A Great Assistant, and a Poor Master

Why AI-Generated Code Requires Architectural Judgment It Cannot Provide

Glenn L. Austin, AustinSoft.com

Glenn Austin is an independent software and security architect, drawing on over 30 years of software architecture across Apple, Adobe, Symantec, and Atlassian — holding US Patent 10,972,253, and operating through AustinSoft in Mount Juliet, Tennessee. The arguments in this paper were developed independent of any particular AI tool as part of AustinSoft's consulting practice; the author has no affiliation with or financial interest in any AI organization.

tl;dr

AI coding tools are a genuine force multiplier. They are also routinely trusted past the point where they are safe. This paper argues a specific, critical distinction: AI-generated code fails in a *characteristic* way: it is locally plausible and globally wrong, and the standard safety nets that teams rely on to catch bad code do not catch this class of failure. The paper separates two kinds of disagreement with an AI code-generation tool: *implementation* errors, which have a right answer the AI got wrong; and *architectural* disagreements, in which the AI makes a conventional, defensible choice that nonetheless violates a superior principle. One example is a choice that quietly instituted two competing sources of truth where there should be one, with the second mechanically derived from the first. In both categories, the same safety nets fall short: tests verify behavior, not architecture; a structured framework enforces shape, not soundness. The result is architecturally broken code that compiles, fits the framework, and passes every test — which is precisely why it reaches production. The conclusion is not that these tools are bad. It is that they are assistants, not masters: they multiply the judgment of the engineer wielding them, and are dangerous in the absence of that judgment. Deep engineering expertise becomes *more* essential in this era, not less, because it is the one layer that cannot be automated and the only layer that catches these errors.

The Frame: Fire

There is an old saying: fire makes a great servant but a poor master. The wisdom in that saying is not that fire is dangerous and should be avoided. This wisdom is that fire is *powerful* — useful, even essential — and its power is exactly what makes it destructive when it is not controlled. You keep it in a hearth, a firepit, a

burn barrel, or a furnace — some space built to contain it. You respect it. You do *not* trust it to run itself.

AI coding tools deserve the same level of respect. They are powerful and genuinely useful. And for the same reason, they are hazardous when handed authority they are not equipped to hold. This paper is about *where* that line sits, and *why* it sits there — not as an opinion, but as an engineering reality.

The Characteristic Failure: Locally Plausible, Globally Wrong

The failure mode that matters is not the obvious one. AI-generated code that does not compile, or that visibly does the wrong thing, is caught immediately and costs nothing. The dangerous failure is the code that *looks right*: it compiles, it reads reasonably, it runs correctly in the common case, it passes all of the tests, and it is architecturally wrong in a way that only becomes visible when you understand the system it was dropped into.

The reason is structural, not incidental. An AI model optimizes the unit of work in front of it. It produces code that is *locally* plausible — correct with respect to the function it was asked to write. What it does not have is the whole system's invariants and intent. It cannot see *why* the surrounding code is shaped the way it is, so it will confidently violate a design constraint it has no ability to perceive.

The remainder of this section presents four representative examples of AI-generated code, drawn from review of a persistence layer under active development, during a hands-on migration of its underlying storage engine. Each one compiled. Each one looked reasonable. Each one even worked in basic usage. However, each one was wrong at the architectural level.

The Fallback That Contradicted the Design

The storage layer used a deliberate *deferred-error* pattern — itself AI-generated, and correct. If the database failed to open, initialization did not crash; it returned a non-functional sentinel object and stored the error, so the failure could be surfaced gracefully at a controlled point during application startup — an error the user could see and act on, rather than a hard crash deep in the launch sequence. This was good code. The AI had reproduced a sound pattern, likely because sound examples of it exist in its training data.

Then the same AI added a "fallback" that defeated it.

The AI-generated code's "fallback" was to open the database a second time, directly and unconditionally. This did two things, both wrong. First, it would crash outright on exactly the failure the AI's own deferred-error code had been carefully built to handle gracefully, defeating the entire design. Second, it silently reintroduced a separate double-open defect (see *The Duplicate Connection*) that had just been designed against. It was annotated with a comment: *"should not happen in production."*

That comment is the signature of the failure. "Should not happen" means the case was not reasoned about. The AI-generated code papered over an unconsidered path with a plausible-sounding rationalization — and that is exactly where the defect lived.

The Orphaned Abstraction

A dependency-injected service and its protocol sat in the codebase, registered and conformed to, apparently required. In fact its original consumer — a since-removed synchronization feature — was gone, and much of the layer was stranded scaffolding whose reason for existing had been deleted. AI-generated architecture has a strong tendency to produce exactly this: plausible-looking structure, layers wrapping layers, that pattern-matches to "sophisticated architecture" without asking whether *this* system needs *this* structure, a simpler solution might be better, or whether the code was necessary at all. Distinguishing the live path from the dead one — and safely removing the dead one — required tracing real usage through the whole system. It was a question of judgment ("does this abstraction earn its place?"), not one any local check could answer.

The Duplicate Connection

A service opened the application's database in a separate location within the code and a second time, independently, rather than using the shared connection the rest of the system was built around. Two connections to the same local database invite a class of intermittent, timing-dependent failures: lock contention, stale reads across the two connections' separate views, split caches, and duplicated startup migration. None of these reproduce reliably; all of them are miserable to diagnose. The generated code was individually reasonable, "open the database I need," and globally corrosive, because it did not hold the system's invariant that one shared connection is the single source of truth — an invariant that had been the explicit guiding principle of the very refactor this code belonged to. The AI had been *given*

the rule, as the stated goal, and violated it anyway. A principle the AI was explicitly told to follow is not a principle the AI held; it will honor the rule where the rule is locally salient and break it where it is not, because it does not carry the invariant as a constraint on every decision.

One Truth, Split and Scattered

The AI organized the persistence code the way it organizes most things: by kind. All the Swift model structures went into one file; all the SQL schema and migrations went into another. Within each file, like sat with like. It looks tidy, and it is the kind of layout a reviewer skims past without alarm.

It is organized on the wrong axis. The unit that matters here is not "Swift" or "SQL" — it is the *model*. A model's structure and its schema are one thing expressed in two forms; they belong together, and they must never disagree. The AI's layout guarantees they *can* disagree. It split each model's definition across two files: the Swift structures declaring the format of the data in use by the code, and the SQL declaring the shape of the store, both authored by hand, with nothing forcing them to agree. Then those definitions of the model's two halves, scattered into two monoliths, sorted by artifact type, are as far apart as the codebase allows. The result is two competing sources of truth for every model — maximally separated, buried in two large files organized by the wrong thing, and kept in agreement only by a human remembering to find and change both, every time, forever.

Unlike the preceding examples, this is not a defect that will crash or corrupt on sight. It compiles, it runs, and it is a layout many teams use without incident — because in normal operation the application reads from the in-memory structures it already holds, not from the store, since going back to the store is the more expensive path. The two can therefore drift for a long time with everything *appearing* correct. The disagreement surfaces only when something finally forces a read from the store, at which point data that seemed saved comes back wrong, or maddeningly fails to appear at all. That invisibility is exactly what makes it the harder case — and it is taken up in full in *Two Kinds of Disagreement*.

The Common Thread

In every case, the generated code was locally reasonable and globally wrong. Three were outright defects; the fourth was a defensible-looking layout carrying the same flaw. In none of them was the problem *in the function itself* — it was in the code's

relationship to the system: its invariants, its intent, its history, its proper shape. Catching each required holding the whole architecture in mind and asking whether the code respected it. That capacity is what we mean by architecture, and it is precisely what the tool does not have.

Two Kinds of Disagreement

It would be a mistake — and a weaker argument — to claim that everything an AI produces is a defect. The examples in *The Characteristic Failure…* are generally *implementation* disagreements: cases with a right answer, where the generated code got it wrong. But there is a second, subtler category, and distinguishing the two is itself an act of the judgment this paper is about.

An *architectural* disagreement is not a case of right versus wrong. It is a case where the AI makes a *legitimate, conventional, defensible* choice that nonetheless conflicts with a superior principle. The code will compile and run. A reviewer skimming it will see nothing alarming. Reasonable engineers organize systems this way every day. And it is still the wrong choice — not because it fails, but because it institutes a latent structural hazard that only a held architectural principle would flag.

A Worked Example: Where the Schema Lives

Consider a migration of a persistence layer from one storage engine to another. The mechanical work — translating calls, rewiring types — is exactly the kind of task an AI generally does well, and one could reasonably hand it over. But the AI-generated approach organized the code in a conventional way: the SQL table definitions, the persistence logic, and the migrations lived in a *separate* file from the Swift structures they described.

This is a common layout. It is not a bug. And it is precisely the kind of choice that should make an architect deeply uneasy — because it creates *two sources of truth*.

The Swift structure declares what the model is. The SQL declares what the model is. Both are authored by hand; both claim authority; and because they are separate and independently editable, they can *disagree*. Nothing structurally forces them to remain in agreement, and nothing in the generated code links the two. Correctness now depends on a human remembering to change three things — the structure, its table definition, its migration — in lockstep, every time, forever. Two independent

sources of truth is functionally *zero* sources of truth: when they drift, you cannot tell which one is right.

This is the same fragility the disciplined architect designs *out* of every system: correctness that must be *maintained by vigilance* rather than *guaranteed by structure*. The goal is always to make the bad state — here, the Swift and the SQL disagreeing — *unrepresentable*, not merely defended against.

The Principle: Single Source of Truth, Mechanically Derived

The resolution is not tidier file organization. It is a categorical change in the relationship between the two representations.

Let the Swift definition be *the* truth — singular. Then the SQL is not a second authored artifact that must be *kept in agreement*; it is *derived* — it falls out of the Swift definition by a fully mechanical transformation. The relationship is that of a compiler or transpiler to its source: the SQL is *generated*, not *written*, the way object code is generated from source code. There is exactly one place where truth lives, and the schema is a deterministic function of it — and so is the data access: reading a row becomes decoding it into the structure, writing becomes binding the structure's values into the statement. Both fall out of the one definition.

Under this arrangement, the Swift and the SQL *cannot* drift — not because a discipline keeps them aligned, but because one is *computed* from the other. Agreement is structural, not disciplinary. The entire class of schema-drift defects is eliminated by construction, the way a strong type system eliminates a class of errors at compile time rather than leaving them to runtime vigilance.

This also answers the obvious rejoinder — *let the AI keep the two in sync; it can see the whole codebase*. It can see both files, but seeing is not holding. Keeping two authored sources in agreement is a non-local invariant: nothing in the Swift structure says a schema elsewhere must change with it, and nothing in the schema points back. It is the same kind of rule the tool already broke when it opened a second connection — honored where locally salient, forgotten where not — except worse, because the coupling must be re-remembered on every edit, and the tool carries no memory of it from one change to the next. A human maintainer at least holds the intent, however imperfectly; the tool holds nothing across invocations. Structural derivation removes the remembering from *every* agent — human or AI — which is the only thing that actually works.

The Second Face: Code That No Longer Applies

The schema layout is one face of architectural disagreement: the AI organizing on the wrong axis. There is a second, and it is the mirror image — not structure in the wrong place, but structure that should not be there at all.

The orphaned abstraction seen earlier is the specimen: a service and its protocol, registered and conformed to, apparently required, whose actual consumer — a removed feature — was gone. The scaffolding remained, still compiling, still executing, serving nothing.

This is the harder problem for an AI, and the reason is worth stating plainly. An AI generally has — and wants — visibility into the entire codebase. But whole-codebase visibility does not confer purpose. The AI can see that a piece of code *exists*, that it is *referenced*, that it *compiles* — and to a tool reasoning over text that is present, present code is indistinguishable from live code. What it cannot see is that the code no longer *applies to any feature*: that its reason for existing was deleted when the feature it served was removed. Deadness is not a property of the text. It is a fact about *purpose and history* — what the code was for, and that the reason is gone — and neither is written anywhere the AI can read.

So the AI preserves the dead code, or builds atop it, or reasons as though it matters, because from where it sits the code *looks* alive. It has more context than any human — the whole codebase at once — and is still worse at this than a human who holds far less of it in view, because the judgment required is not "what is here?" but "what here still earns its place?" That is a question about intent, not text, and it is answered only by someone who remembers what the code was for and knows the reason is gone.

Both faces are the same failure underneath. Wrong shape and dead weight are two ways the tool fails at the one thing it cannot do: hold the system's *purpose* — why each piece exists, where it belongs, and whether it should exist at all. The tool sees the code. It does not see the reasons. And the reasons are the architecture.

The Insidious Part: The Tests Pass

The natural rebuttal is: *this is what tests are for*. Write the tests, run them, catch the regressions. The rebuttal fails, and understanding *why* it fails is a core contribution of this paper.

The implementation defects in *The Characteristic Failure…* pass their tests.

The fallback in *The Fallback That Contradicted the* Design satisfies a test that asks "does this return a storage object?" — it does. No test asks "does this respect the deferred-error pattern?" or "does this avoid a second connection?" — because those are architectural properties, not behavioral assertions, and no one wrote a test for them. You largely *cannot* write a straightforward test for "respects the system's intent," because that is judgment, not a checkable output.

This is a fundamental limit of test-driven development, not a gap in a particular test suite. I use TDD as a matter of course — when I write tests I deliberately shut off the implementation side of my mind, so that I am testing the *designated behavior* and not the code I am about to write, and I design code to be testable in the first place. I say this because the limit I am about to describe is not a failure of discipline. It is a property of what tests *are*: they verify behavior, and behavior is not architecture.

TDD's discipline is to write the smallest code that makes the tests pass — and this treats the tests as the definition of correct. But a test suite is only as good as its author's understanding of the API. The tests assert what the test engineer knew to check; 'minimal code to pass' then satisfies exactly that understanding and stops. Everything the author did not understand — the deferred-error contract, the single-connection invariant, the requirement that dead code be removed — is not asserted, and so, by design, not built for. The minimality that TDD prizes as discipline is, at the architectural level, a mechanism for omitting precisely what no one thought to test. Correctness is defined by the tests; the tests are bounded by understanding; and the method optimizes toward that bound, not past it. This means that the tests — or even worse, the tests written by the AI — have the following structural failures:

- **Tests verify behavior at the bar you thought to set.** They are only as complete as the cases you imagined. Architectural violations are, characteristically, the failures you did *not* anticipate — if you had, you would have designed against them. TDD is structurally blind to the failure modes outside the tests, and this class of defect lives there.
  This bound is not incidental; it is built into the method. TDD assumes the developer and the tester are the same person, and the test-first discipline

exists to make the developer meet a target, but the target is set by that same developer, wearing the tester's hat. The independent check that testing is meant to provide collapses into self-verification: one mind sets the bar and one mind clears it, and the bar can be no higher than that mind's understanding. Write the test before the code and you have disciplined the *timing*; you have not added a second understanding. The discipline of thinking as the tester first — done rigorously, holding the implementation out of mind — is a real separation of *stance*, but it is a separation within one mind, bounded by one understanding. Whatever the developer-tester did not grasp, neither the code nor the tests will reflect — because both come from the same source. The AI is only the limit case of this: developer and tester are not merely the same person but the same model, checking its own work against its own blind spots, with no daylight between them at all.

- **Architecture is global, target-relative, and often implicit.** An invariant like "fail gracefully, never crash the launch" or "one shared connection" spans multiple components and is frequently unstated. It is not a local behavior you assert on one function; it is a whole-system property — and what counts as *correct* depends on the target. The persistence layer that is right on a phone, where the data cannot all fit in memory at once, is wasted overhead on a workstation that can hold it all; a synchronous call that is right against local storage is wrong against a network, and wrong again against the open Internet. The core architecture can be shared, the best implementation stays platform-specific, which is why you can't "average" a good architecture into a product. Encoding all of that as tests would require encoding judgment itself — and judgment is the one thing a test cannot hold.

- **Green means "behaves as specified," not "is correct."** Those two diverge exactly at the architectural level. Code can satisfy every specified behavior while being architecturally unsound: wrong abstraction, violated invariant, latent inconsistency. The tests certify the former and are silent on the latter.

There is a second-order trap. A well-structured, highly testable architecture — and the good ones are deliberately and completely testable — produces *many* green checkmarks with little effort. That abundance of green *feels* like strong

verification. It is not. It is thorough verification of the layers that can be verified — behavior and structure — and complete silence on the layer that requires judgment. Good testability can therefore *amplify* the false confidence: more green, on code that is still architecturally wrong. The framework enforces the shape; it does not certify the soundness of what you put inside it.

The AI passes the verifiable layers and fails the judgment layer. And because the verifiable layers are green, the failure is invisible to the process that is supposed to catch it. This is the insidious case: not code that is obviously broken, but code that is *plausibly correct* — compiles, fits the framework, tests green — which is exactly the code that ships.

The Implication: Expertise Matters More, Not Less

None of this argues against the tools. Fire is not useless. AI coding tools are a real force multiplier: faster first drafts, less boilerplate grind, quick exploration of options, genuine acceleration on well-trodden ground. Used as an assistant, the value is real.

But the multiplier acts on the judgment of the person wielding it. It makes a capable engineer faster; it does not make a non-engineer into an engineer. In the hands of someone with architectural judgment, the tool accelerates the work and the human catches the locally-plausible-but-globally-wrong defects — a net gain. In the hands of someone without that judgment, the same tool produces fluent, plausible, architecturally-flawed code that ships, because no one in the loop can see the flaw.

The multiplier works on *learning*, too, not only on output — and this is where its real leverage lies. A strong architect on one platform, with even a little knowledge of a second and an AI to bridge the gaps, can produce good architecture on the second — better the more of that platform they already know. The tool can take a rough or dated familiarity and help turn it into current, working competence in a short time. But notice what this requires: a foothold. The multiplier needs a multiplicand. Extend from iOS to Android — adjacent platforms, related rules — and a little knowledge plus the tool goes a long way. Extend from iOS to Windows or Linux, where the basis is fundamentally different, and there is far less for the tool to multiply: without enough of the target's rules to *judge* the output, the plausible-but-wrong code has no reviewer, and the danger returns in full. The

rules are knowable, and acquiring even a little of them unlocks the multiplier — but the tool amplifies expertise; it does not supply it. Where you have none, it multiplies nothing, or multiplies your error.

The consequence runs opposite to the anxious narrative. There is now a fluent code generator producing plausible-but-architecturally-suspect code at scale. The scarce, essential, irreplaceable capability is the judgment to review it, catch what is globally wrong, and hold the system's invariants. That is not diminished by these tools. It is the bottleneck, and the differentiator. And because the tool multiplies whatever expertise you bring — amplifying a foothold into reach, current knowledge into speed — the incentive to *acquire* expertise has never been stronger. Deep engineering expertise is *more* valuable in this era, not less.

The Real Danger, and the Discipline It Requires

The danger is not the AI writing a defect. Defects are ordinary; catching them is the job. The danger is treating the tool as the *master*: accepting its output unreviewed because it compiles, fits the framework, and the tests are green. That is the exact path by which the defects in *The Characteristic Failure…* and *Two Kinds of* Disagreement reach production — not because the tool failed, but because the judgment layer was skipped, and the automated safety nets that were trusted to stand in for it cannot, by their nature, verify architecture.

The discipline follows directly. Keep the tool in the role it is suited to: draft generation, boilerplate, exploration, mechanical transformation. Keep human architectural judgment as the authority that reviews every generated unit against the system's invariants and intent. Ask of each piece the question no test and no framework asks: does this respect *why* the system is shaped the way it is? That question is the master's question, and it cannot be delegated to the assistant, because answering it requires the whole-system model the assistant does not possess.

Fire that is respected warms the house. Fire that is trusted to run itself burns it down. AI-generated code is the same. Kept as an assistant, with human judgment as the master, it is a powerful help. Handed the authority of a master, it will fail — confidently, plausibly, and in the common case invisibly.

Conclusion

Like fire, AI makes a great assistant, and a poor master. Its errors are architectural; the automated safety nets teams rely on verify behavior and structure, not architecture; and so its errors pass through green and reach production. The layer that catches them is human architectural judgment — the one layer that cannot be automated, and the one that matters most precisely because the tools are now so fluent at everything else. Use these tools. Respect them. Do not hand them the wheel.

About the Author

AustinSoft is the independent practice of Glenn L. Austin: mobile security architecture; iOS, macOS, and cross-platform systems design; and the kind of architectural judgment this paper is about. The work spans certificate pinning and mobile threat defense, persistence and concurrency design, and the review of systems — human-built or AI-assisted — against the invariants they are supposed to hold.

This paper's argument applies with particular force to security. An AI aimed blindly at a security problem produces code that is locally plausible and globally unsound in exactly the ways described above — and in security, globally unsound is not a maintenance cost, it is an exposure. Used as an assistant, under judgment, the tools are genuinely useful: I use AI myself to double-check "accept-valid-only" interfaces, precisely the kind of narrow, well-specified verification where a second pass earns its place. The distinction the paper draws — assistant, not master — is not academic here. It is the difference between a reviewed defense and a plausible-looking hole.

If your team is integrating AI tooling into systems where architecture matters — and in security, it always matters — this is some of what AustinSoft does: architectural review, security design, and a second set of eyes that holds the whole system in mind.

austinsoft.com · glenn@austinsoft.com