AI Did Not Replace Junior Developers. It Created Fake Senior Code.

A Laravel Wallet Transfer Study With 16 Production Scenarios

Adnan Mehrat

Abstract

This paper documents a small Laravel project that compares two implementations of the same wallet-transfer feature. The feature is simple: one user sends credits to another user. The test is not simple. The code is checked against validation, authorization, database consistency, failure records, logging, and test behavior.

The project contains two services. `FakeSeniorTransferService` is intentionally written to look organized. It has a service class, a shared return object, readable method names, and a working happy path. It also skips production boundaries: no authorization check, no transaction wrapper, no row locking, weak amount validation, and no durable failure record for several failed attempts. `RealSeniorTransferService` implements those boundaries directly.

The Laravel command `wallet:fake-senior-study` runs 16 scenarios against both services. The measured result was clear: the fake implementation passed 9 scenarios and failed 7, with a Production Readiness Score of 51.7 out of 100. The real implementation passed all 16 scenarios and scored 100 out of 100. These numbers are from the local Artisan command, not estimated values.
Keywords: AI-generated code, Laravel, backend engineering, wallet transfer, authorization, transactions, failure handling, tests

Claim

The risky part of AI-generated backend code is not always bad syntax or broken files. Those are easy to catch. The harder problem is code that looks structured and still misses the rules that matter in production.

In this project, the fake implementation is not written as a joke. It is the kind of code that can survive a quick review. It has a service layer. It returns a typed result object. It checks that the amount is numeric. It rejects negative amounts. It creates a successful transaction record.

That is not enough for wallet logic.

A transfer service has to answer concrete questions. Can the actor spend from the sender wallet? What happens when the sender and receiver are the same user? Is zero a valid transfer? Does a failed transfer leave a database record? Are both wallet rows updated inside one database transaction? Does the code protect the read-modify-write sequence with row locks?

The study is built around those questions. There are 16 scenarios. Each scenario has an expected database state after execution. The command runs both services and scores the result.

External Context

The broader evidence on AI coding tools is mixed. The GitHub Copilot productivity study reported faster completion on a bounded JavaScript HTTP-server task. A later METR study on experienced open-source developers found slower completion when AI tools were allowed in familiar repositories. These results are not contradictory for this paper. They describe different work.

Trust data points in the same direction. Stack Overflow's 2025 Developer Survey on AI reported that more developers distrusted the accuracy of AI tool output than trusted it. The DORA 2025 State of AI-assisted Software Development report also treats AI-assisted development as dependent on the quality of the surrounding engineering system, not just the tool.

This project uses that same framing. It does not measure whether AI is good or bad in general. It measures whether one generated-looking Laravel service survives 16 explicit backend checks.

Project Under Test

The project is named `fake-senior-code-laravel-wallet`. It is a runnable Laravel application with migrations, models, services, a policy, tests, a scoring helper, and an Artisan command.

The domain uses Laravel's default `users` table and two wallet tables:

- `wallets`: one row per user, with a decimal balance.
- `wallet_transactions`: one row per successful or failed transfer attempt.

The transfer method has the same public boundary in both services:

Listing 1: Shared service boundary.

```
public function transfer(
    User $actor,
    int $senderId,
    int $receiverId,
    mixed $amount
): TransferResult;
```

That boundary is useful because it keeps the comparison narrow. The services receive the same actor, sender, receiver, and amount. The difference is in the implementation, not in routing or controller code.

Wallet Transfer Contract

The feature contract is intentionally small. It is not a payment gateway. It does not handle currencies, refunds, ledger reconciliation, or idempotency keys. The point is to test the basic backend rules that should exist before those larger concerns are added.

Item	Source	Rule
Actor	Authenticated user	Must be allowed to spend from the sender wallet.
Sender wallet	Database	Must exist and have enough balance.
Receiver wallet	Database	Must exist and must not be the sender wallet.
Amount	Request input	Must be numeric, greater than zero, and valid to two decimal places.
Balance update	Database write	Debit and credit must happen inside one transaction.
Wallet rows	Database read	Rows should be locked before balance calculation.
Transaction history	Database write	Successful and failed attempts should be recorded.
Failure reason	Database/log	Failed attempts should explain why they failed.

Table 1: Wallet transfer contract used by the study.

Two Implementations

FakeSeniorTransferService

The fake service is short and readable. It also misses several rules.

Listing 2: Fake senior validation: readable, incomplete.

```
if (! is_numeric($amount)) {
    return TransferResult::failure('amount_must_be_numeric');
}

$amount = (float) $amount;

if ($amount < 0) {
    return TransferResult::failure('amount_must_be_positive');
}
```

This accepts `0.00`. It also accepts the actor as a parameter and never checks whether that actor owns the sender wallet. It performs two wallet saves without wrapping them in `DB::transaction`. It does not call `lockForUpdate`. Most failed cases return a failure object but do not create a failed row in `wallet_transactions`.

There is also a self-transfer bug. When sender and receiver are the same user, the service loads the same wallet row twice. It saves the debit through one model instance, then saves the credit through another stale instance. In the test, a self-transfer of 10.00 from a 100.00 wallet leaves the wallet at 110.00.

RealSeniorTransferService

The real service handles the same feature with explicit production checks:

- rejects malformed, zero, negative, and sub-cent amounts;
- rejects transfer to self;
- checks that the actor is the sender;
- applies a wallet policy;
- verifies both wallets exist;
- uses `DB::transaction`;
- reads wallet rows with `lockForUpdate`;
- writes success and failure transaction rows;
- logs failed attempts with a reason and context.

The core database boundary is explicit:

Listing 3: Transaction and row-lock boundary.

```
return DB::transaction(function () use ($senderId, $receiverId, $normalized):
    TransferResult {
    $senderWallet = Wallet::query()
```

```
        ->where('user_id', $senderId)
        ->lockForUpdate()
        ->first();

    $receiverWallet = Wallet::query()
        ->where('user_id', $receiverId)
        ->lockForUpdate()
        ->first();

    // validate balances, debit, credit, and record the transfer
});
```

The implementation is longer because it handles more states. The extra code is not decoration. It exists because the service changes stored balances.

Scenario Set

The command `php artisan wallet:fake-senior-study` creates fresh users and wallets for each case. It runs both services through the same checks and records whether each service met the expected behavior.

The measured scenario count is:

$$S = 16$$

The core production scenarios are:

1. successful transfer;
2. insufficient balance;
3. transfer to self;
4. zero amount;
5. negative amount;
6. missing sender wallet;
7. missing receiver wallet;
8. unauthorized transfer attempt;
9. repeated transfer attempt;
10. transaction history consistency;
11. failure reason logging;
12. balance unchanged after failed transfer.

The remaining four checks cover study-level test coverage and maintainability: happy-path coverage, failure-path coverage, structured result object, and stable service API.

Scoring

The Production Readiness Score is a weighted checklist:

$$\text{PRS} = 0.20V + 0.20A + 0.20D + 0.15F + 0.15T + 0.10M$$

where:

- V: validation;
- A: authorization;
- D: database consistency;
- F: failure handling;
- T: test coverage;
- M: maintainability.

Each category score is calculated from passed checks:

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

Category	Weight	Practical reason
Validation	0.20	Bad money input should fail before state changes.
Authorization	0.20	The actor must not spend from another user's wallet.
Database consistency	0.20	Balances and transaction rows must agree.
Failure handling	0.15	Failed attempts need durable reasons.
Test coverage	0.15	The service needs failure-path tests.
Maintainability	0.10	The API should stay understandable and stable.

Table 2: Production Readiness Score weights.

Measured Results

The project was run locally with:

Listing 4: Measured commands.

```
php artisan test
php artisan wallet:fake-senior-study --json
```

The test suite passed:

- 20 tests;
- 72 assertions.

The study command produced the following result.

Scenario	Fake	Real	Expected behavior
Happy path result	PASS	PASS	Balances and success history match.
Transfer to self	FAIL	PASS	Reject and leave the wallet unchanged.
Zero amount	FAIL	PASS	Reject the transfer.
Negative amount	PASS	PASS	Reject without moving money.
Insufficient balance	PASS	PASS	Fail without changing balances.
Missing sender wallet	FAIL	PASS	Write a failed transaction row.
Missing receiver wallet	FAIL	PASS	Write a failed transaction row.
Unauthorized transfer attempt	FAIL	PASS	Do not let the actor spend another wallet.
Repeated transfer attempt	PASS	PASS	The second transfer fails without overdrawing.
Transaction history consistency	FAIL	PASS	Success and failure are both visible.
Failure reason logging	FAIL	PASS	Store the failure reason.
Balance unchanged on failed transfer	PASS	PASS	Failed transfer leaves balances unchanged.

Table 3: Core scenario results.

Category	Fake Pass/Total	Fake Score	Real Pass/Total	Real Score
Validation	1/3	33.3%	3/3	100%
Authorization	0/1	0%	1/1	100%
Database consistency	4/4	100%	4/4	100%
Failure handling	0/4	0%	4/4	100%
Test coverage	2/2	100%	2/2	100%
Maintainability	2/2	100%	2/2	100%

Table 4: Category scores.

Why Some Scores Match

Some results match across both services. That is expected, and it matters.

The fake service scored 100% in the measured database-consistency category. This does not mean it is safe under production traffic. That category measured sequential checks only: balances after a successful transfer, balances after insufficient funds, balances after a repeated transfer, and balances after a failed transfer. In those cases, the fake service did not overdraw or mutate the checked balances incorrectly.

The same service still lacks `DB::transaction` and `lockForUpdate`. The current benchmark checks a repeated transfer in sequence; it does not run parallel workers against the same wallet. A stronger version of the benchmark should split this category into two parts:

- sequential consistency;
- concurrency safety.

Test coverage and maintainability also match. Both services are tested by the study command. Both expose the same method. Both return `TransferResult`. That is useful because it shows the

Service	Passed scenarios	Failed scenarios	PRS
Fake Senior Code	9	7	51.7 / 100
Real Senior Code	16	0	100 / 100

Table 5: Final study result.

main issue clearly: the fake service does not fail because it has no structure. It has structure. The missing parts are authorization, failure audit records, stricter validation, and database protections.

If the fake service failed every category, the comparison would be less useful. The realistic problem is that it passes enough checks to look acceptable, then fails on production-specific rules.

Reading the Numbers

The fake service passed 9 of 16 scenarios. The passes are not meaningless. The happy path works. Negative amounts are rejected. Basic insufficient-balance behavior leaves balances unchanged. The public API is stable.

The failures are the part that matters for shipping:

- unauthorized actor can transfer from another user's wallet;
- zero amount is recorded as a successful transfer;
- transfer to self can corrupt the wallet balance;
- missing-wallet failures do not create failed transaction rows;
- insufficient-balance failure is not stored with a reason;
- failure context is not logged;
- the update path has no explicit transaction or row lock.

The real service passed all 16 scenarios. That does not make it a complete payment system. It means it satisfies the contract defined in this project.

The most useful result is the split. The fake implementation looks clean and still fails 7 production checks. That is the actual risk. The code is not obviously broken at first glance.

Practical Notes

The benchmark uses SQLite. Check locking behavior and performance again on MySQL or PostgreSQL before using this as a production pattern.

The concurrency scenario is not a full concurrent load test. It checks sequential repeated transfer behavior and verifies that the real service uses transaction and row-lock boundaries. A next version should run parallel requests against the same sender wallet.

The score is local to this feature. The weights are simple and visible. A real wallet system should add more checks: idempotency keys, immutable ledger rows, currency handling, retry behavior, reconciliation, and stronger audit requirements.

The study still catches the main issue. A service can look organized, pass the happy path, and return a clean result object while skipping rules that decide whether it can be trusted.

Conclusion

The measured result is straightforward.

- Fake service: 9 passed, 7 failed, 51.7 / 100.
- Real service: 16 passed, 0 failed, 100 / 100.

The fake service is not bad everywhere. That is why it is a useful example. It passes simple checks and keeps a clean public shape. The failures appear when the test asks about ownership, invalid amounts, failure history, and database boundaries.

For AI-assisted backend work, the prompt should include the production rules, and the project should include tests that verify those rules. Do not stop at a service class and a success response. Check the 16 kinds of behavior that decide whether the feature is safe enough to ship.