling, physical verification, evidence capture, and release control. That work spans software, application engineering, driver integration, and troubleshooting.

Use AI to accelerate what is new.

Preserve what is already trusted and make every physical validation cycle count.

The Most Dangerous Failure May Look Like Success

A crashed application is visible. A test that completes normally while reading the wrong channel, applying the wrong limit, or missing an unsafe condition may be more dangerous—and potentially more damaging—than one that crashes.

A passing run proves only that the application completed and satisfied the implemented criteria. It does not prove that the measurement, decision, or safety response is correct.

Why Software-Only Tests Are Not Enough

Unit tests can confirm calculations and state transitions. Simulated instruments can exercise command handling. Replay can reproduce a known failure without consuming another hardware cycle. Static analysis can catch obvious defects.

Each method reduces risk, but each method is bounded by its model. A simulator cannot reveal behavior its model does not represent—for example, a real device returning data in a different format or a timeout during an actual mode transition. **A generated test can repeat the same assumption that produced the generated code.**

Make Every Hardware Cycle Count

Before the next run, use the available evidence to narrow the question. Review the procedure, inspect changed logic, compare configuration, check the expected timeline, and define what the run must prove. Afterward, preserve logs, measurements, operator actions, and configuration so the result can be reconstructed.

The best use of AI is not to replace physical validation.

It is to make each physical validation cycle more informative.



Human Engineers Own the Result

AI can draft interfaces, sequence logic, calculations, limits, tests, and diagnostic hypotheses, shortening the path from a description or failure to a reviewable change.

It cannot assume responsibility for measurement validity, safe hardware behavior, appropriate limits, regulatory requirements, or the release decision. Those remain engineering responsibilities.

The Hidden Cost of Rebuilding the Test Environment

When AI is asked to generate a completely custom application for every new test stand, it creates far more than the procedure. It must also implement execution control, communications, logging, variable management, error handling, reporting, operator interaction, safety behavior, configuration, and recovery from interrupted states.

AI can make this repeated construction look inexpensive because the code appears quickly. But generation was never the only cost. **Every recreated layer expands the portion of the system that must be understood, reviewed, debugged, maintained, and verified.**

The better engineering question is not *"Can AI write the complete application?"* but *"Which parts genuinely need to be new?"*

KEEP STABLE	ADAPT FOR THE APPLICATION
Execution and state control Logging and data handling Error and abort behavior Operator interaction Versioned interfaces	Test procedure Product-specific limits Channel configuration Calculations and decisions Required report content

This separation can be implemented in a commercial sequencer, a reusable internal framework, a validated library, or a carefully managed custom architecture. The principle is software agnostic: preserve trusted infrastructure, concentrate new engineering where the application is actually different, and focus verification where it creates lasting value.





Three Uses of AI That Create Practical Value

AI is most useful when it accelerates a bounded engineering task and leaves a clear review target. AI should expose uncertainty, not silently resolve it. The following examples apply across test platforms and programming environments.

1. Build an Instrument Interface from a Programmer's Guide

Situation: A new programmable instrument must be added to a test stand. The team has the manufacturer's manual, but no existing driver or command library.

AI contribution: Extract communication settings, command syntax, response formats, initialization requirements, status queries, error behavior, and supported operating modes. Generate a structured first draft and a verification checklist.

Engineering check: Confirm every command that can change energy, load, switching, or operating state on the physical device. Verify scaling, timing, error responses, and safe behavior.

The work shifts from transcription to verification. **A generated interface remains unverified until it is proven on the actual instrument.**

2. Find Bad Logic in Run Evidence

Situation: A test fails after a corrective action. The measurements look reasonable, but the final decision is still wrong.

AI contribution: Review the runtime log, executed steps, variables, and timestamps together. Trace which value was used, where it was created, and whether the application failed to refresh it after the system state changed.

Engineering check: Confirm that the proposed correction matches the intended method, then design the next run to prove that one change.

3. Translate Source Requirements into Visible Limits

Situation: A data sheet or internal specification defines voltage, current, temperature, timing, or fault limits that have not yet been translated into the application.

AI contribution: Extract stated values, distinguish recommended operating limits from absolute ratings, calculate derived thresholds, identify ambiguities, and draft traceable limit logic.

Engineering check: Verify scaling, units, configuration, hardware ratings, sensor behavior, selected margins, and any more conservative organizational or regulatory requirement.



Engineering Control Must Remain Visible

Fast generation becomes useful engineering when ownership, evidence, and the boundaries of change remain visible. A disciplined AI-assisted workflow should answer:

Source: What requirement, procedure, manual, or prior result supports the change?

Scope: Which files, sequence steps, limits, channels, or interfaces changed?

Assumptions: What did the model infer because the source information was incomplete?

Risk: What could cause a measurement error, unsafe state, or incorrect pass/fail decision?

Evidence: What simulation, bench check, hardware run, or independent calculation will verify it?

Approval: Who accepts the remaining risk and authorizes release?

Traceability Beats Confidence

A confident explanation is not evidence. The useful output is a visible chain from requirement to implementation to verification result. That chain allows another engineer to review the work, reproduce the decision, and identify what must be retested after a change.

Generated code, tests, and documentation can all help build that chain. They should not be treated as independent confirmation simply because they agree with one another.





Tests Must Be Independent of the Implementation

If the same AI workflow creates the interpretation of the requirements, the implementation, and the tests, all three can preserve the same mistaken assumption. The test may pass because it confirms what the code does rather than what the system was supposed to do.

Critical checks should be derived from approved requirements, known-good data, independent calculations, or deliberately introduced fault conditions. The question is not only whether the tests pass. It is whether they will fail when the defect the team is trying to prevent is present.

Protect Safety Behavior

Software limits are one layer of system safety. Hardware ratings, contactors, interlocks, emergency stops, watchdogs, sensor plausibility checks, and de-energized states must be considered together as part of a complete safety architecture.

AI may propose safety logic, but software logic should never be mistaken for the complete safety architecture. Engineering must verify the entire safety response on the real system.

Closing the Last Mile

AI changes how quickly software can be created, but not the validation required to prove that the system behaves correctly.

Reuse and standardization can narrow the verification surface and prevent unnecessary reinvention. Purpose-built software remains appropriate when the application genuinely requires it. The strongest designs make that distinction deliberately—and keep the resulting verification burden visible.

Dr Volt helps engineering teams make those decisions and turn them into working test systems through architecture, integration, troubleshooting, physical validation, and the evidence needed to trust the result.