Constraint-Driven Vibe Coding: Measuring AI-Generated Code Beyond Speed

A Laravel Importer Study on Correctness, Failure, and Production Pressure

Adnan Mehrat

Abstract

Vibe Coding is often discussed as a speed story: how quickly an AI tool can turn a prompt into working code. That is the least interesting measurement. Fast code can still be wrong, expensive, fragile, or unsafe. This study treats Vibe Coding as an engineering problem and asks a narrower question: what happens when generated code is judged by production constraints rather than by first-run success?

The case study is a Laravel product CSV importer. Three strategies are compared on the same Amazon-derived benchmark dataset: a mid-level Laravel baseline, raw vibe coding, and constraint-driven vibe coding. The benchmark fixes the expected number of inserted, updated, and rejected rows before execution. The final comparison uses measured Laravel Artisan command runs, not estimated results.

The Claim

The danger in AI-generated code is not that it can be bad. Developers have always been able to write bad code. The new danger is that the code can look complete before it has been forced to prove anything.

A demo importer may read a CSV, insert products, and print a cheerful summary. Production asks colder questions. Does it reject broken rows without stopping? Does it avoid loading the whole dataset into memory? Does it turn one import into thousands of queries? Does it log enough evidence to debug bad supplier data next week?

Most public discussion around AI coding is too confident for the amount of evidence it shows. One side says AI is already better than most programmers. Another side says AI code is disposable junk. Both claims are usually argued through impressions: a screen recording, a surprising completion, a broken generated function, or a personal workflow story. Those examples are not useless, but they are not measurement.

This study takes a smaller route. It does not ask whether AI is better in general. It asks whether a generated importer behaves better when the prompt contains the constraints a production engineer would normally carry in their head. That makes the question testable. The code either inserts the expected rows, or it does not. It either rejects the controlled invalid rows, or it lets them in. It either creates avoidable database pressure, or it avoids it.

This article compares three ways of producing the same feature:

- **Mid-level Laravel baseline**: practical Laravel code using familiar tools.
- **Raw vibe coding**: a broad AI prompt with only small integration fixes.
- **Constraint-driven vibe coding**: AI generation with explicit limits for memory, validation, queries, logging, and testability.

The comparison is not "AI versus human." It is a comparison between levels of engineering pressure.

Prior Evidence and Public Debate

Karpathy's original vibe coding post matters less as a rigorous claim than as a cultural label. It named a mode of work many developers had already felt: describe intent, accept large generated changes, and steer the result by conversation. That is useful as a description of practice, but it is not by itself a standard of quality.

The optimistic case is real. GitHub's Copilot research reported that, in a controlled JavaScript HTTP-server task with 95 professional developers, the Copilot group finished substantially faster than the control group. That result is worth taking seriously because it used random assignment and a scored task. It is also narrower than many public retellings of it: the task was bounded, the success criteria were known, and the study measured completion of that task rather than production durability.

The skeptical case is also real. METR's 2025 study tested experienced open-source developers on real issues in repositories they already knew, and found that allowing AI increased completion time in that setting. METR now labels those results historical because newer 2026 results are available, so the point here is not that AI always slows developers down. The point is methodological: realistic work can overturn what feels obvious in smaller demonstrations.

Trust surveys point in the same direction. The 2025 Stack Overflow Developer Survey reported that more developers distrusted the accuracy of AI-tool output than trusted it. That does not mean developers reject AI. It means verification has become part of the workflow, especially for people accountable for what ships.

The strongest framing for this article comes from DORA's 2025 AI-assisted software development report: AI behaves like an amplifier of an organization's existing strengths and weaknesses. A strong prompt, a known contract, tests, logging, and review give the model a system to amplify. A vague request gives it mostly surface shape to optimize.

Security research makes the same warning sharper. Broken by Default, a 2026 formal-verification study of AI-generated security-critical code, reported vulnerable artifacts across all tested models. This importer benchmark is not a security benchmark, but it borrows the same discipline: do not ask whether the output looks plausible; ask which contract it can survive.

Why Speed Is a Weak Signal

The fastest implementation is often the easiest one to overvalue. A command exists. A database table receives rows. The terminal prints a summary. The empty application now has motion, and motion feels like progress.

For importers, that feeling is especially misleading. A bad importer can look successful because bad data is still data. It may store products with invalid statuses, quietly coerce broken quantities to zero, accept malformed variants, or overwrite existing rows by accident. The damage appears later, when search filters fail, stock counts drift, or support has to explain why a product exists with a key that should have been rejected.

That is why this benchmark gives every strategy the same dirty input and a known expected outcome. The test is not whether the command runs. The test is whether it leaves the database in the right state and tells the truth about what it rejected.

Benchmark Shape

The raw source is an Amazon product dataset. It contains realistic product text, prices, categories, and variant noise. It is not used directly as the benchmark because public data does not automatically provide a known expected outcome. Instead, it is transformed into a controlled import workload.

This distinction is important. Real data gives the benchmark texture: long product names, inconsistent descriptions, empty supplier fields, prices formatted as strings, and variant data that arrives in awkward shapes. Controlled data gives the benchmark judgment: before a command runs, the expected inserted, updated, and rejected counts are already known. Without that second part, the study would only describe behavior; it could not score it.

The generated benchmark contains:

$$
\begin{aligned}
I &= 600 \\
U &= 250 \\
E &= 80 \\
\text{Total Rows} &= 930
\end{aligned}
$$

I is expected inserts, U is expected updates, and E is expected rejects. The update rows are made measurable by seeding matching products before each run. Invalid rows are injected deliberately: missing product keys, empty names, invalid prices, bad quantities, unsupported statuses, malformed JSON variants, oversized descriptions, and duplicate product keys.

The operational setup has three separate parts: controlled import input, preloaded existing products, and a ground-truth table used only after execution. The importer sees the input rows and the preloaded products. It never sees the expected answer while it runs; that answer is used only for scoring.

Importer Contract

The benchmark does not model Amazon's full catalog. It defines a small internal product contract that is enough to test importer behavior.

Internal field	Amazon source	Type	Rule
`source_id`	`Uniq Id`	string	Original source row identifier.
`sku`	`Sku`	string	Primary product key when present.
`asin`	`Asin`	string	Fallback identifier when `Sku` is missing.
`name`	`Product Name`	string	Required and non-empty.
`brand`	`Brand Name`	string	Optional.
`category`	`Category`	string	Optional.
`price`	`Selling Price`	decimal	Required; numeric and greater than zero.
`list_price`	`List Price`	decimal	Optional; numeric when present.
`quantity`	`Quantity`	integer	Required in benchmark; zero or greater.
`status`	`Stock`	string	Must be one of the allowed benchmark statuses.
`variants`	`Variants`	json	Optional; valid JSON when present.
`description`	`Product Description`	text	Optional; length-limited.

Product identity follows a fixed fallback chain: `Sku`, then `Asin`, then `Uniq Id`. A row without any usable key is rejected. This matters because the inspected source data has sparse `Sku`, `Asin`, `Quantity`, and `Stock` fields, while `Uniq Id` and `Selling Price` are more reliable. The benchmark therefore keeps the realism of the source data but gives every implementation a stable contract.

The contract is intentionally modest. It does not try to model images, shipping weight, ingredients, dimensions, or every Amazon detail. That restraint is part of the benchmark design. The goal is not to build a catalog system. The goal is to isolate the importer behaviors that tend to break first: identity, numeric parsing, status normalization, JSON validation, duplicate handling, and error reporting.

Scenario Protocol

Each strategy is implemented as a Laravel Artisan command in the same application. All commands read the same controlled import data and the same preloaded product state. They write reject logs and return inserted, updated, rejected, failed, runtime, peak memory, and query count.

The protocol matters because otherwise the comparison becomes theater. If the raw version is repaired after seeing the results, it is no longer raw. If the baseline is quietly optimized, it stops representing ordinary Laravel code. If the constraint-driven version gets an easier dataset, the result is meaningless. The rules below keep each strategy in its lane.

Scenario	Implementation rules	What it is meant to reveal
Mid-level Laravel baseline	Normal command, Eloquent, strict validation, row-by-row reading, basic reject logging, `updateOrCreate` allowed, no batching redesign.	Whether ordinary correct Laravel code pays hidden query and runtime costs.
Raw vibe coding	Broad importer prompt, shallow validation, full-dataset loading, minimal fixes only, no architecture repair after seeing results.	Whether "it runs" hides bad validation and weak failure behavior.
Constraint-driven vibe coding	No full-dataset loading, strict validation, preloaded product keys, transaction, structured reject logging, reduced query pressure, testable helper classes.	Whether explicit constraints change the generated solution, not only the prompt wording.

The raw strategy is not sabotaged. It is intentionally under-specified. That is the point. Many generated features fail not because the prompt asks for bad code, but because it does not define the pressure the code must survive.

The mid-level baseline is also not a straw man. A Laravel command with Eloquent validation and `updateOrCreate` is a recognizable solution. Many teams would accept it for a first production pass, especially when the dataset size looks manageable. Its weakness is expected to appear in query shape rather than in correctness.

The constraint-driven version is allowed to be more deliberate. It can preload existing product keys, stream the CSV, separate validation rules, use a transaction, and log rejections with reasons. It is not receiving a different

problem. It is receiving a more precise contract before code is written.

What the Constraint Changes

The difference starts before code exists. A raw prompt usually names the feature. A constraint-driven prompt names the feature and the conditions under which the feature is allowed to count as finished.

```
Raw prompt:
Build a Laravel CSV importer for products. It should read the CSV,
insert new products, update existing products, reject invalid rows,
and return a summary.
```

```
Constraint-driven prompt:
Build a Laravel product importer for the benchmark CSV. Do not load
the full dataset into memory. Use sku, then asin, then source_id as the
product-key fallback. Validate price, quantity, status, variants, and
description length. Reject duplicate product keys inside the import
input. Preload existing keys to reduce database queries. Log every
rejected row with row number and reason. Return measured counts,
runtime, memory, and query count.
```

That difference also appears in code. The raw version checks only the obvious fields. The constrained version treats validation as a contract.

```
// Raw-style validation: enough for a demo, not enough for the contract.
if ($productKey === '' || trim($row['name']) === '' || $row['price'] <= 0) {
    reject($rowNumber);
}
```

```
// Constraint-driven validation: every controlled failure has a rule.
if ($productKey === '') return reject($rowNumber, 'missing_product_key');
if (trim($row['name']) === '') return reject($rowNumber, 'empty_name');
if (! is_numeric($row['price']) || $row['price'] <= 0) return reject($rowNumber, 'invalid_price');
if (! preg_match('/^-?\d+$/', $row['quantity']) || $row['quantity'] < 0) return reject($rowNumber, '
    invalid_quantity');
if (! in_array($row['status'], $allowedStatuses, true)) return reject($rowNumber, 'invalid_status');
if ($row['variants'] !== '' && json_decode($row['variants']) === null && json_last_error() !==
    JSON_ERROR_NONE) {
    return reject($rowNumber, 'invalid_variants_json');
}
```

The important part is not that the second snippet is longer. The important part is that it makes failure visible. A row is not merely skipped; it is classified. That classification is what lets the benchmark compare implementations instead of relying on a general feeling that one version looks cleaner.

Scoring

The Vibe Coding Production Readiness Index, or VCPRI, combines correctness, resource use, security, failure behavior, and maintainability:

$$\text{VCPRI} = 0.25C + 0.20M + 0.15R + 0.15Q + 0.10S + 0.10F + 0.05T$$

where C is Correctness, M is Memory Efficiency, R is Runtime, Q is Query Efficiency, S is Security, F is Failure Handling, and T is Testability and Maintainability.

Checklist scores use:

$$\text{Score} = \frac{\text{Passed Checks}}{\text{Total Checks}} \times 100$$

Resource scores compare each implementation with the best measured value:

$$\text{Memory Score} = \frac{\text{Lowest Peak Memory}}{\text{Current Peak Memory}} \times 100$$

$$\text{Runtime Score} = \frac{\text{Fastest Runtime}}{\text{Current Runtime}} \times 100$$

$$\text{Query Score} = \frac{\text{Lowest Query Count}}{\text{Current Query Count}} \times 100$$

The weights are not universal. They fit this importer, where a fast wrong result has no value and database pressure matters.

Correctness receives the highest weight because wrong data is not a performance problem; it is a trust problem. Memory, runtime, and query count receive strong weights because import jobs often fail under scale before they fail in small examples. Security, failure handling, and maintainability receive smaller weights, but they keep the score from rewarding code that is fast and brittle.

Metric	Weight	Reason
Correctness	0.25	The importer must leave the database in the expected state. A fast wrong import is still wrong.
Memory	0.20	CSV imports often grow by input size. A memory ceiling is a hard failure, so memory is weighted above runtime.
Runtime	0.15	Runtime matters, but a slower correct import can still be scheduled or optimized.
Query count	0.15	Query pressure predicts database load and usually exposes row-by-row persistence mistakes.
Security	0.10	Unsafe parsing or unchecked values can turn a data job into an attack surface.
Failure handling	0.10	Bad rows should be isolated, counted, and explainable without stopping the job.
Testability	0.05	Maintainability matters, but it supports the other metrics rather than replacing them.

Memory is weighted higher than runtime in this scenario because runtime is elastic and memory is not. A job that takes twelve seconds can often run at night. A job that exhausts memory does not finish at all. Query count receives the same weight as runtime because the database is shared infrastructure: one careless importer can slow down work that has nothing to do with the import.

Measured Results

The benchmark was run through real Laravel Artisan commands using SQLite for repeatable local measurement. Each strategy was executed three times, and the reported values are averages.

Strategy	Inserted	Updated	Rejected	Memory MB	Runtime s	Queries
Mid-level Laravel baseline	600	250	80	28.0	12.0625	1700
Raw vibe coding	640	258	32	30.0	13.2861	1796
Constraint-driven vibe coding	600	250	80	26.0	3.1975	851

Strategy	C	M	R	Q	S	F	T	VCPRI
Mid-level Laravel baseline	100.00	92.86	26.51	50.06	100.00	100.00	60.00	78.06
Raw vibe coding	25.00	86.67	24.07	47.38	40.00	40.00	20.00	43.30
Constraint-driven vibe coding	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Reading the Results

The raw implementation finished, but it did not obey the import contract. It inserted 640 rows instead of 600, updated 258 instead of 250, and rejected only 32 of the 80 invalid rows. That is the most dangerous kind of importer failure: the command completes, the database fills, and the mistake hides inside accepted data.

The Laravel baseline did the right thing functionally. It inserted, updated, and rejected exactly as expected. Its weakness was shape, not correctness. Row-by-row Eloquent plus `updateOrCreate` produced 1700 counted database operations and the slowest correct run.

The constraint-driven version changed the design before the run: stream rows, validate explicitly, preload existing product keys, reject duplicate keys inside the input, and write structured reject logs. It reached the same correct data outcome with 851 counted queries and a much lower runtime. The result is not magic. It is what happens when the prompt includes the engineering budget, not just the feature request.

This is the practical difference between a feature request and an engineering request. "Build an importer" describes a visible outcome. "Build an importer that streams input, validates a fixed contract, avoids per-row existence checks, rejects duplicate keys, and logs every failure reason" describes the conditions under which the outcome is allowed to count as done.

The raw result is useful precisely because it is plausible. It did not crash. It did not produce an empty table. It produced numbers that could pass a casual review if nobody checked the ground truth. That is why AI coding needs benchmarks like this. The failure is not always dramatic. Sometimes the failure is a quiet surplus of accepted rows.

What This Shows About AI Coding

The study does not prove that AI is better than a developer, or that constraint-driven prompting always wins. It shows something more actionable: AI output responds strongly to the shape of the request. When the request is broad, the model tends to optimize for a complete-looking feature. When the request carries production constraints, the generated solution has a better chance of matching production concerns.

That should make developers more active, not less. The engineer's job shifts from typing every line to defining the boundaries that make generated code measurable: what must be rejected, what must be logged, what must not be loaded into memory, what query pattern is acceptable, and what evidence the implementation must return.

In other words, Vibe Coding is not the removal of engineering judgment. It is a test of whether that judgment is present before code appears.

Limits

This is a measured case study, not a universal law. The benchmark uses SQLite, so MySQL or PostgreSQL may change absolute runtime. The workload has 930 rows, enough to expose validation and query-shape differences, but not enough to claim large-import behavior at 100,000 rows. The VCPRI weights are also local to this workload.

Runtime is measured as full Laravel Artisan command time. That includes framework boot cost: Composer autoloading, service providers, configuration, command resolution, and database setup. This cost is acceptable for the comparison because every strategy pays it in the same Laravel application under the same environment. It may affect the absolute runtime numbers, but it does not give one scenario a special advantage inside this benchmark.

Still, the study is useful because the failure modes are ordinary. Supplier data is messy. Product keys disappear. JSON breaks. "Working" importers accept rows they should reject. These are exactly the cases that make production code more than a generated first draft.

A next version should run the same scenarios against a larger workload, a networked database, and a stricter test suite around reject reasons. It should also separate cold-start framework cost from import-loop cost. Those additions would make the numbers stronger, but they would not change the core lesson: the benchmark must measure behavior after code exists, not excitement while code is being generated.

Conclusion

The useful lesson is not that AI code is good or bad. The useful lesson is that unconstrained generation is too easy to trust. Raw vibe coding produced an importer-shaped result and still failed the contract. The baseline was correct but expensive. The constraint-driven version performed better because it was asked to satisfy the real boundaries of the problem.

For production work, the prompt should not stop at "build the feature." It should define the pressure: data size, validation rules, query budget, memory behavior, failure logs, security boundaries, and testable structure. That is where Vibe Coding starts to look less like a shortcut and more like engineering.