FIELD NOTES FROM A PRESALES SOLUTIONS ARCHITECT

Web App Slow Only Through the WAF — Stripped gzip and a Per-Connection Throttle

A field note about a web app that was fast on its own server and took 40 seconds to load for every user — two problems hiding in the security layer in front of it, and the curl tests I now run before anyone touches the database.

Daanial Farrukh · 26 November 2026 · https://quabin.com/web-app-slow-behind-waf-gzip-per-connection-throttle/

The call started the way these calls always start.

"The application is slow. Every page. Forty, fifty seconds. We think the database needs more RAM."

It was a records-heavy internal web app. The usual stack: nginx in front, a Python app server behind it, a database behind that. Users reached it through a public address that sat on a web application firewall, the WAF, and the WAF passed traffic to the VM.

"Is it slow on the server too?" I asked. "Or only for users?"

There was a pause.

"We haven't checked it on the server. We just know users are complaining."

That question decided the next two days. Because the answer turned out to be: the server was fine. The VM could push the same file **60 times faster** on its own loopback than any user ever received it. The slow part was not the app and not the database. It was the box in the middle that everyone treated as invisible.

That's the trap. A WAF does not show up as "slow". It shows up as a healthy-looking app with unhappy users, and every eye goes to the database first.

Two Problems, One Symptom

There were two separate problems at the edge. They had nothing to do with each other, and both made the same symptom.

Problem one: compression was stripped in transit. The browser asked for gzip. The server could do gzip. What arrived at the browser was not gzipped. The main stylesheet alone was 1,398 KB raw. Gzipped, it is 164 KB.

Problem two: every TCP connection was capped at about 400 Kbps. Not the link. Not the total. Each single connection, at roughly 45 KB/s, flat, no matter how big the file or how fast the user's line.

Put them together and the maths is brutal. A 1.4 MB stylesheet at 45 KB/s is about 30 seconds — for one file. The page has dozens.

Here is the path the traffic took:

Browser → public address on the WAF → firewall policy → VM (nginx → app server → database)

Most teams test only the last part. Both problems lived in the first part.

Why the Usual Checks Missed It

Three things made this hard to see.

- **The server looked healthy.** CPU was low. The app logs showed requests finishing quickly. nginx did its job. Nothing on the VM was wrong, so nothing on the VM complained.
- **The network team looked for the wrong kind of limit.** Their first answer was "no bandwidth limit is configured". They were looking for a total quota on the address. What was there was a **per-session** traffic shaper: a cap that applies to each TCP connection on its own. Total throughput looked fine, because six connections at 400 Kbps still add up to something.
- **Compression looked configured.** nginx had gzip on. Someone had already added the right gzip_types. The config was correct. It just was not what reached the user, because the WAF sat in the middle and decided what to compress.

This is the part that trips people up. A config file proves what a server *would* send. Only a response header proves what a user *gets*.

How to Actually Prove It

Every step below is read-only. You can run all of it from a laptop and from the VM without changing anything. Use your own hostname and a real static file from the app — the biggest CSS or JS file is ideal.

Step 1: Measure the same file from outside and from inside

From your own machine, pull one large file and print the speed:

```
curl -s -o /dev/null -w "%{size_download} bytes  %{speed_download} B/s  %{time_total}s\n" https://app.example.com/static/css/app.min.css
```

Then on the VM, pull the same file through nginx on loopback, so the WAF is not in the path:

```
curl -sk -o /dev/null -w "%{size_download} bytes  %{speed_download} B/s  %{time_total}s\n" --resolve app.example.com:443:127.0.0.1 https://app.example.com/static/css/app.min.css
```

--resolve makes curl send the real hostname to 127.0.0.1, so nginx picks the right site and certificate. Use 127.0.0.1, not the public address — a VM behind NAT often cannot reach its own public address at all, and the test just hangs.

In this case the outside pull ran at about 47 KB/s. The loopback pull ran at about 2.8 MB/s. **Sixty times faster, same file, same moment.** That one comparison ends the database discussion. If the server can send a file at 2.8 MB/s and users get 47 KB/s, the problem is between them.

Step 2: Check what is really compressed

Ask for gzip the way a browser does, and read the headers that come back:

```
curl -s -D - -o /dev/null -H "Accept-Encoding: gzip" https://app.example.com/static/css/app.min.
css
```

Look for one line:

```
Content-Encoding: gzip
```

If it is missing, the file arrives raw. In this case the CSS and JS came back with no Content-Encoding at all.

Now look on the VM at what nginx thinks it does:

```
sudo nginx -T | grep -i gzip
```

Two lines matter. gzip_types lists what nginx will compress. gzip_proxied decides whether nginx compresses at all when the request came through a proxy. By default it is off, so a request that arrives from a WAF carrying a Via header is sent raw by nginx — and the WAF becomes the only layer that can compress it. If the WAF's compression profile does not cover that content type, nobody compresses it.

That is exactly what happened here, twice. First for CSS and JS. Later, after that was fixed, for JSON.

Step 3: Prove the per-connection cap

A slow transfer is not yet a cap. It could be congestion, jitter or a bad Wi-Fi link. Three tests separate them.

Force a large, uncompressed transfer and watch whether it ever speeds up. TCP normally starts slow and ramps up. A cap does not ramp.

```
curl -s -o /dev/null -H "Accept-Encoding: identity" -w "%{speed_download} B/s  %{time_total}s\
n" https://app.example.com/static/css/app.min.css
```

Here, 1.4 MB held a flat 49.7 KB/s for 28 seconds. No ramp. Three back-to-back runs gave 40.6, 40.7 and 45.1 KB/s. Flat and repeatable means a ceiling, not noise.

Run six pulls at the same time. If the limit is on the link, six connections share it and each gets slower. If the limit is per connection, each one gets the same cap and the total goes up.

```
seq 6 | xargs -P 6 -I{} curl -s -o /dev/null -H "Accept-Encoding: identity" -w "{}: %{
speed_download} B/s\n" https://app.example.com/static/css/app.min.css
```

Each of the six held about 48 KB/s. The total was 236 KB/s. **The link was fine. The cap was per TCP flow.** That is the fingerprint of a per-session shaper.

Compare with a sister site on the same path. The same app ran at a second site behind the same provider. From the same machine, on the same line, at the same moment, the same 1.4 MB file came from the sister site at 1.5 MB/s and from this site at 47 KB/s. Thirty-two times faster.

Then trace both:

```
tracert -d app.example.com
```

```
tracert -d sister.example.com
```

(On Linux, traceroute -n.) The two routes were identical hop for hop until the very last hop — the destination address. So it was not the user's ISP and not the shared backbone. It was the policy on that one public address.

Step 4: Separate network time from server time

Not every slow request is the network. In the browser, open DevTools › Network, click the slow request and open the **Timing** tab. Two numbers matter:

- **Waiting for server response** — the server is thinking.
- **Content Download** — the bytes are moving.

In this app, one dashboard request spent 17.6 seconds waiting and 7 ms downloading. That one is a slow query, and no WAF change will fix it. A user-list request was the opposite: 2.7 MB of JSON, almost no wait, all download time. That one is pure transfer, and the fix is compression and smaller responses, not query tuning.

Measure first, then pick the team. Sending a transfer problem to the database team wastes a week.

How to Actually Fix It

The fixes were all at the edge, and none of them touched the app.

Fix 1: Compress at the layer that faces the user

The WAF team enabled compression in its profile for CSS and JS. The stylesheet went from 1,398 KB on the wire to 164 KB. CSS load time dropped from about 32 seconds to about 4.

A day later the user list was still slow. The headers told the story:

```
Content-Type: application/json
Content-Length: 2748005
Vary: Accept-Encoding
```

No Content-Encoding. Vary: Accept-Encoding is present, so a layer capable of compression had looked at it and chosen not to. The WAF profile had CSS and JS but not JSON. Adding application/json (and HTML) to the WAF compression profile fixed it. JSON compresses very well, so the 2.7 MB list became a small fraction of that, and the slow tab became fast.

If you control nginx and want it to compress proxied requests too, this is the change (**this changes state; reload nginx afterwards**):

```
gzip_proxied any;
```

```
sudo nginx -t && sudo systemctl reload nginx
```

It is a belt-and-braces setting. It only helps if the WAF passes gzip through instead of inflating it again, so always re-check with the Step 2 header test from outside.

Fix 2: Remove the per-session shaper — and save it

The network team found the shaper on the firewall policy for that one public address and lifted it. From an outside vantage that had been stuck at 0.4 Mbps all day, the same file now pulled at 5 to 8 Mbps. The

login page went from 34.65 seconds to 2.0.

About forty minutes later, it was back. Same file, same vantage, flat 0.4 Mbps again.

The removal had not persisted. The usual reasons are boring ones: a change applied to the running configuration and never saved, an HA pair where the other unit still had the old policy, or a scheduled shaping rule. Whatever the cause, the lesson is the same.

One good test after a change is not a fix. It is a moment.

Fix 3: Verify over hours, not seconds

This step is where most tutorials stop and where mine starts. Run a logged loop from outside the provider's network and leave it running:

```
while true; do echo "$(date +%H:%M:%S) $(curl -s -o /dev/null -H 'Accept-Encoding: identity' -w '%{speed_download}' https://app.example.com/static/css/app.min.css)"; sleep 60; done | tee waf-speed.log
```

Each line is a time and a speed in bytes per second. A healthy edge stays well above the old cap for hours. A shaper that comes back shows up as a sudden flat line near 45,000 at the same value, minute after minute.

Run it from two different ISPs if you can. In this case one ISP saw the hard cap all the time and another saw a softer, bursty limit. Same address, different paths — which matters when the edge rule matches on the source.

If the log stays fast all afternoon, close the ticket. If one line drops back to the old number, it is not done. That drop is not noise. It is the verification.

The Checklist I Use Now

When an app is "slow for users", I run these before anyone resizes a VM or tunes a query:

1. **Same file, inside vs outside.** curl --resolve host:443:127.0.0.1 on the VM vs curl from a laptop. A big gap means the problem is in the path.
2. **Compression on the wire.** curl -D - -H "Accept-Encoding: gzip" and look for Content-Encoding: gzip on CSS, JS, JSON and HTML.
3. **Who compresses?** nginx -T | grep gzip, and read gzip_proxied. If it is off, the WAF must compress.
4. **Flat or ramping?** One large uncompressed pull. Flat for 20+ seconds means a cap.
5. **Per connection or total?** Six parallel pulls. Same speed each, higher total = per-session shaper.
6. **Which hop?** Trace to this site and to a sister site. Where the routes split is where the policy is.
7. **Waiting or downloading?** DevTools › Timing for the slowest request. Waiting = app team. Download = edge team.
8. **Over time.** A logged loop for hours after every fix.

What I Tell Every Customer Now

When a WAF or a firewall policy sits in front of an app, I ask one question in the design review:

"Who owns compression and per-connection limits on the public address — and how will we prove what reaches the user?"

If the answer is a name and a header test, they're in good shape. If the answer is "the WAF just passes traffic", we're not done yet.

I write about the questions I ask in real customer rooms — the ones that change architectures and surface the problems no one else noticed. If you found this useful, drop your email below and I'll send the next one when it's ready.

FAQs

Why is my web app fast on the server but slow for users?

If the same file downloads much faster on the server's loopback than from outside, the delay is in the path between them. A WAF or firewall policy in front of the app can strip compression or cap each connection. Compare curl from a laptop with curl --resolve to 127.0.0.1 on the server before changing the app or database.

How do I check if gzip compression is reaching the browser?

Request the file with Accept-Encoding: gzip and read the response headers. If there is no Content-Encoding: gzip line, the file arrives uncompressed, whatever the server config says. Check CSS, JS, JSON and HTML separately, because a WAF profile can cover some types and not others.

What does gzip_proxied do in nginx?

It decides whether nginx compresses responses to requests that came through a proxy. The default is off, so requests from a WAF that adds a Via header get uncompressed responses from nginx. Setting gzip_proxied any lets nginx compress them, but the WAF must pass the gzip through for users to benefit.

How can I tell a per-connection throttle from a slow link?

Run one large uncompressed download and see if the speed stays flat instead of ramping up. Then run six downloads in parallel. If each one gets the same low speed and the total rises, the limit is per TCP connection, not on the link — the fingerprint of a per-session traffic shaper.

Why did the throttle come back after the firewall team removed it?

A change that is not saved to the startup configuration, an HA peer that still holds the old policy, or a scheduled shaping rule can all bring it back. That is why one fast test after a change proves nothing. Log the download speed every minute from outside for hours before closing the ticket.

Will compression fix a slow page that is waiting on the server?

No. In browser DevTools, the Timing tab splits a request into waiting time and download time. Compression and edge fixes only shorten the download part. A request that spends most of its time waiting for the server needs query or app work instead.

Read online: https://quabin.com/web-app-slow-behind-waf-gzip-per-connection-throttle/