# Handling Zero Cost Failed Requests in Multi-Provider AI APIs

When you build on top of an AI marketplace API, you get access to many models through a single endpoint. The convenience is real, but so is the complexity. One of the quieter challenges that teams face is dealing with failures that don't cost money directly. These are "zero cost failed requests", calls that return an error or time out but never hit a paid provider. At first glance they seem harmless because your credit card isn't charged. But they still consume time, degrade user experience, and can cascade into bigger problems if you don't handle them properly.

## Why a Failed Request Isn't Free

It's easy to think that if you aren't billed, nothing was lost. In reality, every failed request costs you in other ways. Your application spends CPU cycles and memory waiting for a response that never comes. Your user sits through a spinner or an error message. If the failure happens during a critical workflow, the user might abandon the task entirely. Over time, a high rate of failed requests erodes trust and makes your service feel unreliable, even when the root cause is upstream.

On a marketplace platform like UberLLM, where multiple sellers compete to serve a request, a failure can happen for many reasons. A seller's model might be overloaded, their endpoint might be down, or the request itself might be malformed. The platform absorbs the cost of the failed call, but your application still bears the latency and the logic overhead of retrying or falling back.

What makes "zero cost failed requests" especially tricky is that they are invisible in your billing dashboard. You have to actively monitor them through logs and metrics. Without that visibility, you might not realize that a particular seller or model is failing more often than others, or that a certain time of day sees a spike in errors.

## Common Causes in a Multi-Seller Setup

Understanding why failures happen helps you design better handling. Here are the most frequent culprits:

- **Seller timeout:** A seller's model takes too long to respond. The platform eventually gives up and returns an error. This is the most common type of zero cost failure.
- **Invalid request:** Your payload doesn't match what the seller expects. Maybe you sent a field they don't support, or the model name is misspelled. The seller rejects the

request before processing it.

- **Model unavailability:** A seller's endpoint is down for maintenance or has reached its rate limit. The platform can't route the request, so it returns an error immediately.
- **Content filtering:** Some sellers apply stricter content policies. A request that passes through one seller might be blocked by another. The platform reports this as a failure without charging you.
- **Routing failure:** The marketplace can't find a seller that meets your criteria (e.g., a specific model version or latency requirement). No attempt is made, and no cost is incurred.

Each of these cases requires a different response. A timeout might be worth retrying with a different seller. An invalid request means you need to fix your payload. A content filter might require you to rephrase the input. Knowing which case you're dealing with is the first step to recovery.

## Strategies for Handling Failures Gracefully

You can't prevent every failure. But you can design your system to absorb them without breaking the user experience. Here are practical approaches that work well in production.

### Retry with Exponential Backoff

If a request fails due to a transient issue like a timeout or a temporary overload, retrying after a short delay often succeeds. Use exponential backoff, wait 1 second, then 2, then 4, and so on, to avoid hammering the platform. Add jitter to prevent all your retries from hitting at the same time. Set a maximum retry count so you don't loop forever.

### Fallback to Alternative Sellers

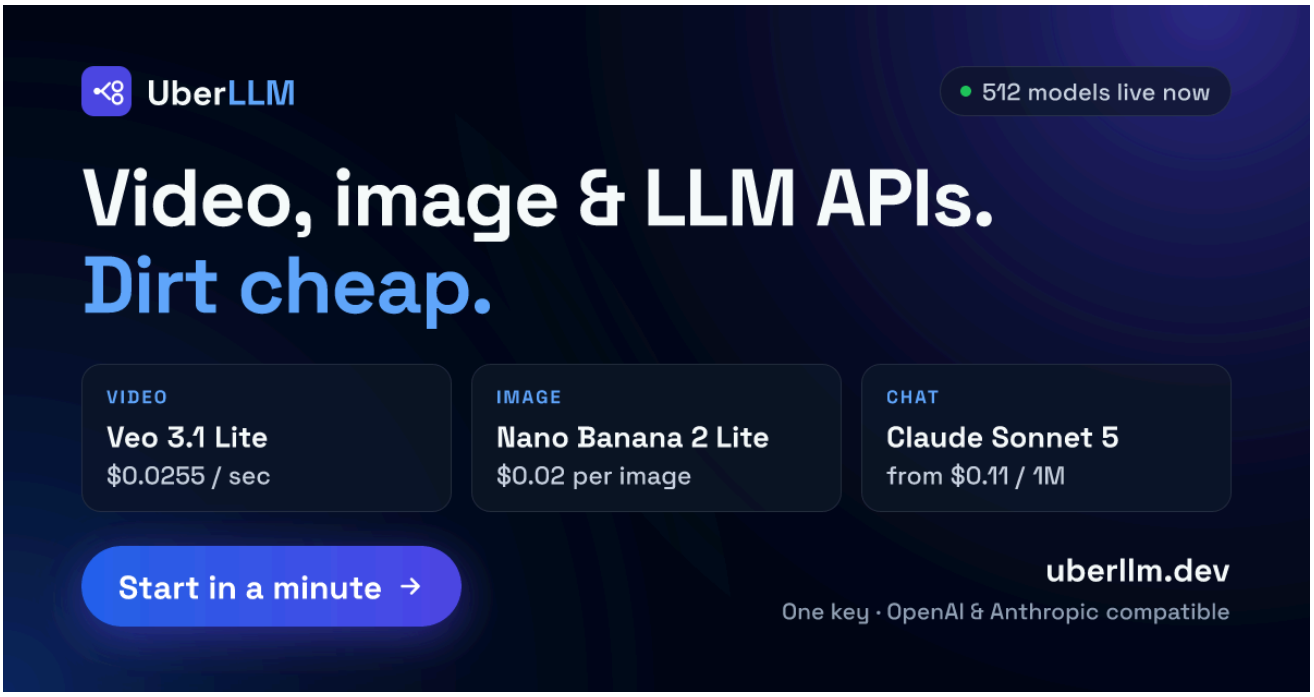
A marketplace gives you multiple sellers for the same model. If one seller fails, try another. This is where "uberllm zero cost failed requests" becomes a feature rather than a bug. Because the failed call didn't cost anything, you can afford to try several sellers in sequence until one succeeds. The key is to track which sellers fail often and deprioritize them in future routing decisions.

### Use Timeouts Wisely

Set a reasonable timeout for each request. If the seller doesn't respond within that window, treat it as a failure and move on. A timeout that is too short causes false positives; one that is too long makes your users wait. Start with a timeout that matches the model's typical response time and adjust based on observed behavior.

### Log and Analyze Failure Patterns

Every failed request is a data point. Log the seller, the model, the error type, and the timestamp. Aggregate these logs to find patterns. Maybe seller A fails during peak hours but seller B stays stable. Maybe certain prompts consistently trigger content filters. Use this information to tweak your routing logic or to pre-filter requests before sending them to the marketplace.




## The Role of the Marketplace Platform

A good marketplace doesn't just route requests; it also helps you manage failures. Platforms like UberLLM provide error codes that distinguish between a seller timeout, a routing failure, and a content filter. They also offer fallback routing options that automatically retry with a different seller when the first one fails. By leveraging these features, you can reduce the impact of zero cost failed requests on your application.

That said, the platform can only do so much. Ultimately, your application needs to be resilient to failures at every layer. A robust integration treats the marketplace as an external dependency that can and will fail. You should never assume a request will succeed, and you should always have a plan for when it doesn't.

## Measuring What Matters

If you only track successful requests, you're flying blind. You need to measure failure rates, error types, and latency distributions. Set up alerts for when the failure rate exceeds a

threshold, or when a particular seller starts failing more than usual. This proactive monitoring lets you catch problems before they affect your users.

One metric worth watching is the ratio of zero cost failures to total failures. A high ratio means many requests are failing before reaching a seller. This could indicate a configuration issue on your side, or it could mean the marketplace is having trouble finding a suitable seller. Either way, it's a signal that something needs attention.

## Practical Example: Handling a Timeout

Let's say you send a request to generate a summary. The platform routes it to a seller, but that seller doesn't respond within 10 seconds. The platform returns a timeout error and does not charge you. This is a classic "uberllm zero cost failed requests" scenario. Instead of showing an error to the user, your code could retry the same request with a different seller. If the second seller responds in 3 seconds, the user never knows the first attempt failed. The only cost was a few extra seconds of latency.

This pattern works because the failed request was free. If the marketplace charged for every attempt, you would think twice about retrying. But since it doesn't, you can be aggressive with fallbacks. The result is a more reliable user experience without extra cost.

## Common Pitfalls to Avoid

Even with good intentions, it's easy to make mistakes. Here are some pitfalls to watch for:

- **Retrying too aggressively:** Sending retries without backoff can overwhelm the platform and lead to more failures. Always add a delay.
- **Ignoring error codes:** Different errors need different responses. Treating a timeout the same as a content filter leads to wasted retries.
- **Not distinguishing transient vs. permanent failures:** A model that doesn't exist will never work. Don't retry it. A model that is temporarily overloaded might work after a delay.
- **Assuming all sellers are equal:** Sellers have different reliability, latency, and content policies. Route based on historical performance, not just price or model name.
- **Neglecting user feedback:** If a request fails after all retries, tell the user clearly. Offer alternatives or suggest rephrasing the input.

## Conclusion

Zero cost failed requests are a reality of working with any multi-provider API. They aren't free in the broader sense because they consume time and degrade experience. But they do offer a

unique advantage: you can try again without financial penalty. By understanding the causes, implementing smart retry logic, and monitoring failure patterns, you can turn these failures into a strength. The phrase "uberllm zero cost failed requests" captures this idea well, it's not about avoiding failures entirely, but about handling them in a way that keeps your application reliable and your users happy. With the right approach, you can build a system that gracefully handles the inevitable bumps in the road.