Here’s a crisp post-mortem + a future-proof setup so this doesn’t bite you again—on this VM or any other.

**What went wrong (root causes)**

1. **Proxy upstream mismatch (wrong service name in Caddyfile).**  
   Inside the caddy container, the Caddyfile still had reverse\_proxy myapp:3000 even after you edited the host file to nine-tones-app:3000. Because the container mounts the host file at start, Caddy kept using the old contents until you restarted/reloaded the container with the correct mount.
2. **Partial reloads / stale mounts.**  
   A few times you edited the host file but the container either wasn’t using that path (or had a short placeholder file) until docker compose down && up -d. That’s why “adapt” showed myapp until you bounced the stack.
3. **Firewall/tag drift.**  
   You cleaned up rules (good!) but briefly ended up with no rule allowing :80 (or the tag didn’t match), giving ERR\_CONNECTION\_TIMED\_OUT from the internet while local curl still worked.
4. **Directly exposing app ports (3000/3001) to the internet.**  
   That “worked sometimes” until you removed the 3000 rule, then :3000 timed out—as expected. Mixing direct exposure with a reverse proxy adds confusion and CORS problems.
5. **CORS / cross-origin calls from the browser.**  
   Frontend called http://<ip>:3001 which is a *different origin* than the page, so you hit CORS preflights and timeouts if 3001 wasn’t reachable or allowed. Same-origin via Caddy solves this cleanly.
6. **Line endings warning (CRLF vs LF).**  
   Windows git wanted to rewrite line endings; in config/scripts that run in Linux containers, CRLF can sometimes break things.
7. **IAM hiccups in Cloud Shell.**  
   You used an identity without compute.instances.get / compute.firewalls.create etc. That blocked tags/rules and created confusion (“is the rule there or not?”).

**How you fixed it (signals → actions)**

* Verified Caddy saw the *actual* file it uses (inside the container) and reloaded.
* Forced the correct upstream (nine-tones-app:3000) and bounced the stack; validated with caddy adapt and curl from Caddy ➜ app.
* Standardized GCP firewall: single rule allow-http-https for tag web; tagged the VM as web.
* Confirmed end-to-end with curls: inside VM ➜ Caddy (127.0.0.1:80); Caddy ➜ app (nine-tones-app:3000); Cloud Shell ➜ public IP.
* App came back; scanners in logs prove the public IP is reachable.

**Prevent it next time — opinionated, repeatable setup**

**0) Repo layout (everything in git; nothing “SSH-only”)**

nine-tones/

docker/

caddy/

Caddyfile

compose.yml

app/

Dockerfile

backend/ # listens on 3001

frontend/ # static build served on 3000

deploy/

gcloud-setup.sh # tags + firewall + IP test

healthcheck.sh

.env.example

.gitattributes

Makefile

README.md

**.gitattributes (enforce LF everywhere)**

\* text eol=lf

\*.bat text eol=crlf

\*.cmd text eol=crlf

\*.png binary

\*.jpg binary

\*.ico binary

**1) Compose (single publisher: Caddy only)**

docker/compose.yml

services:

app:

image: europe-west3-docker.pkg.dev/…/nine-tones-app:latest

# Do NOT publish 3000/3001 to host

expose:

- "3000" # frontend (serve)

- "3001" # backend (API)

env\_file:

- ../../.env # non-secrets (safe)

# Secrets are injected at runtime via environment or Docker/secret store

healthcheck:

test: ["CMD", "curl", "-fsS", "http://localhost:3000/"]

interval: 30s

timeout: 3s

retries: 3

networks: [ web ]

caddy:

image: caddy:2

ports:

- "80:80"

- "443:443"

volumes:

- ./caddy/Caddyfile:/etc/caddy/Caddyfile:ro

- caddy\_data:/data

- caddy\_config:/config

depends\_on:

- app

networks: [ web ]

networks:

web:

volumes:

caddy\_data:

caddy\_config:

**Why**: Only Caddy binds 80/443; app stays private on the Docker network. This eliminates mixed paths and CORS.

**2) Caddyfile (same-origin routing; add CORS only if you need it)**

docker/caddy/Caddyfile

:80 {

log {

output stdout

format console

}

# Frontend (static)

reverse\_proxy app:3000

# API under same origin → no CORS needed

handle\_path /api/\* {

reverse\_proxy app:3001

}

@health {

path /health

}

respond @health 200

}

**Notes**

* The service name is app (Compose DNS), not a hostname/IP.
* The browser fetches /api/... on the *same origin*—no CORS preflight pain.
* If you later add a domain, change :80 to your domain and enable HTTPS (Caddy will get certs automatically).

**3) Frontend config (no hardcoded IPs/ports)**

* Use relative paths: fetch('/api/rs/get\_waybills', …)
* If you need runtime base URL, use an env variable at build time:
  + CRA: REACT\_APP\_API\_BASE=/api
  + Next/Vite: similar, but keep it **relative** for production.
* Never bake http://<ip>:3001 into the frontend bundle.

**4) Backend (3001) — optional CORS, better is same-origin**

If you ever need cross-origin (local dev), enable narrow CORS:

app.use(require('cors')({

origin: ['http://localhost:3000'],

methods: ['GET','POST','PUT','DELETE','OPTIONS'],

credentials: false

}));

For production via Caddy, you *shouldn’t* need CORS at all because frontend and API share the same origin.

**5) Secrets & config (Cloud Secret Manager + env injection)**

* Keep **non-sensitive** defaults in .env.example.
* Store secrets in **Google Secret Manager**:
  + RS\_USERNAME, RS\_PASSWORD, RS\_ENDPOINT=https://services.rs.ge/...
* At deploy time, fetch and inject into the container environment (don’t write to disk). Two common approaches:

**A) GCE Metadata + Startup script (VM)**  
Use a tiny script that reads secrets via gcloud secrets versions access and runs docker compose up with those envs exported. Example:

deploy/gcloud-setup.sh (run once from Cloud Shell)

set -euo pipefail

PROJECT\_ID="nine-tones-bots-2025-468320"

ZONE="europe-west3-b"

VM="vm-runtime"

gcloud config set project "$PROJECT\_ID"

gcloud compute instances add-tags "$VM" --zone "$ZONE" --tags web || true

gcloud compute firewall-rules describe allow-http-https >/dev/null 2>&1 || \

gcloud compute firewall-rules create allow-http-https \

--network=default --direction=INGRESS --priority=1000 \

--action=ALLOW --rules=tcp:80,tcp:443 \

--source-ranges=0.0.0.0/0 --target-tags=web

deploy/start.sh (on the VM; **in repo**)

#!/usr/bin/env bash

set -euo pipefail

# Pull secrets at runtime

export RS\_USERNAME="$(gcloud secrets versions access latest --secret RS\_USERNAME)"

export RS\_PASSWORD="$(gcloud secrets versions access latest --secret RS\_PASSWORD)"

export RS\_ENDPOINT="$(gcloud secrets versions access latest --secret RS\_ENDPOINT)"

cd /opt/nine-tones/docker

docker compose pull

docker compose up -d

Give the VM service account Secret Manager Secret Accessor and the minimal Compute roles it needs.

**B) Cloud Build/Artifact Registry**  
Bake only *non-secret* config into the image; inject secrets as envs at runtime with docker run -e RS\_USERNAME=.... (Same net effect.)

**6) GCP networking (one simple rule)**

* Tags on VM: web
* **One** firewall rule:
  + name: allow-http-https
  + target tags: web
  + allow: tcp:80, tcp:443
* Nothing else inbound. Don’t expose 3000/3001 publicly.

**Handy one-liners (Cloud Shell):**

# Tag VM

gcloud compute instances add-tags vm-runtime --zone=europe-west3-b --tags web

# Create single rule if missing

gcloud compute firewall-rules describe allow-http-https >/dev/null 2>&1 || \

gcloud compute firewall-rules create allow-http-https \

--network=default --direction=INGRESS --priority=1000 \

--action=ALLOW --rules=tcp:80,tcp:443 \

--source-ranges=0.0.0.0/0 --target-tags=web

**IAM you need (for who runs those commands):**

* Compute Instance Admin (v1) to set tags, read instance IPs.
* Compute Security Admin (or Network Admin) to create firewall rules.
* Secret Manager Secret Accessor to read secrets at runtime.  
  (Or temporarily Project Editor if you’re okay with broad rights while bootstrapping.)

**7) Local dev & builds (so prod matches dev)**

* **Dockerize dev**: run docker compose -f docker/compose.yml up locally with the same Caddy routing (you can map localhost:8080 to Caddy if you don’t want to use :80 locally).
* Frontend never calls hardcoded IPs—always /api/....
* Add a **Makefile** with developer-friendly targets:

Makefile

.PHONY: up down logs caddy app ps curl ip fmt

up:

\tcd docker && docker compose up -d

down:

\tcd docker && docker compose down

logs:

\tcd docker && docker compose logs -f --tail=200

ps:

\tcd docker && docker compose ps

caddy:

\tcd docker && docker compose exec caddy sh -lc 'caddy adapt --config /etc/caddy/Caddyfile --pretty | sed -n "1,120p"'

curl:

\tcurl -I http://127.0.0.1:80/

fmt:

\tcd docker && docker compose exec caddy sh -lc 'caddy fmt --overwrite /etc/caddy/Caddyfile'

ip:

\tgcloud compute instances describe vm-runtime --zone=europe-west3-b --format='get(networkInterfaces[0].accessConfigs[0].natIP)'

**8) Health & diagnostics runbook (copy/paste)**

**Inside the VM**

# 1) Is Caddy listening?

ss -ltnp | awk '$4 ~ /:80$/'

# 2) Is the caddy container up?

cd /opt/nine-tones/docker && docker compose ps

# 3) Did Caddy load the file we think?

docker compose exec caddy sh -lc 'cat /etc/caddy/Caddyfile; echo; caddy adapt --config /etc/caddy/Caddyfile --pretty | sed -n "1,120p"'

# 4) Local test VM -> Caddy

curl -I http://127.0.0.1:80/ -m 5

# 5) Caddy -> app by service name

docker compose exec caddy sh -lc 'apk add --no-cache curl >/dev/null 2>&1 || true; curl -I http://app:3000/ -m 5'

# 6) App logs

docker logs --tail=200 app

# 7) Public test (from Cloud Shell)

VM\_IP="$(gcloud compute instances describe vm-runtime --zone=europe-west3-b --format="get(networkInterfaces[0].accessConfigs[0].natIP)")"

curl -I "http://$VM\_IP/" -m 10 -v

**If public fails but local works:** it’s almost always firewall/tag/IAM. Re-run the tag + single-rule recipe.

**9) Hardening quick wins**

* Add a default 404 handler in Caddy so scanner garbage doesn’t return your SPA index.html.
* Put the app behind **HTTPS & a domain** (Caddy will auto-cert).
* Consider rate-limiting / and blocking weird methods if you care.

**10) Why the RS endpoint confusion happened**

Your frontend tried to call /api/rs/get\_waybills at :3001 directly. When 3001 wasn’t exposed (or was blocked), it timed out. Even when reachable, it triggered CORS. The right pattern is:

* Frontend → /api/rs/get\_waybills (same origin, via Caddy)
* Caddy → app:3001
* Backend → **external** https://services.rs.ge/WayBillService/… using credentials from Secret Manager.

This removes CORS, firewall surprises, and any need to expose 3001 publicly.

**TL;DR checklist you can reuse on any VM**

* All infra/config lives in repo (compose.yml, Caddyfile, scripts).
* Only Caddy publishes 80/443; app ports are **not** published.
* Frontend uses **relative** /api URLs (no IPs/ports).
* Caddy routes /api/\* → app:3001.
* One GCP firewall rule allow-http-https for tag web; VM has tag web.
* Proper IAM (compute + firewall + secrets).
* Secrets in Secret Manager; injected at runtime; .env.example in repo for non-secrets.
* .gitattributes enforces LF line endings.
* Health/runbook scripts for fast triage.
* Validate with caddy adapt, curl 127.0.0.1:80, curl from caddy to app, and curl public IP.

If you adopt this as your **standard** (copy the folder skeleton into new projects), you’ll avoid the proxy drift, CORS, firewall/tag inconsistencies, and “it works locally but not on the IP” issues.