API Error Diagnosis and Monitoring Tool

Fernando Djingga
26 June 2025

Introduction

I created a lightweight node.js command-line tool that probes API endpoints, diagnoses common failure modes (timeouts, DNS/host resolution, client 4xx, server 5xx, and slow responses), then produces two outputs per run:

1. report.csv – a machine-readable CSV ready for spreadsheets, BI dashboards, or ticketing systems

2. report.html – a clean, browser-ready summary for quick human triage and screenshots

With only standard Node.js features (built-in fetch and 'AbortController') and no external runtime dependencies, my project delivers practical value: it turns raw API checks into actionable diagnostics with latency measurements, severity labels, and simple recommendations. My aim is to make incident triage and API health checks faster, more consistent, and easier to hand off to product/support teams.

High-Level Implementation

My tool is config-driven. A JSON file defines a list of endpoints (name, method, URL, optional headers/body, and an optional expected status). The CLI loads this config and runs a concurrent probe: each endpoint is requested with a timeout and limited retries (fixed backoff). For each response (or error), my tool measures latency, inspects the status/body, and runs a small diagnosis engine that assigns an issue label and severity with a suggested action.

I tune the concurrency where by default, all endpoints in the test set are probed in parallel (set at pool size 5). This keeps total wall-time near the slowest endpoint while giving a full snapshot of API health. After probing, results are written to two outputs: a normalized CSV for analysis and import, and a minimal HTML report for quick review and sharing. The CLI prints a concise run summary indicating how many endpoints were OK and how many had issues, along with the output paths.

Coding Implementation

My program is split into a small CLI (main.js) and report writers (reporters.js).

The CLI is responsible for parsing flags (--config, --concurrency, --retries, --timeout, output file names), loading endpoints, running the probe with the configured pool size, and writing the reports. The probe function creates an 'AbortController' for each request and enforces a per-attempt timeout. On failure, it retries with a simple fixed backoff (set at 500 ms * attempt). Each attempt captures 'status', 'latencyMs', a short 'response_snippet', and any thrown error for the diagnosis step.

The diagnosis engine maps inputs to an issue and severity. Timeouts are labeled ‘Timeout (High)’ with guidance to increase client timeout, add retries/backoff, and profile server latency. DNS/host resolution failures are labeled ‘DNS/Host Resolution Failure (High)’ with guidance to check DNS/VPC/peering and environment base URLs. Server 5xx are labeled ‘Server Error (5xx) (High)’ with guidance to inspect logs and upstream dependencies. ‘429’ rate limits are labeled ‘Rate Limited (429) (Medium)’ with guidance to implement exponential backoff/jitter or request higher limits. ‘401/403’ are labeled ‘Auth/Permission Error (High)’ with guidance to verify tokens/keys/scopes are time skew. Other 4xx are labeled ‘Client Error (4xx) (Medium)’ with a note to validate payloads and headers. Successful 2xx with high latency are labeled ‘Slow Response (Medium)’; fast 2xx are labeled ‘OK (Low)’.

The CSV writer outputs a stable column set: ‘timestamp’, ‘name’, ‘method’, ‘url’, ‘status’, ‘ok’, ‘latency_ms’, ‘issue’, ‘severity’, ‘expected_status’, ‘attempts’, ‘error_message’, ‘response_snippet’. The HTML writer generates a portable single-file report with color-coded rows (OK, Medium, severity, High severity) and a short legend for quick scanning.

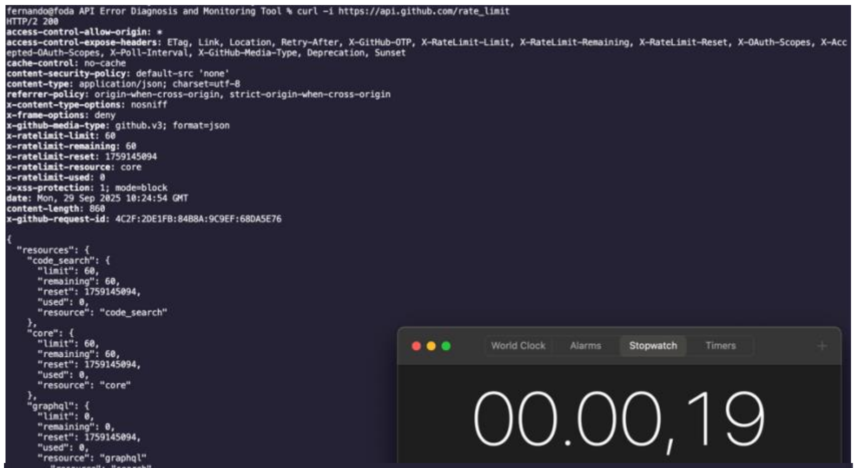
To ensure correctness, I included a Jest test suite that validates diagnosis branches (Timeout, DNS, 4xx/5xx, Slow, OK), the integration of the runner with a mocked fetch, and the CSV formatting. For readability, I refactored the reporters and CLI to a simple JavaScript style: explicit if/else checks instead of chaining, and straightforward variable assignments so the code is easy to follow at a college level.

The instructions to run my program are found in README.md.

Project Results

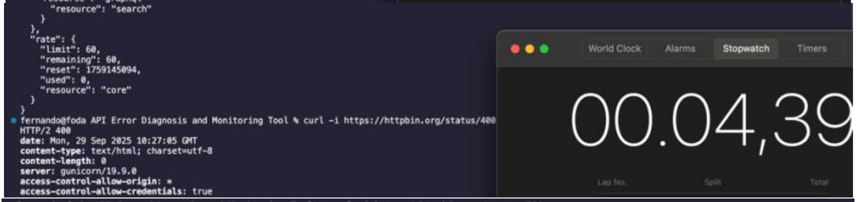
I measured both manual and automated diagnosis on a 5-endpoint demo set (Github 200 OK, httpbin 400, httpbin 500, httpbin delay/2, and a DNS-failure host). Manual checks were done with curl, timing the end-to-end human process for each endpoint: send request → read headers/body → interpret status → note diagnosis. The automated run used the CLI with a concurrency of 5, timeout at 8000 ms, and 1 retry.

The following are the results of manually doing the 5 endpoints individually:

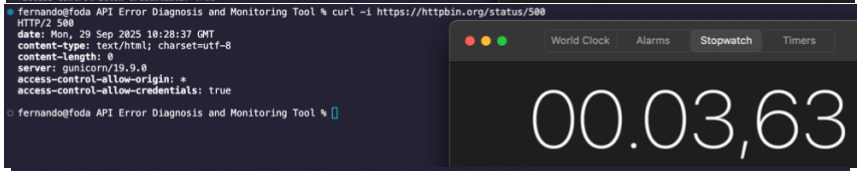


```
fernando@foda API Error Diagnosis and Monitoring Tool % curl -i https://api.github.com/rate_limit
HTTP/2 200
access-control-allow-origin: *
access-control-expose-headers: ETag, Link, Location, Retry-After, X-GitHub-OTP, X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-OAuth-Scopes, X-Acc
epted-OAuth-Scopes, X-Poll-Interval, X-GitHub-Media-Type, Deprecation, Sunset
cache-control: no-cache
content-security-policy: default-src 'none'
content-type: application/json; charset=utf-8
referrer-policy: origin-when-cross-origin, strict-origin-when-cross-origin
x-content-type-options: nosniff
x-frame-options: deny
x-github-media-type: github.v3; format=json
x-ratelimit-limit: 60
x-ratelimit-remaining: 60
x-ratelimit-reset: 1759145094
x-ratelimit-resource: core
x-ratelimit-used: 0
x-xss-protection: 1; mode=block
date: Mon, 29 Sep 2025 10:24:54 GMT
content-length: 860
x-github-request-id: 4C2F:2DE1FB:84B8A:9C9EF:68DA5E76

{
  "resources": {
    "code_search": {
      "limit": 60,
      "remaining": 60,
      "reset": 1759145094,
      "used": 0,
      "resource": "code_search"
    },
    "core": {
      "limit": 60,
      "remaining": 60,
      "reset": 1759145094,
      "used": 0,
      "resource": "core"
    },
    "graphql": {
      "limit": 0,
      "remaining": 0,
      "reset": 1759145094,
      "used": 0,
      "resource": "graphql"
```

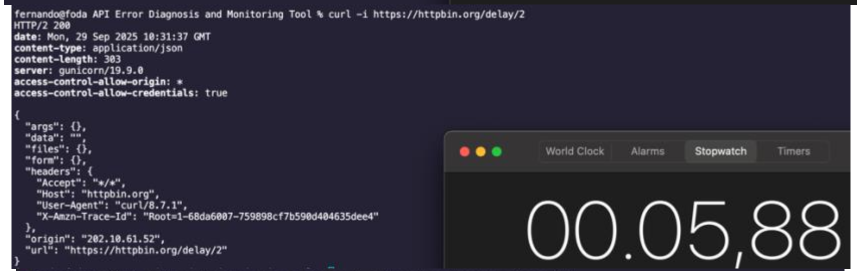


```
        "resource": "search"
      }
    },
    "rate": {
      "limit": 60,
      "remaining": 60,
      "reset": 1759145094,
      "used": 0,
      "resource": "core"
    }
  }
fernando@foda API Error Diagnosis and Monitoring Tool % curl -i https://httpbin.org/status/400
HTTP/2 400
date: Mon, 29 Sep 2025 10:27:05 GMT
content-type: text/html; charset=utf-8
content-length: 0
server: gunicorn/19.9.0
access-control-allow-origin: *
access-control-allow-credentials: true
```



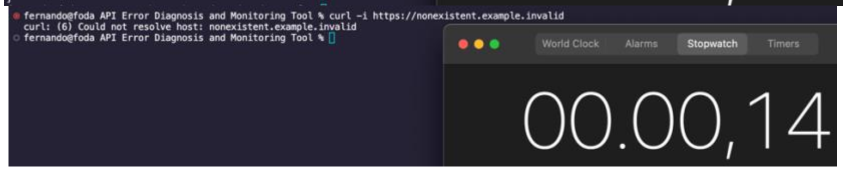
```
fernando@foda API Error Diagnosis and Monitoring Tool % curl -i https://httpbin.org/status/500
HTTP/2 500
date: Mon, 29 Sep 2025 10:28:37 GMT
content-type: text/html; charset=utf-8
content-length: 0
server: gunicorn/19.9.0
access-control-allow-origin: *
access-control-allow-credentials: true

fernando@foda API Error Diagnosis and Monitoring Tool % 
```



```
fernando@foda API Error Diagnosis and Monitoring Tool % curl -i https://httpbin.org/delay/2
HTTP/2 200
date: Mon, 29 Sep 2025 10:31:37 GMT
content-type: application/json
content-length: 303
server: gunicorn/19.9.0
access-control-allow-origin: *
access-control-allow-credentials: true

{
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/8.7.1",
    "X-Amzn-Trace-Id": "Root=1-68da6007-759898cf7b590d404635dee4"
  },
  "origin": "202.10.61.52",
  "url": "https://httpbin.org/delay/2"
}
```

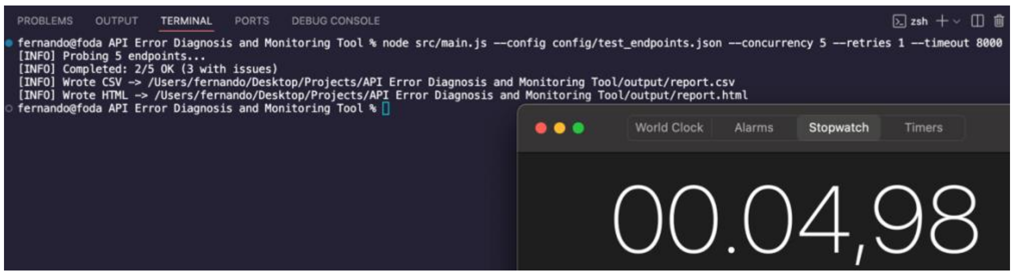


```
fernando@foda API Error Diagnosis and Monitoring Tool % curl -i https://nonexistent.example.invalid
curl: (6) Could not resolve host: nonexistent.example.invalid
fernando@foda API Error Diagnosis and Monitoring Tool % 
```

$$0.19 + 4.39 + 3.63 + 5.88 + 0.14 = 14.23$$

The manual time is about 14.23 seconds for the 5 endpoints.

The following is the results of my automated diagnosis tool:



```
fernando@foda API Error Diagnosis and Monitoring Tool % node src/main.js --config config/test_endpoints.json --concurrency 5 --retries 1 --timeout 8000
[INFO] Probing 5 endpoints...
[INFO] Completed: 2/5 OK (3 with issues)
[INFO] Wrote CSV -> /Users/fernando/Desktop/Projects/API Error Diagnosis and Monitoring Tool/output/report.csv
[INFO] Wrote HTML -> /Users/fernando/Desktop/Projects/API Error Diagnosis and Monitoring Tool/output/report.html
fernando@foda API Error Diagnosis and Monitoring Tool % 
```

The automated program time is 4.98 seconds.

Based on report.csv, my tool produced a success rate of 2/5 endpoints OK/
My tool produced failures of 3/5 (one 400 client error, one 500 server error, one DNS/Network error). My tool has an overall severity mix of 1 Low, 2 Medium, and 2 High. The slow 2xx response (httpbin delay/2) was flagged as 'Slow Response (Medium)'.

My improved time from 14.23 seconds to 4.98 seconds shows that my automated tool completes the task 65% faster.

$$14.23 - 4.98 = 9.25$$

The improvement was by 9.25 seconds.

$$9.25 \div 14.23 = 65\%$$

Conclusion and Project Significance

At a small scale, both manual and automated total time are short, but my automated approach is still about 65% faster while producing standardized reports in CSV and HTML form with consistent language and severity specification. The benefit scales with endpoint count where manual time grows linearly with the number of checks; my tool's wall-time remains close to the slowest request because of concurrency. So, for larger sets (more than 20 endpoints), the relative triage time savings become more pronounced, while still maintain the output that simplifies creating the report.

My approach combines concurrent probing, timeouts and retries, and finally uses the HTTP responses to create actionable and uniform findings that are accessible either as a technical spreadsheet report, or HTML for a quick reveiew. My project has benefits IT and application support, QA/UAT, technical support (SaaS/Fintech), customer success and implementation, and even for tech or IT auditors. It saves time my compressing wall-time to near the slowest endpoint, with no manual per-request reading/typing required. It increases consistency and accuracy by providing clear security labels and recommendations. It is also scalable for larger number of endpoints and is easily integrated. Finally, it is simple and usable to run and use, as well as modifying to fit the specific needs of a relatively wide range of technical environments.

As a fresh graduate, I designed this project with the purpose of simulating useful and applicable automation tools for real-world operations such as reproducing diagnostics, providing structured outputs and measurable improvement over manual checks. In technical environments where uptime, latency, fast incident response matter greatly, my tool offers teams the ability to detect and classify different issues quickly and transparently in a structured and standardized way.