NGINX Troubleshooting Runbook Pack

A senior engineer's incident runbook for chasing 502s, 504s, TLS failures, and config emergencies through NGINX. DevOps AI ToolKit · devopsaitoolkit.com

NGINX is almost never the thing that's broken — it's the honest messenger reporting that an upstream is down, a timeout fired, a cert is wrong, or a config directive is illegal. The skill is reading NGINX's own signals (`error_log`, `nginx -t`, the status line) and deciding whether the fault is in NGINX, in the network between NGINX and the backend, or in the backend itself. Work top-to-bottom: confirm the exact status code and the log line behind it, then walk the request path until one hop explains the symptom.

> Examples use sanitized hostnames (`app-1:8080`, `example.com`) and Debian/Ubuntu paths (`/etc/nginx`, `/var/log/nginx`). Adjust for your distro. Run read-only checks first, and **always `nginx -t` before any reload** — a bad config reload can take the whole edge down.

---

1. First-Response Triage

Start here for every NGINX incident. Confirm the process is healthy, the config is valid, and read what the error log is actually saying before you touch anything.

```
# Is the config even valid? (read-only, safe to run anytime)
sudo nginx -t

# Is the master/worker process up, and did it reload cleanly?
systemctl status nginx --no-pager -l

# What is NGINX complaining about right now? (the single most useful command)
sudo tail -n 50 /var/log/nginx/error.log

# Reproduce from the box itself, bypassing DNS/CDN/client network
curl -sS -I http://127.0.0.1/            # status line + headers only
curl -sS -o /dev/null -w 'code=%{http_code} time=%{time_total}s\n' https://example.com/
```

- If `nginx -t` fails, stop — this is a §7 config-emergency, and a reload will not apply anyway.
- A 5xx from `curl -I 127.0.0.1` means the fault is NGINX↔upstream; a clean 200 locally but errors from outside points at DNS, firewall, or the CDN in front.
- Note the **exact status code** and the matching `error_log` line — every section below keys off that pair.

---

2. 502 Bad Gateway — Upstream Down / Connection Refused

A 502 means NGINX reached out to the upstream and got an invalid or refused response. NGINX is fine; the backend it proxies to is not listening, crashed, or is speaking the wrong protocol.

```
# The defining log lines for a 502:
sudo grep -E "connect\(\) failed|no live upstreams|upstream prematurely closed|recv\(\) failed" \
  /var/log/nginx/error.log | tail

# Is the backend actually listening on the expected port?
ss -ltnp | grep -E ':8080|:9000'
curl -sS -I http://app-1:8080/           # hit the upstream directly, bypassing NGINX
```

```
upstream app_backend {
    server app-1:8080 max_fails=3 fail_timeout=15s;
    server app-2:8080 backup;
}
server {
    location / {
        proxy_pass http://app_backend;
        proxy_next_upstream error timeout http_502 http_503;
    }
}
```

- `connect() failed (111: Connection refused) while connecting to upstream` → the backend process is down or bound to the wrong interface (e.g. `127.0.0.1` instead of `0.0.0.0`).
- `no live upstreams while connecting to upstream` → NGINX marked every server in the pool as failed (`max_fails` tripped). Fix the backend, then NGINX recovers on its own after `fail_timeout`.
- `upstream prematurely closed connection` → the backend accepted then crashed mid-response (OOM, worker restart). Chase the backend logs.
- For PHP/FastCGI: `connect() to unix:/run/php/php-fpm.sock failed` means php-fpm is down or the socket path/permissions are wrong — check `systemctl status php8.2-fpm`.

---

3. 504 Gateway Timeout — Slow Backend

A 504 means the upstream accepted the connection but did not send a response in time. The backend is alive but slow — a slow query, a blocked worker, an external API hanging. Raising the timeout is a stopgap, not a fix.

```
sudo grep "upstream timed out" /var/log/nginx/error.log | tail

# Time the upstream directly to prove it's slow, not NGINX
curl -sS -o /dev/null -w 'upstream: %{time_total}s\n' http://app-1:8080/slow-endpoint
```

```
location / {
    proxy_pass http://app_backend;
    proxy_connect_timeout 5s;     # time to establish the TCP connection
    proxy_send_timeout   60s;     # time between successive writes to upstream
    proxy_read_timeout   60s;     # time between successive reads — the usual 504 culprit
}
```

- `upstream timed out (110: Connection timed out) while reading response header` → the backend took longer than `proxy_read_timeout` (default 60s) to send the first byte. Fix the slow backend; only raise the timeout if the long operation is legitimate.
- If `%{time_total}` from the direct curl is also slow, NGINX is exonerated — the work is in the app.
- For FastCGI backends the equivalent knob is `fastcgi_read_timeout`; for uwsgi, `uwsgi_read_timeout`.

---

4. 413 Request Entity Too Large

NGINX rejected the request body because it exceeded `client_max_body_size` (default 1MB). Common on file uploads and large JSON payloads.

```
sudo grep "client intended to send too large body" /var/log/nginx/error.log | tail
```

```
# Set at http, server, or location scope. Scope it to the upload path, not globally.
location /api/upload/ {
    client_max_body_size 50m;
    proxy_pass http://app_backend;
}
```

- The log reads `client intended to send too large body: N bytes` — `N` tells you the real payload size, so size the limit to fit.
- Remember to raise it on the **backend too** — php-fpm (`upload_max_filesize`, `post_max_size`) or the app framework may reject it independently, producing a different error after NGINX passes it.
- `nginx -t` then `systemctl reload nginx` — no full restart needed.

---

5. 403 Forbidden / 404 Not Found — Filesystem & try_files

These are filesystem and routing problems: NGINX resolved a request to a path it can't read (403) or that doesn't exist (404). The `root`/`alias` mismatch and `try_files` behavior are the usual traps.

```
sudo grep -E "\" (403|404) " /var/log/nginx/access.log | tail
sudo grep -E "directory index of|Permission denied|No such file" /var/log/nginx/error.log | tail

# Resolve the actual path NGINX is computing, then check it exists and is readable
namei -l /var/www/example.com/html/index.html
sudo -u www-data test -r /var/www/example.com/html/index.html && echo readable || echo BLOCKED
```

```
server {
    root /var/www/example.com/html;
    index index.html;

    location / {
        try_files $uri $uri/ =404;      # serve file, then dir, else clean 404
    }

    # alias vs root: alias REPLACES the location prefix — mind the trailing slash
    location /assets/ {
        alias /var/www/example.com/static/;   # /assets/x.js -> /static/x.js
    }
}
```

- `directory index of "..." is forbidden` (403) → the request hit a directory with no `index` file and `autoindex off`. Add an `index` or a `try_files ... =404`.
- `Permission denied` (403) → the `www-data` user can't traverse the path. Every parent dir needs `+x`; the file needs `+r`. `namei -l` shows exactly where the chain breaks.
- 404 on a path you *know* exists → almost always `root` vs `alias` confusion, or a missing trailing slash on an `alias`.

---

6. TLS / SSL Handshake Failures

Handshake failures happen before any HTTP status code exists, so they surface in `error.log` and in `curl`/`openssl` output, not in

`access.log`. The usual suspects are an incomplete cert chain, a wrong SNI match, or a protocol/cipher mismatch.

```
# Full handshake + chain, honoring SNI
openssl s_client -connect example.com:443 -servername example.com </dev/null 2>/dev/null \
  | openssl x509 -noout -dates -subject -issuer

# Prove the served chain is complete (0 = good)
echo | openssl s_client -connect example.com:443 -servername example.com 2>&1 | grep -i "verify"

sudo grep -E "SSL_do_handshake|no suitable signature algorithm|unknown ca|certificate verify failed" \
  /var/log/nginx/error.log | tail
```

```
server {
    listen 443 ssl;
    server_name example.com;

    ssl_certificate     /etc/nginx/ssl/example.com.fullchain.pem;  # cert + intermediates
    ssl_certificate_key /etc/nginx/ssl/example.com.key;
    ssl_protocols       TLSv1.2 TLSv1.3;
    ssl_ciphers         HIGH:!aNULL:!MD5;
}
```

- `unable to get local issuer certificate` from the client → you served the leaf cert without intermediates. Point `ssl_certificate` at the **fullchain** file, not `cert.pem`.
- Wrong cert served for a hostname → SNI is matching the wrong `server` block; the first `server` with a `listen 443` and no match becomes the default. Verify with `-servername`.
- `no shared cipher` / `handshake failure` → the client and `ssl_ciphers`/`ssl_protocols` don't overlap (common when TLSv1.0/1.1 is disabled and an old client connects).
- Cert `notAfter` in the past → expired; renew and reload.

---

7. Config Emergency — nginx -t Fails to Reload

When `nginx -t` fails, the running config is unchanged (good) but you cannot apply anything new (bad). NGINX names the exact file and line — read it literally.

```
sudo nginx -t        # ALWAYS the first and last word on config validity
# Dump the effective merged config to find duplicated blocks
sudo nginx -T 2>/dev/null | grep -nE "server_name|listen|location /" | sort | uniq -d
```

- `duplicate location "/" in /etc/nginx/...:NN` → the same `location` appears twice in one `server` block. NGINX gives the line — delete or merge one.
- `unknown directive "..."` → a typo, a missing semicolon on the *previous* line (so NGINX reads two directives as one), or a directive from a module that isn't compiled in.
- `host not found in upstream "app-1"` → NGINX resolves upstream hostnames **at startup**; the name doesn't resolve right now. Fix DNS/`/etc/hosts`, or use a `resolver` + variable (see §9).
- `could not build the server_names_hash, you should increase server_names_hash_bucket_size` → you have long/many server names; bump `server_names_hash_bucket_size 128;` in the `http` block.
- Never `systemctl restart` to "clear" a bad config — if it's invalid, the service comes up stopped. Fix `nginx -t` green first, then `reload`.

---

8. Connection & File-Descriptor Limits

Under load NGINX can exhaust worker connections or file descriptors. The symptom is dropped connections and telltale `error_log` lines, not a specific HTTP code.

```
sudo grep -E "worker_connections are not enough|too many open files|24: Too many open files" \
  /var/log/nginx/error.log | tail

# What limit is the running master actually enforcing?
cat /proc/$(cat /run/nginx.pid)/limits | grep -i "open files"
```

```
worker_processes  auto;
worker_rlimit_nofile 65535;        # must be >= worker_connections * 2 (proxy = 2 fds/req)

events {
    worker_connections 16384;      # per worker; total = this * worker_processes
}
```

- `worker_connections are not enough while connecting to upstream` → each proxied request uses **two** fds (client + upstream); raise `worker_connections` and confirm `worker_rlimit_nofile` covers it.
- `Too many open files (24)` → the OS fd ceiling. Set `worker_rlimit_nofile` in nginx.conf **and** the systemd `LimitNOFILE=` (nginx.conf alone won't override a lower systemd limit).
- After changing systemd limits you must `systemctl daemon-reload` and fully restart NGINX — a reload won't raise `LimitNOFILE`.

---

9. proxy_pass DNS Resolution

By default NGINX resolves any hostname in `proxy_pass` (or `upstream`) **once, at config load**, and caches the IP forever. When a backend's IP changes (containers, cloud autoscaling, service discovery), NGINX keeps hitting the dead IP and 502s — while DNS itself is perfectly healthy.

```
# Prove DNS is fine but NGINX is holding a stale IP
getent hosts app-1
sudo grep -E "no live upstreams|connect\(\) failed" /var/log/nginx/error.log | tail
```

```
server {
    # Force per-request re-resolution: set a resolver AND use a variable in proxy_pass
    resolver 127.0.0.11 valid=10s ipv6=off;   # Docker's embedded DNS; use your resolver

    location / {
        set $upstream http://app-1:8080;      # variable -> NGINX re-resolves each request
        proxy_pass $upstream;
    }
}
```

- Static `proxy_pass http://app-1:8080;` (no variable) = resolved once at startup. If `app-1`'s IP moves, you get 502s until a reload.
- Using a **variable** in `proxy_pass` forces runtime resolution — but then a `resolver` directive is **mandatory**, or NGINX fails with `no resolver defined to resolve app-1`.
- `valid=10s` caps how long NGINX trusts a DNS answer; tune it to your churn rate.
- Note: a variable `proxy_pass` drops the implicit URI — include the path explicitly (`proxy_pass $upstream$request_uri;`) if you were relying on it.

---

10. Rate Limiting, Buffering & the 499

This bucket covers deliberate `limit_req` rejections (503) and buffer-related failures that masquerade as backend errors. The 499 is NGINX's own code for "client hung up before we answered."

```
sudo grep -E "limiting requests|upstream sent too big header|upstream sent more data than specified" \
  /var/log/nginx/error.log | tail
# Count 499s — client-side abandonment, often a downstream/CDN timeout shorter than yours
awk '$9==499' /var/log/nginx/access.log | wc -l
```

```
http {
    limit_req_zone $binary_remote_addr zone=api:10m rate=10r/s;

    server {
        location /api/ {
            limit_req zone=api burst=20 nodelay;   # 503 when exceeded
            proxy_pass http://app_backend;

            # Fix "upstream sent too big header": grow the header buffers
            proxy_buffer_size   16k;
            proxy_buffers       8 16k;
            proxy_busy_buffers_size 32k;
        }
    }
}
```

- `limiting requests, excess: N by zone "api"` → your own `limit_req` rejected the client with a 503. Expected under abuse; if it's hitting real users, raise `rate` / `burst`.
- `upstream sent too big header while reading response header from upstream` → the backend's response headers (often a huge `Set-Cookie` or auth token) overflowed `proxy_buffer_size`. Increase it; this is *not* a backend crash.
- **499** in `access.log` → the client (or the CDN/LB in front of NGINX) closed the connection before NGINX replied — usually a downstream idle-timeout shorter than your backend's response time. Align the front timeout with `proxy_read_timeout`.

---

11. Decision Tree

```
What did the client / curl -I get back?

├─ 502 Bad Gateway
│    error_log: "connect() failed (111)"  → backend not listening        → §2
│    error_log: "no live upstreams"       → max_fails tripped whole pool → §2
│    error_log: "prematurely closed"      → backend crashed mid-response → §2
│    502 only after a deploy/scale event  → stale upstream IP            → §9
│
├─ 504 Gateway Timeout
│    error_log: "upstream timed out ... reading response header"          → §3
│    (direct curl to upstream is also slow → it's the app, not NGINX)
│
├─ 413 Request Entity Too Large  → client_max_body_size too low          → §4
│
├─ 403 / 404
│    "Permission denied" / "directory index forbidden" → filesystem perms → §5
│    path exists but 404                  → root vs alias / try_files      → §5
│
├─ TLS error before any HTTP code
│    "unable to get local issuer"         → incomplete chain (fullchain)  → §6
│    wrong cert for host                  → SNI / default_server          → §6
│    "no shared cipher"                   → protocol/cipher mismatch       → §6
│
├─ Nothing loads + nginx -t FAILS         → config emergency              → §7
│
├─ Dropped conns under load
│    "worker_connections are not enough" / "Too many open files"          → §8
│
└─ 503 "limiting requests"  → limit_req                                    → §10
   499 in access.log        → client/CDN hung up first                     → §10
   "upstream sent too big header" → proxy_buffer_size                      → §10
```

12. Copy/Paste One-Shot Triage

```
echo "== config valid? ==" ; sudo nginx -t
echo "== service state ==" ; systemctl is-active nginx; systemctl show nginx -p NRestarts --value
echo "== local response ==" ; curl -sS -o /dev/null -w 'code=%{http_code} time=%{time_total}s\n' http://127.0.0.1/
echo "== recent 5xx (access) ==" ; sudo awk '$9 ~ /^5/ {print $9}' /var/log/nginx/access.log | sort | uniq -c | tail
echo "== 499 count ==" ; sudo awk '$9==499' /var/log/nginx/access.log | wc -l
echo "== top error_log lines ==" ; sudo grep -Eio "connect\(\) failed|upstream timed out|no live upstreams|too large body|Permission denied|Too
many open files|upstream sent too big header|SSL_do_handshake" /var/log/nginx/error.log | sort | uniq -c | sort -rn | head
echo "== listening backends ==" ; ss -ltnp | grep nginx
echo "== fd limit (master) ==" ; cat /proc/$(cat /run/nginx.pid 2>/dev/null)/limits 2>/dev/null | grep -i "open files"
```

13. Incident Notes Template

```
INCIDENT: NGINX <status code / symptom>
Started (UTC):        ____    Detected by: ____    Sev: ____
Impact:               (which vhost / paths / % of requests / users)
Status code seen:     [ ] 502  [ ] 504  [ ] 413  [ ] 403/404  [ ] TLS  [ ] 503  [ ] 499
nginx -t:             [ ] pass  [ ] FAIL: ____ (file:line)
Key error_log line:   ____________________________________________
Fault localized to:   [ ] NGINX config  [ ] NGINX<->upstream net  [ ] backend app
                      [ ] TLS/cert       [ ] DNS/resolver          [ ] limits/buffers
Upstream direct test: curl time=____s  code=____   (proves app vs proxy)
Action taken:         (directive changed + nginx -t + reload / restart + who + when)
Result:               [ ] recovered @ ____    [ ] escalated to ____
Root cause:           ____
Follow-ups:           (timeouts / limits / cert renewal automation / alerts) ____
```

DevOps AI ToolKit · devopsaitoolkit.com — practical AI workflows for real cloud engineers. Free to use for your own and your employer's internal work; please don't resell or republish the pack itself. Commands are provided as-is; always validate against your own environment before running in production. When you're stuck, paste the errors into the free AI Incident Response assistant at devopsaitoolkit.com/dashboard/incident-response.