GUL'S NOTEBOOK · ENGINEERING · OPEN ACCESS

laravel-agent-guard: why your Laravel AI agents need brakes

Muhammad Gulzaib
https://orcid.org/0009-0000-8709-6989
SynergyFlow Labs · Islamabad, Pakistan

Published	2026-11-25
URL	https://gulzaib.vercel.app/blog/laravel-agent-guard-why-your-laravel-ai-agents-need-brakes
Licence	Creative Commons Attribution 4.0 International (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/

ABSTRACT

AI agents can loop for hours and run up a bill nobody sees. Why I built laravel-agent-guard, a free, open-source safety net for Laravel apps that use agents.

Keywords: laravel, ai, agents, open-source

> **TL;DR**
>
> AI agents work in loops, and a loop that gets stuck keeps spending money while everything looks healthy. **laravel-agent-guard** is a free, open-source package I built for Laravel apps that use AI agents. It stops runs that repeat themselves, gives every customer a spending limit that holds even under heavy load, and switches to a cheaper model near the limit so people get an answer instead of an error page. It's MIT-licensed, so anyone can use it, read it and improve it.

In November 2025, a team left four AI agents running a market research job. Two of them got stuck handing the same work back and forth. They kept at it for 11 days. No error fired and nothing crashed. A billing alert finally caught it, after about `47,000 had gone to the AI provider [\[1\]](#ref-loop47k "Vectara (awesome-agent-failures), Case study: the` 47,000 LangChain A2A infinite loop (2026)").

When I read their write-up, the cause wasn't exotic. Nothing in the system had a budget per agent, and nothing decided when the agents should stop. That story is the reason this package exists.

Agents are useful, and they don't know when to stop

A normal AI feature asks one question and gets one answer. An **agent** works more like a junior employee with a to-do list. It asks the AI what to do next, uses a tool (search the database, read a file, call an API), looks at the result and asks again. Each round trip is one

step. Laravel's official AI package lets you build agents like this in a few lines [2], which is great.

The catch is in how agents pay. Every step sends the whole conversation so far back to the AI provider, so each step costs a little more than the one before. When an agent gets confused and keeps repeating the same search, the bill climbs faster with every step.

From the outside, a stuck agent looks fine. The provider answers every request. Your queue says the job is still running. Your uptime monitor stays green.

Three ways it goes wrong in a real app

I see three problems again and again when I think about agents inside a Laravel product with paying customers.

The agent gets stuck in a loop. It asks for the same thing, gets the same answer and asks again. Laravel lets you set a maximum number of steps, which helps, but the agent still pays for every step up to that limit. It can't tell that it's going in circles.

Many workers spend from the same budget at once. Busy apps run AI jobs on several background workers in parallel. Picture ten workers that each check the budget at the same moment, each see "80% used, there's room for one more" and each go ahead. Together they blow straight past the limit. A simple check in your code can't prevent that.

The customer sees an error. When a limit is reached, the easy reaction is to throw an error. Your bill is safe, but the customer who was halfway through a task now sees a broken page. That is often your most active customer, because they're the one using the AI the most.

Why I built it

I build multi-tenant software: one Laravel app, many customers, each with their own data. At Oxygen LMS, every school is a customer in the same system (how I set that up), and AI work runs in the background (how I keep it from slowing the app down).

So one question kept bothering me. If one school's agent gets stuck at 2 a.m., what stops it from using up the AI budget meant for every other school?

Laravel's AI package already has a good place to answer that. It lets you run your own code before every single step an agent takes. That's exactly where a guard should sit: after the agent decides what to do next, and before the request goes out and costs money. I looked at what already existed, found useful pieces, and didn't find one package that put loop detection, shared budgets and a graceful fallback together at that point. So I wrote it.

What it does, in plain words

laravel-agent-guard checks each step of an agent before it's sent.

1. **Agent picks a step**: what to do next
2. **Guard checks it**: stuck? over budget?
3. **Step goes out**: or is stopped
4. **Guard learns**: real cost, repeats

The guard decides before the step costs anything, then records what it really cost.

It notices when an agent repeats itself

The guard keeps a short memory of what the agent has done in this run. If the agent asks for the same thing three times, or its conversation keeps growing while nothing new comes back, the guard ends the run. The agent hands back what it has done so far, and no more requests go out. Small tricks, like asking for "page 3" instead of "page 2" of the same thing, don't fool it.

It gives every customer a real spending limit

You set limits for one run, one agent, one customer or one tool, counted in tokens or in dollars. Each limit has two levels: a soft limit and a hard limit. The guard sets the money aside before each step, in one go, inside Redis, so two workers can't both squeeze through the same gap. After the step, it records what the provider actually charged.

It switches to a cheaper model before it gives up

When a customer gets close to their soft limit, the guard can move the rest of the run to a cheaper, faster model and turn off expensive tools. The customer still gets an answer, a simpler one, and your costs stop climbing. Only at the hard limit does the run actually stop.

It has an emergency stop

If something looks wrong in production, one command stops every agent for one customer, or for everyone, at its very next step. You don't need to deploy anything.

Safe to try on a live app

The thing I cared about most was that adding it shouldn't break anything. So it starts in **observe mode**. It watches every step and reports what it would have done ("this run looks like a loop", "this customer just crossed their soft limit"), but it doesn't change or stop anything.

You leave it like that for a week, look at the reports, adjust the limits to fit your real traffic,

and only then switch enforcement on. Developers can also test it in their automated tests without setting up Redis.

The package includes a small simulation of a stuck agent: five steps, all asking for the same search. Without the guard, all five steps run. With it, the run stops after the third identical request, and it uses about half the tokens. Real loops usually run much longer than five steps, so in practice the saving is bigger.

What it isn't

It's a safety net for spending and runaway behaviour. It doesn't decide what an agent is allowed to do. For that, Laravel's AI package already supports asking a human to approve a risky action [2], and other community packages handle permissions. Use them together.

It's also new. Version 1 came out in November 2026. It has automated tests, including one that runs several processes against the same budget at once, but it hasn't had months of other people's production traffic yet. That's one more reason to start in observe mode.

Why it's open source

I kept it MIT-licensed and free because nearly everything I've built stands on free Laravel packages, including Laravel's own AI tools. This is a small way to give some of that back.

There's a practical reason too. Every team that runs agents ends up writing some kind of spending guard, usually right after their first surprise bill, and usually alone. One shared version that many people read, test and fix will be safer than a hundred private ones. Some parts can only get better with help:

- **Prices.** AI providers change their prices often. When a number goes stale, one pull request fixes it for everyone.
- **Loops it missed.** If your agent got stuck and the guard didn't notice, a cleaned-up record of what the agent did is the most useful bug report you can send.
- **Other setups.** Different queues, different ways of naming customers, different storage. I can't test them all alone.

Where to find it

The code is on GitHub at imgul/laravel-agent-guard, and the package is on Packagist. For a Laravel developer, installing it is one line:

```
composer require imgul/laravel-agent-guard
```

This post is the "why". Next I'll write the developer posts: how to set it up, how each check works and how to tune it for your app. If you're still deciding what your agents should do in

the first place, start with what makes an AI agent useful for real work.

> **Key takeaways**
>
> - An AI agent pays more for each step it takes, so a stuck agent gets more expensive the longer it runs, and it rarely crashes.
> - A maximum step count limits how long a run can go. It doesn't notice that the agent is going in circles.
> - Spending limits need to hold when many workers spend at once, which a simple check in your code can't guarantee.
> - Falling back to a cheaper model near the limit keeps customers working instead of showing them an error.
> - laravel-agent-guard is free and MIT-licensed. Try it in observe mode, and send a pull request when you find something to fix.

Questions readers ask

What is laravel-agent-guard?

It's a free, MIT-licensed package for Laravel apps that run AI agents. It watches every step an agent takes, stops runs that keep repeating themselves, keeps each customer's AI spending under a limit and switches to a cheaper model near that limit instead of showing an error.

Why can an AI agent cost so much money?

An agent works in steps, and each step sends the whole conversation so far back to the AI provider. If the agent gets stuck repeating itself, every new step costs more than the last, and nothing crashes, so nobody notices until the bill arrives.

Do I need to change my app to try laravel-agent-guard?

No. It starts in observe mode, where it only records what it would have done. You turn on enforcement when you've seen how it behaves with your real traffic.

References

1. Vectara (awesome-agent-failures). Case study: the $47,000 LangChain A2A infinite loop, 2026.
2. Laravel. Laravel AI SDK (Laravel 13.x documentation), 2026.

CITE AS

Gulzaib, M. (2026, November 25). laravel-agent-guard: why your Laravel AI agents need brakes. Gul's Notebook. https://gulzaib.vercel.app/blog/laravel-agent-guard-why-your-laravel-ai-agents-need-brakes

You may share and adapt this text for any purpose, including commercially, as long as you credit the author and link to the original. Licence: https://creativecommons.org/licenses/by/4.0/