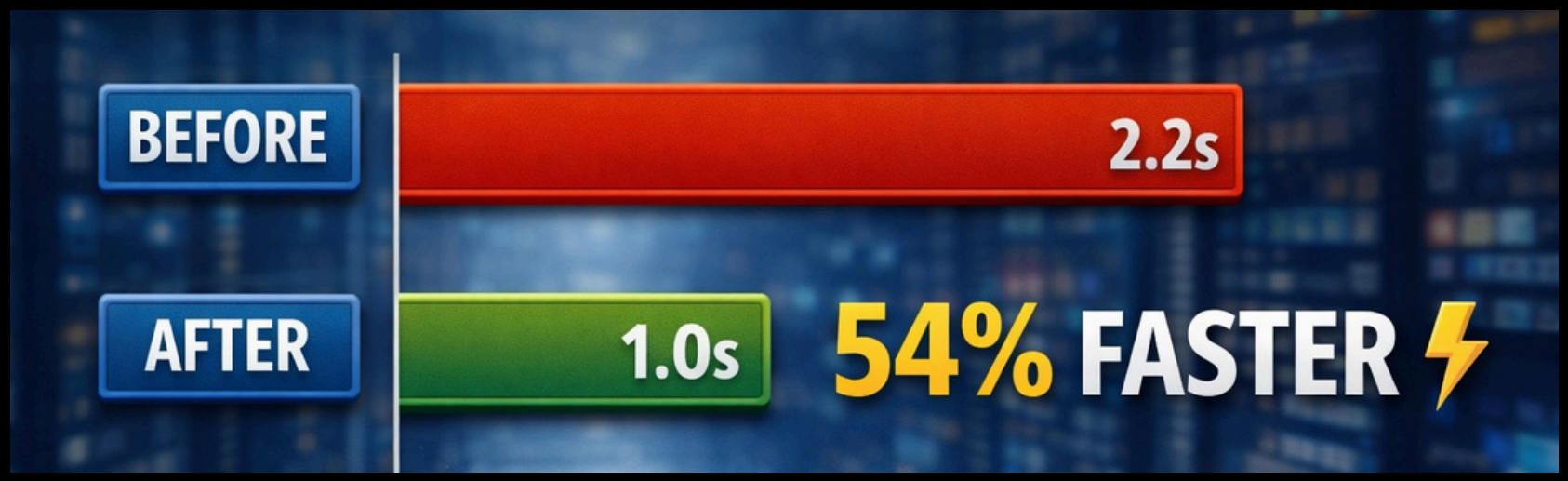
How I Achieved a 54% Reduction in API Response Time



From 2.2 seconds to 1 second, Without spending a single dollar.

A backend optimization guide for Node.js developers

A Node.js Performance Case Study

THE SETUP

What I Was Developing: A real-time cryptocurrency price tracker.

- **Backend:** Node.js and Express
- **Frontend:** Flutter mobile application
- **Hosting:** Render (free tier)
- **API:** CoinCap (subject to free tier limitations)

THE PROBLEM

If you call an API too many times, it can lead to slow responses, potentially resulting in reaching your free limits or charged money.

That means:

- Users experienced a 2-second wait
- App felt broken and slow
- Bad user experience
- Risk of hitting API limits

I needed to optimize my code for better performance.

Why Was It So Slow?

I profiled my code and found 4 issues:

- Creating new connections every request (Wasting 200ms on handshakes)
- Using setInterval for updates (Requests piling up during slow network)
- Large JSON responses (No compression, wasting bandwidth)
- No safety mechanism (Could burn all API credits overnight)

Fix 1: HTTP Keep-Alive

Every time my app asked for data, it was making a new connection. This wasted about 200ms every time.

The Fix: I told the server to keep the connection open (keepAlive: true). Now, it reuses the same line. The data comes much faster because we don't have to connect again and again. Saved around 200ms per request

This is the easiest optimization you can make

```
const httpsAgent = new https.Agent({
  keepAlive: true,
  keepAliveMsecs: 10000
});
```

Fix 2: Recursive SetTimeout

I was using setInterval to fetch data every 15 minutes.

setInterval has a dangerous flaw: If one request is slow or If the API lags, the next one still fires. hence, requests pile up and crash the server.

The Fix: I switched to a recursive setTimeout pattern. The next request only fires after the previous one finishes. It prevents traffic jams and keeps memory usage predictable.

```
const fetchData = async () => {
  await apiClient.get("/assets");
  // Next request starts AFTER this completes
  setTimeout(fetchData, REFRESH_INTERVAL);
}
```

Fix 3: GZIP Compression

API responses were large JSON files. Uncompressed text eats bandwidth. hence, slower the speed.

The Fix: I added a small setting (gzip) to compress the data. The files are now smaller, so they download 70% faster and use less internet.

```
headers: {
  'Accept-Encoding': 'gzip,deflate,compress'
}
```

Fix 4: Circuit Breaker Pattern

What happens if I forget to disable my server? Free tier APIs come with monthly limits, and I could quickly exhaust all my credits in a single night.

The Solution: The server will automatically shut down after reaching 100 requests.

```
let requestCount = 0;
const MAX_REQUESTS = 100;
if (requestCount >= MAX_REQUESTS) {
  console.log("Limit reached, stopping");
  return;
}
```

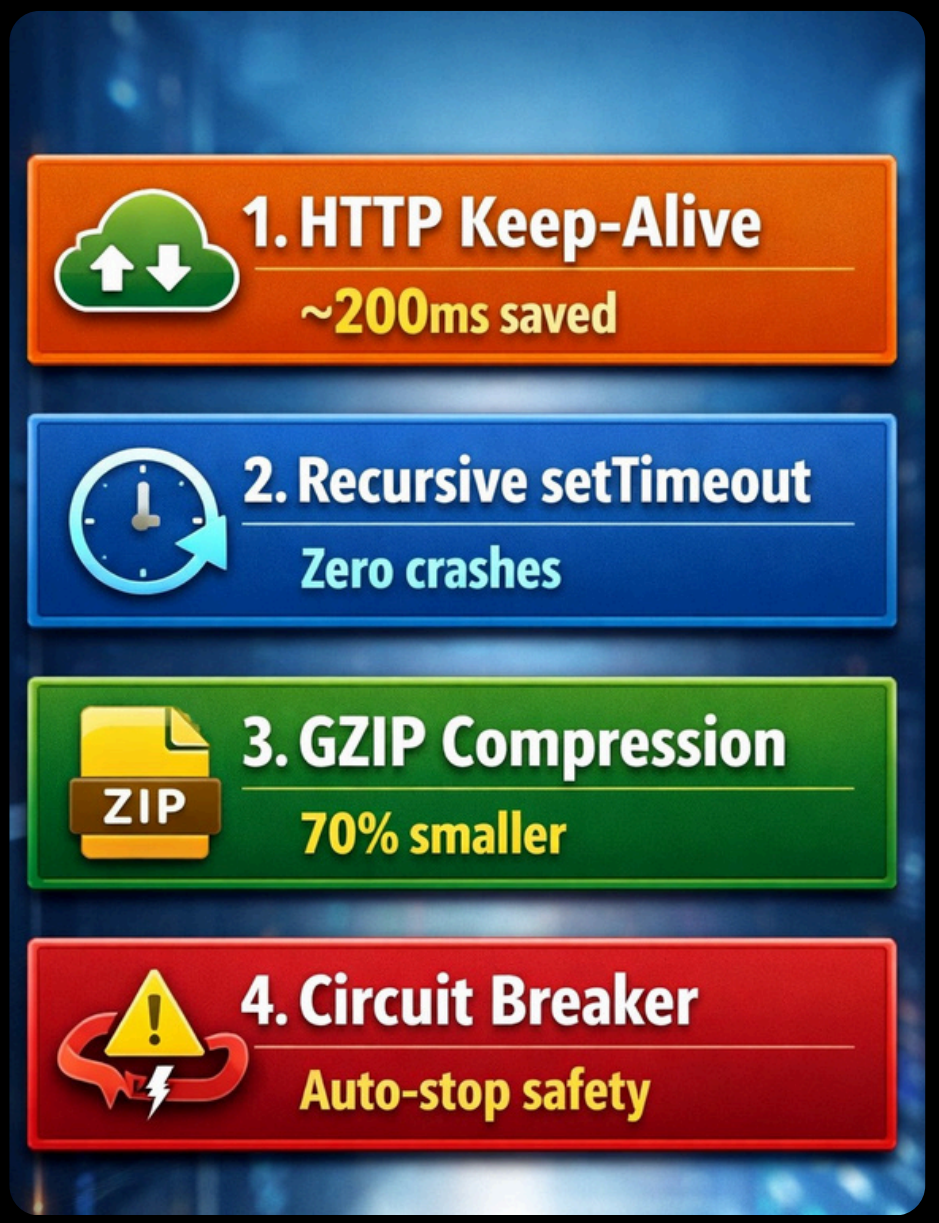
Performance Improvement

Before Optimization:

- Response time: 2260 ms
- Multiple handshakes
- Uncompressed large payloads
- Risk of crashes and overuse

After Optimization:

- Response time: 1042 ms
- Connections reused
- Compressed responses
- Safe and controlled

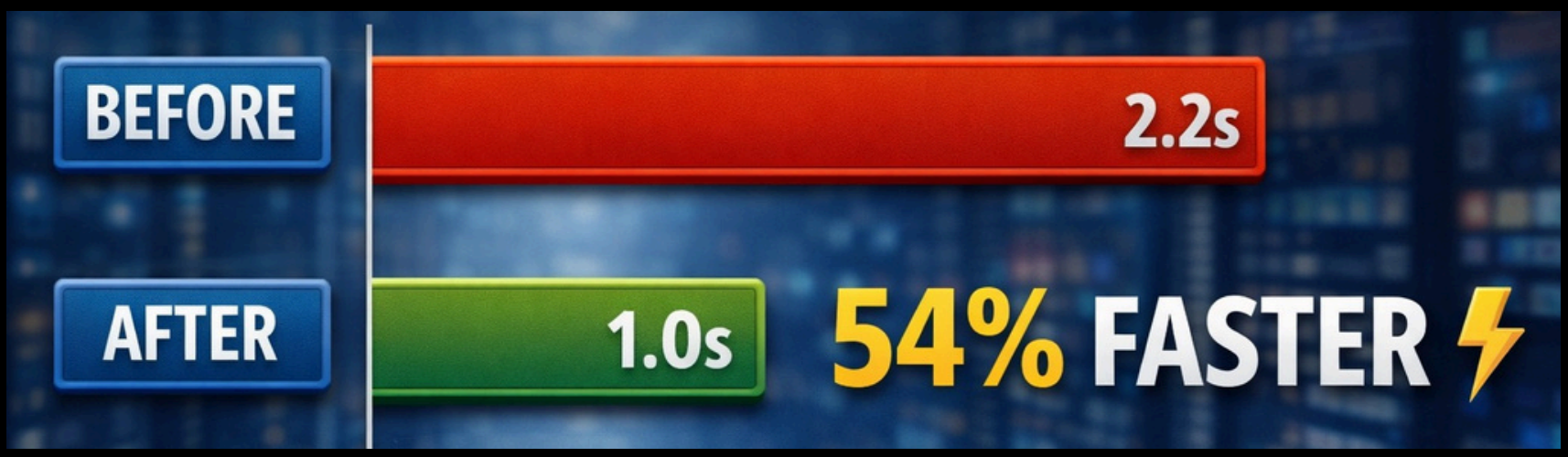


54% faster.

A Node.js Performance Case Study

Performance Improvement

	Standard Request	Optimized (VeloxFi)
Response Time	2,260ms	1,042ms
Improvement	—	54% Faster
Connection	New TCP Handshake	Persistent (Keep-Alive)
Cost Efficiency	High Risk	Capped (Circuit Breaker)



A Node.js Performance Case Study

Ready To Optimize Your API?

Full source code is on GitHub:
https://github.com/Pinkisingh13/VeloxFi-Real-Time-Watcher

🌟 Star the repo if you found this useful 🌟

Follow me for more real-world optimization guides.

What performance challenge are you facing?

Comment below and let's solve it together