You Can't Touch This

Vibe coding raised the floor on visible bad code. But the flood rose faster, rented confidence replaced visible incompetence, and the basement became searchable. On authorship, competence, and risk in AI-generated software.

2026.05.17 01:19 DENNIS HEDEGREEN ANALYSE V1.0
HTTPS://HEDEGREENRESEARCH.COM/ARTICLES/YOU-CANT-TOUCH-THIS/

Vibe Coding, Rented Confidence, and the Code Nobody Credits

I. You Can't Touch This

Bad code did not disappear.

That would be too easy.

What disappeared first was the old visible kind of bad code: the broken import, the missing dependency, the syntax error, the hardcoded key sitting in plain sight, the SQL query that looked wrong before it even ran, the endpoint with no tests, no validation, no error handling, and no shame.

That kind of bad code has become harder to ship.

Not impossible.

Harder.

In May 2026, a developer working with Claude Code, Codex, Cursor, or any serious AI coding environment is no longer alone with the editor. The tool watches. It formats. It explains. It patches. It notices the missing test. It asks why the secret is in the repository. It suggests the safer query. It refactors the file before the human has finished being messy.

The floor rose.

But the flood rose faster.

Because the same tools that make obvious bad code harder to ship also make code much easier to produce. More code. Faster code. Cleaner-looking code. Code with tests. Code with comments. Code with interfaces, handlers, migrations, mocks, fixtures, schemas, and deployment notes.

The old question was:

Can this person write code?

The new question is:

How much code can this person cause to exist before anyone understands what was built?

That is the shift.

AI coding tools did not end bad code.

They changed where bad code hides.

And they introduced something more dangerous than broken code:

rented confidence.

II. My, My, My Music Hits Me So Hard

The most dangerous sentence in software right now is not:

"This code is broken."

It is:

"This code looks professional."

AI-generated software often looks like senior engineering. The naming is reasonable. The folder structure makes sense. The error handling is present. The tests exist. The commit message sounds calm.

That does not mean the system is understood.

A tool can generate the shape of competence without transferring competence to the operator. It can produce a working authentication flow without teaching threat models. It can create a database schema without transferring migration judgment. It can write a beautiful abstraction around a service that should not exist.

This is rented confidence.

The operator feels safer because the output looks safer. The interface is clean. The warnings are fewer. The tests are green. The agent speaks with the tone of someone who has seen this pattern before.

Maybe it has.

But the operator may not have.

That gap matters.

Broken code announces itself.

Plausible code negotiates with you.

It says:

ship me.

III. Mythos Heard the Basement

Then Claude Mythos appeared.

Mythos is not simply a coding assistant. It is closer to a machine reader for the buried structure of software. Anthropic describes Claude Mythos Preview as a cyber-capable model used in Project Glasswing to help find and fix vulnerabilities in important open-source systems.

That matters because the optimistic story of AI coding says:

Bad code becomes harder to ship.

Mythos adds the darker version:

Old bad code becomes easier to find.

A vulnerability that sat unnoticed for ten years is not gone because nobody touched it. It is still there, under the floorboards, waiting for a system capable of hearing the pattern.

That changes the meaning of software quality.

Code is no longer judged only when it is written, reviewed, tested, and deployed. It can now be re-read later by machines with capabilities the original developers did not have, against threat models they never imagined.

The floor rose.

But the basement became searchable.

That is the part most vibe-coding optimism misses.

AI does not only write new software.

It rereads old software.

And tomorrow's reader will be stronger than today's writer.

Ethan Mollick's line about AI is usually given as a warning about acceleration: today's AI is the worst AI you will ever use.

For software, the darker version is this:

Today's coding tool is the weakest code reviewer your software will ever face.

The code shipped now is not being written into a stable world. It is being written into a future where every year brings better auditors, better exploiters, better migration agents, better maintainers, better attackers, and better systems for finding what humans missed.

Bad code no longer ages in silence.

IV. I Told You Homeboy

There is a man named Bob who ran a paper distribution business.

He never touched a single newspaper.

Not once.

That was the point.

He understood that the value was not in touching the paper. The value was in the system that moved it: trucks, routes, timing, distribution, trust, reliability.

He did not need to be the paper to own the business around the paper.

"Don't touch the paper" became a principle.

Vibe coding is that principle applied to software.

You describe what you want. The tool builds it. You review. You steer. You reject. You approve. You ship. You do not necessarily write the function. You do not manually chase the import. You do not personally hold every edge case in your head.

This is not automatically laziness.

It is abstraction.

A music producer does not play every instrument. A film director does not hold every camera. A logistics company does not manufacture every object it moves.

The question is not whether abstraction is real work.

It is.

The question is whether the operator still understands the material well enough to own the consequences.

Because Bob did not touch the paper.

But Bob understood the distribution system.

The dangerous vibe coder does not touch the code and does not understand the system.

That is not abstraction.

That is rented confidence.

V. Fresh New Kicks and Pants

Vibe-coded software looks good.

That is part of the trap.

The app has authentication. The dashboard has filters. The database has indexes. The API has validation. The readme has setup instructions. The deployment script works. The UI has empty states. The tests pass.

Fresh kicks.

New pants.

You look like you belong on stage.

But underneath the visible product is a stack of inherited competence.

Patterns from open-source repositories. Security practices from old blog posts. Architecture decisions from frameworks. Naming conventions from documentation. Training traces from millions of developers who solved similar problems before. Platform defaults from cloud providers. Billing rules from model vendors. Invisible assumptions from the agent that chose one design path and not another.

The output is real.

The product may work.

Users do not care who wrote the code if the button does what the button says.

But somewhere between the prompt and the product, a question gets skipped:

Whose knowledge is this?

Not in the simple legal sense.

In the structural sense.

The model did not invent competence from nothing. It carries compressed history. That history may not appear as a direct copy. It may not be a recognizable sample. It may not be infringement. It may be pattern, convention, habit, idiom, architecture.

But it is still inherited.

And inheritance without memory is difficult to credit.

VI. Break It Down — Stop. Hammer Time.

In 1990, MC Hammer released "U Can't Touch This."

It became a global hit. The song was everywhere. The phrase became a cultural object. The pants became a silhouette. The dance became a signal.

Hammer did not merely release a track.

He created a moment.

But the foundation of the song — the beat, the riff, the hook underneath the movement — came from Rick James' "Super Freak."

The writing credits for "U Can't Touch This" include MC Hammer, Rick James, and Alonzo Miller. The dispute over the "Super Freak" sample was reportedly settled out of court, with James receiving songwriting credit and royalties.

Hammer made the hit.

But the beat had a prior owner.

That is why the comparison matters.

Not because AI-generated code is literally the same as sampling music.

It is not.

A sample is audible. A listener can often hear the borrowed material. A training dependency is buried. A model does not usually replay one recognizable source. It produces something new from a statistical landscape of prior work.

So the structural question is not identical.

It is harder.

MC Hammer's beat could be heard.

The beat under AI-generated code often cannot.

The vibe coder prompts Claude Code or Codex to build a SaaS application. The code works. It is secure enough to deploy. It follows patterns the operator did not invent and may not understand. Revenue comes in.

Who wrote the music?

The person who described the product?

The model that generated the implementation?

The company that rents access to the model?

The open-source developers whose collective work trained the patterns?

The cloud provider that hosts the system?

The customer whose behavior reveals what the product should become?

The answer is not one name.

It is a stack.

VII. Who Owns What Is Left?

Copyright law is not ready for the emotional simplicity of:

I prompted it, so I made it.

The U.S. Copyright Office has said that generative AI outputs can receive copyright protection only where a human author has determined sufficient expressive elements, including human-authored material, creative arrangement, or meaningful modification — but not from the mere provision of prompts alone.

That does not settle software.

Software is not only expression. It is also function, process, interface, architecture, contract, deployment, service, business logic, and operational behavior.

The vibe coder may own the company.

They may own the domain.

They may own the customer relationship.

They may own the product direction.

They may own the risk.

They may own the deployment.

They may own the dance.

But ownership of the dance is not ownership of the beat.

And ownership of the beat is not ownership of the entire machine that plays it.

This is where AI coding becomes uncomfortable. The operator can create real commercial value without being the primary source of the competence embodied in the code. That does not make the operator fake. It makes authorship layered.

In vibe coding, the human increasingly owns intention, framing, taste, risk, market timing, and final acceptance.

The model owns execution.

The training corpus owns buried craft.

The platform owns access.

The cloud owns runtime.

The future auditor owns the right to reveal what nobody understood.

That is not clean authorship.

That is infrastructure authorship.

VIII. Either Work Hard or You Might as Well Quit

"Work hard" used to mean learning the machine closely.

Read the docs.

Understand the framework.

Write the function.

Debug the error.

Refactor the mess.

Deploy the service.

Fix the incident at 02:14.

That work still exists.

But the center of gravity has moved.

Working hard on implementation is no longer enough, because implementation itself is becoming increasingly available as a service. Modern coding agents can implement, refactor, debug, test, validate, review, operate browser tabs, inspect running applications, and increasingly act across environments that used to require direct human manipulation.

The tool works harder than most people can.

It does not sleep.

It does not get bored by boilerplate.

It does not resent migrations.

It does not need a clean desk, coffee, or emotional readiness.

So the new version of work hard is not:

type more.

It is:

define better.

Constrain better.

Question better.

Read better.

Own better.

The creative act shifts from implementation to definition.

From how to what.

From code to concept.

From touching the paper to understanding the route.

This is why vibe coding is both liberating and dangerous. It gives building power to people who were locked out of software by syntax, tooling, and gatekeeping. That is real. It matters. It opens doors.

But it also lets people build systems they cannot yet read.

And a system you cannot read is a system you cannot fully govern.

IX. Build, Fix, Compute, Repeat

Build/fix has always existed.

Software was never finished.

It was shipped, patched, refactored, migrated, deprecated, rewritten, secured, and shipped again. Every serious codebase has always lived inside a loop.

Build.

Fix.

Build.

Fix.

AI does not remove that loop.

It accelerates it.

A tool that can generate a thousand lines in minutes also creates a thousand lines that must later be understood, audited, migrated, secured, and perhaps replaced. The code may be cleaner at the surface. The syntax may be better. The tests may be present.

But the system has still grown.

And every growing system becomes a future maintenance object.

This is where compute enters the story.

The first wave sells generation.

The second wave sells review.

The third wave sells repair.

The fourth wave sells migration from the code the first wave helped create.

That is not necessarily fraud.

Software has always worked this way.

But the meter changed.

The old cost was human time.

The new cost is human attention plus machine time.

Tokens to write it.

Tokens to test it.

Tokens to review it.

Tokens to fix it.

Tokens to explain it.

Tokens to migrate it when the framework changes, the model improves, the dependency breaks, the security landscape shifts, or the company decides the old architecture was wrong.

Compute does not only build the future.

It invoices the repair of the future it just built.

This may become the great hidden economy of AI software: not the first generation of code, but the permanent re-reading of code by stronger systems.

Generate.

Audit.

Patch.

Refactor.

Regenerate.

Audit again.

Compute, compute, compute.

The machine does not end maintenance.

It turns maintenance into a billable loop.

X. The Rest Can Go and Play

MC Hammer filed for bankruptcy in 1996.

That fact is often used as a cheap joke.

It should not be.

The more useful lesson is not that Hammer was foolish. It is that a hit can work while the machine around the hit becomes unsustainable.

The song worked.

The dance worked.

The image worked.

The market worked.

Until the costs around the moment outweighed the moment.

That is the warning for vibe coding.

The first build can be cheap.

The system around the build is not.

The subscription. The tokens. The cloud. The database. The observability. The security review. The migration. The compliance layer. The customer support. The rewrite when the first architecture turns out to be wrong. The audit when a future Mythos-class system finds what the original build missed.

The code may be cheap.

The codebase is not.

Hammer did not disappear. He survived the hit, the lawsuit, the bankruptcy, the punchline, and the reinvention. He remains a cultural signal because he understood something real: distribution, spectacle, timing, and transformation.

That makes him a better metaphor, not a worse one.

The vibe coder may be Hammer.

Not fake.

Not useless.

Not merely lucky.

But not the sole origin either.

The operator who turns rented competence into a moment.

The person who makes the dance visible.

The person who understands timing before others do.

But still:

the beat matters.

And the beat has a history.

XI. You Can't Touch This

Rick James made the beat.

MC Hammer made the moment.

The crowd remembered the dance.

The lawyers remembered the credits.

Software is entering the same uncomfortable territory.

The app works.

The code compiles.

The tests pass.

The customer pays.

The tool improves.

The auditor gets smarter.

The basement becomes searchable.

And somewhere under the product, under the prompt, under the model, under the compute, under the repository, under the history of millions of developers whose patterns became infrastructure, the beat keeps playing.

You can ship the dance.

But you should know whether you own the music.

Writer(s): Rick James · Alonzo Miller · MC Hammer
Writer(s): Claude Mythos · GPT-5.5 · Dennis Hedegreen
Publisher: Compute, probably

Hedegreen Research · Open analysis · Open method · Open question

RELATION MEMORY

NEARBY When a Platform Leaves the Market · Both articles examine what happens when platform control limits access to ordinary users.

NEARBY Scale Raises Confidence, Not Capacity · Both warn against mistaking technical scale or polish for real capacity in the user's life.

RETURNS TO Today's AI Is The Worst AI You Will Ever Use · Returns to the capability/access gap through a cultural and platform-control example.

CANONICAL URL: HTTPS://HEDEGREENRESEARCH.COM/ARTICLES/YOU-CANT-TOUCH-THIS/
PDF VIEW GENERATED: 2026-07-11T15:35:46.160Z