BIT BROS LLC

WHITE PAPER

The Future of Coding Looks a Lot Like Optimization

Adam DeJans Jr.

August 24, 2026

Originally published on LinkedIn

White-paper edition: original prose, reformatted for reading and print.

I've flipped my position hard. A couple years ago, I treated ChatGPT like a worse Stack Overflow. Now I'm all-in on agents. Not vibe coding. I increasingly think humans need to disengage from implementation-level code if we actually want to capture the productivity gains these systems offer. If an agent can produce a day's worth of implementation in twenty minutes and I insist on reading every line, then I haven't really changed the development model.

I've just created an incredibly fast producer and bottlenecked it through human reading speed.

The way I think about this is increasingly similar to how I think about optimization. I don't want to manually inspect every possible decision a solver considered before accepting a solution. I define the objective, encode the constraints, verify the formulation, test the outputs, and then let the machinery search a space that is far too large for me to inspect myself. Agentic software development is starting to feel like the same abstraction shift.

My job is moving away from personally inspecting every generated action and toward designing the system in which good actions are produced.

That does not mean I trust the model. In fact, the entire point is that I don't want to have to trust the model. I want to surround the agent with constraints it cannot negotiate its way around. Function size limits. Cyclomatic complexity thresholds. Dependency and architecture checks. Coverage requirements. CRAP scores. Mutation testing when appropriate. Separate unit and acceptance test streams.

I want deterministic tools to enforce the engineering standards rather than hoping the model remembers a paragraph in a markdown file telling it to "write clean code."

This is where my optimization brain really kicks in. An unconstrained optimizer is usually useless. Give it the wrong objective and it will very efficiently give you the wrong answer. Give it incomplete constraints and it will exploit every hole in the formulation. That is not the optimizer misbehaving. It is doing exactly what you asked. LLM agents are remarkably similar.

If I tell an agent to "finish the feature as quickly as possible," I should not be shocked when it discovers shortcuts I never intended. The problem is not that the agent failed to understand my values. The problem is that I never encoded them.

That is one reason I dislike the idea that prompting alone is enough. In optimization, I would never describe a capacity constraint in a paragraph next to the model and hope the solver respects it. The constraint belongs in the formulation. The same should increasingly be true for agents. "Keep functions small" is a suggestion. A build that fails when complexity exceeds a threshold is a constraint. "Maintain clean architecture" is a prompt.

An architecture test preventing an illegal dependency is a constraint. "Write good tests" is subjective. Mutation testing that demonstrates whether those tests detect behavioral changes gives me something measurable.

I've also seen agents create their own mess and then struggle to get themselves back out of it. They are perfectly capable of producing tangled code, attempting to fix the tangle, introducing another problem, and then spending enormous amounts of compute arguing with the consequences of their own previous decisions. Again, there is an optimization analogy here. A bad formulation can make even an excellent solver look terrible.

If I give the system a poorly structured search space, weak constraints, and no useful signals about quality, I should expect poor performance. Good engineering is increasingly about shaping the search space before the search begins.

That changes what I review. I increasingly care less about reviewing implementation and more about reviewing behavior. The agent can write the implementation. The agent can write the unit tests. I care much more about the acceptance criteria, the Gherkin scenarios, the QA procedures, the architecture, and whether the resulting system actually behaves correctly. I may still manually exercise important paths periodically, but my human attention is concentrated at the level of meaning rather than syntax.

To me, unit tests are largely an implementation artifact. Acceptance tests describe what "correct" means to the business. That is where human judgment has the most leverage. I don't need to personally approve every conditional statement if I have confidence that the behavior contract is correct and that the implementation is forced through a sufficiently strong verification system.

The question becomes less "Do I like this function?" and more "Does this system make the right decision, under the right conditions, for the reasons I intended?"

That last question matters to me because I spend a lot of time thinking about the difference between predictions and decisions. A forecast can be statistically excellent and still lead to a terrible inventory decision. The forecast is an input. The decision is what creates economic value. I think we are going to make a similar mistake with AI-generated software if we obsess over the generated artifact instead of the resulting behavior. Beautiful code is not the objective.

Correct, maintainable, economically useful behavior is the objective. Code is one of the mechanisms used to produce it.

I think this is another step in the same abstraction ladder software engineering has been climbing for decades. We moved from binary to assembly, from assembly to higher-level languages, from manual memory management to garbage collection, and from low-level primitives to frameworks. At every step, the artifact humans directly manipulated moved higher up the stack. Nobody expects me to audit the assembly emitted by a compiler before deploying software.

I trust that layer because I have confidence in the transformation process and because I verify the behavior of the resulting system.

Agents may be pushing us toward another version of that transition. The artifact I author is increasingly not the implementation itself. It is the specification, the behavior contract, the architecture, the constraints, and the evaluation system around the generated implementation. The code becomes something closer to solver output or compiler output. I care about the formulation. I care about feasibility. I care about the objective.

I care about whether the solution generalizes outside the exact scenario used to produce it. I do not need to manually enumerate the search path.

But that only works if the inputs are rigorous and the outputs are aggressively verified. Otherwise this is not a new abstraction layer. It is simply a faster mechanism for producing sludge.

That distinction is why I separate agentic engineering from vibe coding. Vibe coding is basically unconstrained optimization with a badly specified objective. "Make this work." "Build me this feature." "Fix the bug." The agent finds some path to an apparent local optimum, and everybody celebrates until the hidden constraints show up in production. You may have optimized exactly what you asked for while completely missing what actually mattered.

Disciplined agentic engineering is different. I want explicit specifications, domain-language acceptance criteria, architecture rules, automated tests, static analysis, quality thresholds, and verification mechanisms that exist outside the model itself. In optimization terms, I want a well-defined objective function, a feasible region that actually represents reality, and independent checks that prevent the system from gaming whatever proxy I happened to give it.

That is also why I am skeptical of a single metric becoming the target. Anyone who has worked in optimization knows what happens when you optimize a proxy too aggressively. Goodhart’s law shows up fast. If I optimize only coverage, I can get terrible tests with great coverage. If I optimize only cyclomatic complexity, I can produce tiny functions arranged into nonsense. If I optimize only mutation score, I may spend ridiculous amounts of effort defending low-value behavior.

The engineering system needs multiple signals because software quality, like most real decision problems, is multi-objective.

The right answer is not necessarily to create one giant weighted score either. I tend to prefer thinking in terms of hard constraints, soft constraints, and economic tradeoffs. Some things should simply be forbidden. Dependency cycles may be unacceptable. Certain acceptance tests must pass. Security checks cannot be traded away because the code happened to be simpler. Other dimensions should be optimized within that feasible region.

That framing feels much more natural to me than telling an agent to "write high-quality software" and hoping it understands what I meant.

I also assume agents are locally clever and globally dumb. They can be remarkably effective at solving the task immediately in front of them while having almost no instinct for the long-term health of the system. This reminds me of local optimization methods: they can make excellent moves within the neighborhood you give them while completely missing the global structure of the problem. That is why I like explicit roles and handoffs: specifier, coder, cleaner, architect, hardener, QA.

I am not claiming those exact roles are sacred. The broader point is that process gives the agent access to different views of the objective rather than letting one locally focused trajectory dominate everything.

I also don’t think every change requires every possible quality mechanism. It would be easy to turn this philosophy into another religion where every two-line change requires Gherkin, unit tests, QA procedures, mutation testing, architecture analysis, and twelve other checks. That misses the economics of the problem. Verification has a cost. Failures have a cost. Different components have different risk profiles.

The rational amount of verification should depend on criticality and expected consequence, not ideology.

For plenty of work, strong unit tests, static analysis, CRAP thresholds, and architecture checks may be enough. For something financially material, operationally critical, or safety sensitive, I want much stronger independent verification. I think about the gauntlet almost like a risk-adjusted decision policy. Spend verification effort where the expected downside justifies it.

None of this means juniors should skip learning how software actually works. I think the opposite is true. Someone still needs enough engineering judgment to decide which constraints matter. If I have never experienced the damage caused by a dependency cycle, I probably won’t know to prevent one. If I don’t understand modularity, I cannot meaningfully evaluate the architecture.

If I can’t read an acceptance test and recognize that it accidentally encodes implementation rather than behavior, then I am not actually reviewing the contract. I’m just nodding at English.

This is similar to optimization tooling. Having Gurobi does not eliminate the need to understand optimization. In many ways, the more powerful the solver becomes, the more important the formulation becomes. A world-class solver will happily solve the wrong model faster than you ever could manually. The same is going to be true of agents. Better models do not eliminate engineering judgment. They amplify whatever judgment you encode into the surrounding system.

The work is moving up the abstraction ladder, but the underlying craft still matters. Architecture matters. Modularity matters. Testing matters. Customer empathy matters. Economic reasoning matters. If anything, these things become more important because the implementation can now be generated so quickly that bad judgment propagates much faster than before.

That is also why I don’t think "AI replaces programmers" is a particularly useful framing. It replaces a tremendous amount of typing and mechanical implementation work. But somebody still has to decide what should exist, define what correct means, establish the objective, encode the constraints, reason about tradeoffs, design the verification system, and keep the resulting codebase from collapsing into chaos.

The biggest caveat is that none of this works if the gauntlet is imaginary. "Please write maintainable code" is not a gauntlet. "Please follow SOLID" is not a gauntlet. A list of principles buried in an AGENTS.md file is not a gauntlet. I want linters, architecture checks, complexity limits, coverage requirements, mutation scores, acceptance tests, and builds that actually fail when the system violates the rules.

The constraints need to exist independently of the model’s willingness to obey them.

So yes, I am increasingly comfortable with the idea of not reading most AI-generated implementation code. But I’m comfortable doing that because I want to demand considerably more from everything surrounding the implementation. I want to inspect the objective. I want to understand the constraints. I want to review the acceptance criteria. I want to validate the architecture. I want to inspect the failures. I want to see whether the tests actually discriminate good behavior from bad behavior.

If I want agent speed, I cannot continue using human reading speed as the primary quality gate. I need to move quality into the specification, architecture, constraints, metrics, tests, and verification mechanisms that the agent cannot negotiate with.

That feels very familiar to me. In optimization, the solver is rarely the hard part. The hard part is deciding what problem you are actually solving, what objective represents value, what constraints represent reality, and how you will know whether the answer is any good.

AI engineering is heading in the same direction.

The code is becoming the solution.

Our job is increasingly to formulate the problem.