How to Move From Browser Tests to Reliable AI API Calls

Start with the browser version of the task

Before touching code, make the task clear in a browser. Write down what goes in, what comes out, and what a useful result looks like. If the job is image generation, note the prompt, aspect ratio, style limits, and the kind of file you expect. If it is text work, note the tone, length, format, and any facts that must stay unchanged. If audio or video is involved, describe duration, format, voice, scene, and any constraints that would make the result unusable.

This step matters because an API call is stricter than a browser form. A form may hide defaults, fill blanks for you, or give a friendly warning. An endpoint usually does none of that. The clearer the browser version is, the easier it is to turn the same intent into a request schema. Watch for hidden choices too, such as a default model, a default language, or a size setting that changes when another option is selected.

Map the browser controls to request fields

Once the task is written down, match each visible control to a field in the request. A prompt box becomes a text field. A model picker becomes a model name. A file upload becomes either a file part or a hosted file reference, depending on the service. A slider becomes a numeric value with a valid range. This is where many beginners get stuck, because the browser may show friendly labels while the API expects short, exact field names.

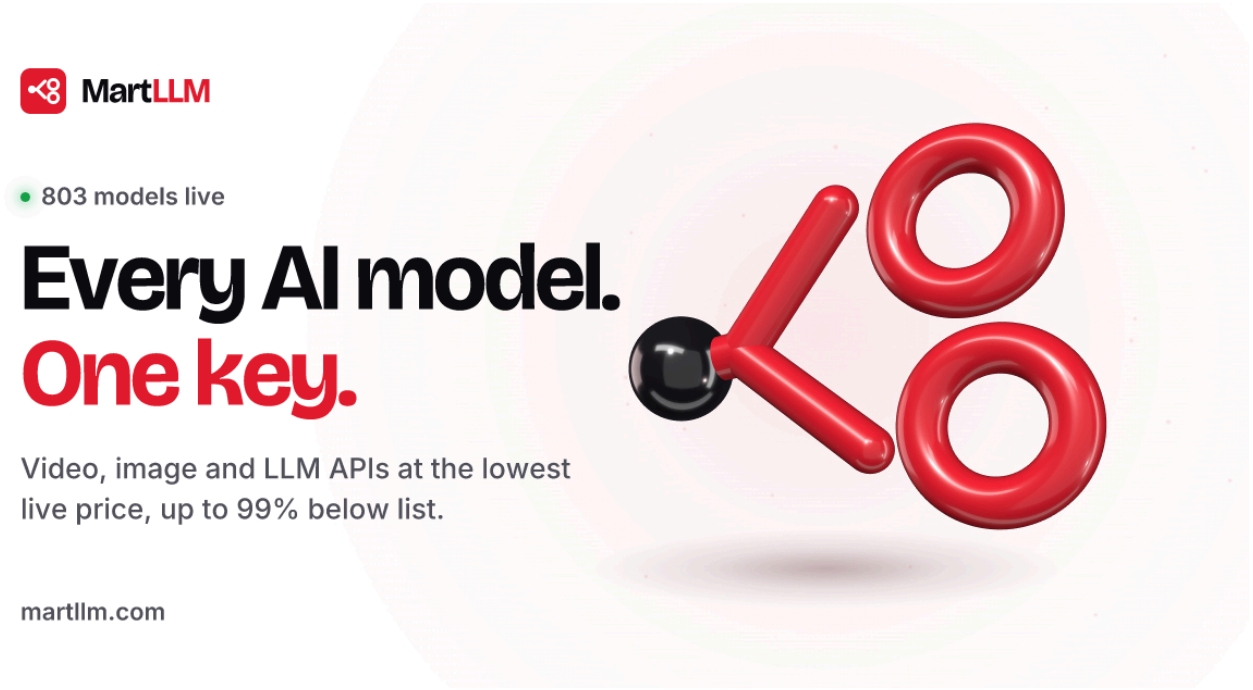
Keep a small table during this stage. On one side, write the browser label. On the other, write the matching field name, value type, and whether it is required. Mark fields that have a narrow range, such as length, resolution, temperature, or duration. Also mark anything that depends on something else, such as a voice that only works for a certain language or a video setting that only applies to one model. That table becomes your reference when the request fails.

A unified AI API is useful here because it gives one pattern for work that might otherwise feel unrelated. Text, image, audio, and video tasks can be approached through the same habits: set the model, send a clear request body, handle the response, and keep track of job status. The content changes, but the shape of the work stays familiar.

Set up authentication without exposing the API key

Next, create a safe place for the API key. Treat it like a password. Keep it out of source files, screenshots, public notes, and client-side code. A common approach is to store it as an

environment variable on the machine or in the deployment platform, then read it from there at runtime. The code can check that the value exists before sending any request.



Watch for two mistakes at this stage. The first is putting the key directly into a frontend file, where visitors may be able to read it. The second is copying the key into example files that later get shared or committed. A clean pattern is to keep example code with a placeholder and load the real value only from the environment. If a key has been exposed, replace it rather than hoping nobody noticed.

Authentication also affects how you debug. When a request fails, check whether the error is about identity or about the request itself. A missing or invalid key usually produces a clear authentication error. A valid key with a malformed body usually produces a validation error. Knowing which one you have saves a lot of random changes.

Make the smallest possible request first

Do not begin with the full production workload. Send one short, simple request that matches the browser example as closely as possible. Use a small text prompt, a low-cost test setting where available, or a short output length. The aim is to confirm the basics: the base URL is correct, the endpoint exists, authentication works, and the response can be parsed.

Keep the first request boring. Use plain values, not clever templates. Avoid long prompts, unusual characters, nested options, and large files. If the first call works, save the request body as a reference. If it fails, change one thing at a time. Changing several fields at once makes it hard to learn what caused the problem.

Pay attention to status codes and error messages. Some failures mean the request never reached the service. Some mean the service received it but rejected a field. Others mean the job was accepted but failed while processing. These are different problems, and they need different fixes. A clear log of the request, status code, and response body turns debugging into a process rather than a guess.

Read the response before writing more code

After the first successful call, inspect the response carefully. Find where the actual result lives, along with any job identifier, status field, creation time, usage details, or error object. Some APIs return the result immediately. Others accept the job and require a separate status check. For media work, the result may be a file link or a temporary output reference rather than raw content in the response body.

Write down the response shape in plain language. For example, note that the result is under one object, the status appears in another field, and errors use a specific structure. This habit helps when you later build code that handles many response types. It also helps you avoid brittle parsing, where a small response change breaks the whole workflow.

For MartLLM, the context to keep in mind is a marketplace for video, image, audio, and text APIs through one key, with expiring-free credit and no billing for failed jobs. That context does not remove the need for careful error handling. It simply makes the testing path more coherent, since the same key and general approach can be used while moving between media types.



Add validation before adding features

Once the basic call works, add checks around it. Validate the fields your code sends. Reject empty prompts, unsupported file types, numbers outside the allowed range, and missing required options before a request is made. This protects you from wasted calls and gives clearer errors to whoever uses your application.

Validation should also cover the response. Confirm that the expected fields exist before reading them. If a job is still processing, handle that state instead of treating it as a finished result. If the response contains an error object, log enough detail to understand it without exposing sensitive values. Good validation is quiet when things work and very helpful when they do not.

Watch for the temptation to build a large wrapper too early. A thin wrapper around the first few endpoints is often better. It can hold the API key lookup, request headers, timeout settings, and basic error handling. Later, if the project grows, an SDK or shared client can make the code easier to maintain. Start with clarity, then add abstraction when repetition becomes real.

Handle waiting, retries, and rate limit errors

Production traffic is different from a single browser test. Requests may arrive at the same time, some jobs may take longer, and temporary errors may appear. Build a waiting strategy that matches the task. For quick text calls, a short timeout may be enough. For media generation, a job may need a status check after a delay. Avoid checking status in a tight loop, since that creates noise and wastes resources.

Retries need care. Retry a temporary network error or a rate limit response with a sensible delay. Do not blindly retry a validation error, an authentication failure, or a request that clearly violates the request schema. Repeated bad requests will not become good through persistence. If a job fails after processing begins, treat it as a failed output and decide whether the user should retry, edit the prompt, or choose another setting.

Rate limit handling should be visible in the design. If your application accepts many requests, queue them or slow them down rather than letting every call fail at once. Show a clear message when a limit is reached. Internally, log the response details so the cause can be identified later. A small amount of pacing is often cheaper than a flood of failed attempts.

Move secrets and settings into deployment

When the workflow is stable locally, prepare it for deployment. Move the API key to the deployment environment and confirm that the application reads it correctly. Keep the base URL and model choices in configuration if they vary between environments. Use separate keys or separate environments where practical, so a test change does not disturb a live workflow.

Check the deployment for logging habits before going live. Logs are useful, but they should not contain the API key, private user content, or full file references that should stay restricted. Log request ids, status codes, field names, and short error summaries. If a user reports a problem, those details make it possible to trace the issue without exposing sensitive material.



Also decide what the user sees while work is in progress. A clear pending state, a failure message, and a retry path are better than a blank screen. Media tasks can take longer than text tasks, so avoid assuming every call will finish inside one short page request. Background jobs, status polling, or delayed callbacks may be more suitable depending on the application.

Test the production path with real boundaries

The final stage is to test the full path, not just the API call. Send a request through the same interface users will use, with the same validation, authentication, waiting, error handling, and output display. Try ordinary input, empty input, very long input, unsupported files, and repeated submissions. The aim is not to break the service. The aim is to see whether your own application behaves sensibly when conditions are less than perfect.

Watch the handoff points. A browser form may trim spaces while your code does not. A file upload may succeed but produce a type the endpoint rejects. A status check may return a finished job before the output is ready to display. A user may refresh the page while work is pending. These are normal production conditions, and each one should have a planned response.

Keep a short record of expected behaviors before launch. For each main task, note the happy path, the likely failure path, and what the user should see in both cases. When the record matches real behavior, the move from browser testing to production calls stops feeling like a leap. It becomes a careful transfer of the same task into a more controlled, repeatable form.