A DISTRIBUTED SYSTEM THAT DOES NOT FALL OVER

tapper.

Distributed CS2 matchmaking on two load-balanced API instances. Kill one mid-match, the platform keeps running.

React 19 · Traefik v3.7.1 · Go 1.26.3 · Fiber v2 · PostgreSQL 18 · Access + refresh tokens · Docker Compose · k6 + ApacheBench

What was asked, and where it lives.

#	REQUIREMENT	WHERE IT LIVES	STATUS
1	Load balancer + failover	`traefik/traefik.yml`, healthcheck on `/api/health` every 5s	PASS
2	Two service instances	`docker-compose.yml` · `api-1` + `api-2`, shared Traefik service	PASS
3	Simple frontend	React 19 + Vite 8 SPA in `frontend/`	PASS
4	Access/refresh auth (OWASP)	RS256 access JWTs, opaque rotating refresh tokens, bcrypt cost 12, SSE purpose tokens	PASS
5	Persistent storage	PostgreSQL 18 with named volume `pgdata`	PASS
6	Containerized	Every service in `docker compose up`, multi-stage builds	PASS
7	Latest stable releases	Go 1.26.3, Node 26 alpine3.23, React 19, Vite 8, Tailwind 4, Postgres 18 alpine3.23, Traefik v3.7.1	PASS
8	Single-command bring-up	`docker compose up --build`, auto migrate, auto seed, auto keygen	PASS
9	Load test + docs	k6 smoke / load / stress / failover + ApacheBench 1000-request auth read, `loadtest/results/report.md`	PASS
10	Open source only	Fiber, GORM, jwt/v5, React, Zustand, Traefik, Postgres, all OSS	PASS

The shape of a request.

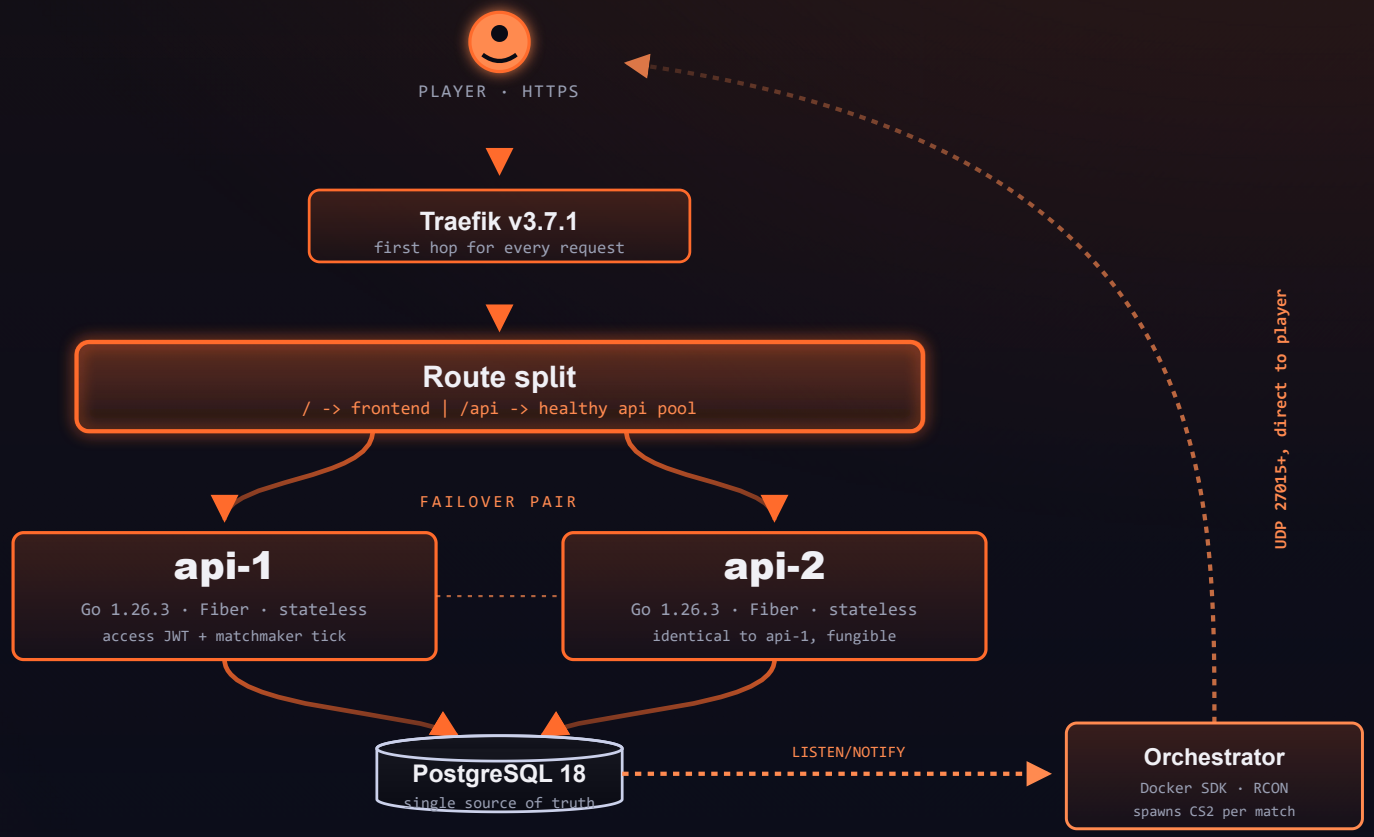
01 Browser enters Traefik.

02 Traefik serves SPA route.

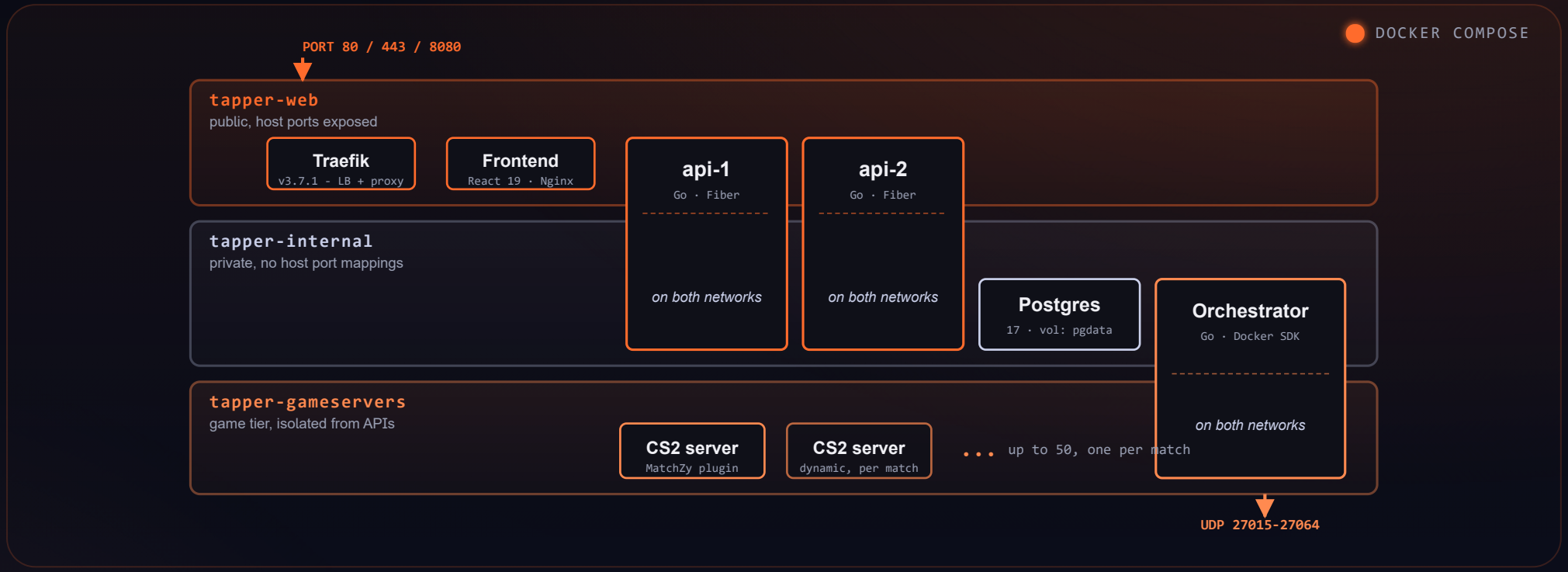
03 Traefik load-balances APIs.

04 Either api can answer.

05 Single source of truth.



Three Docker networks, segmented by trust.



tapper-web

Public-facing. Traefik, frontend, both APIs. The only network with host port mappings.

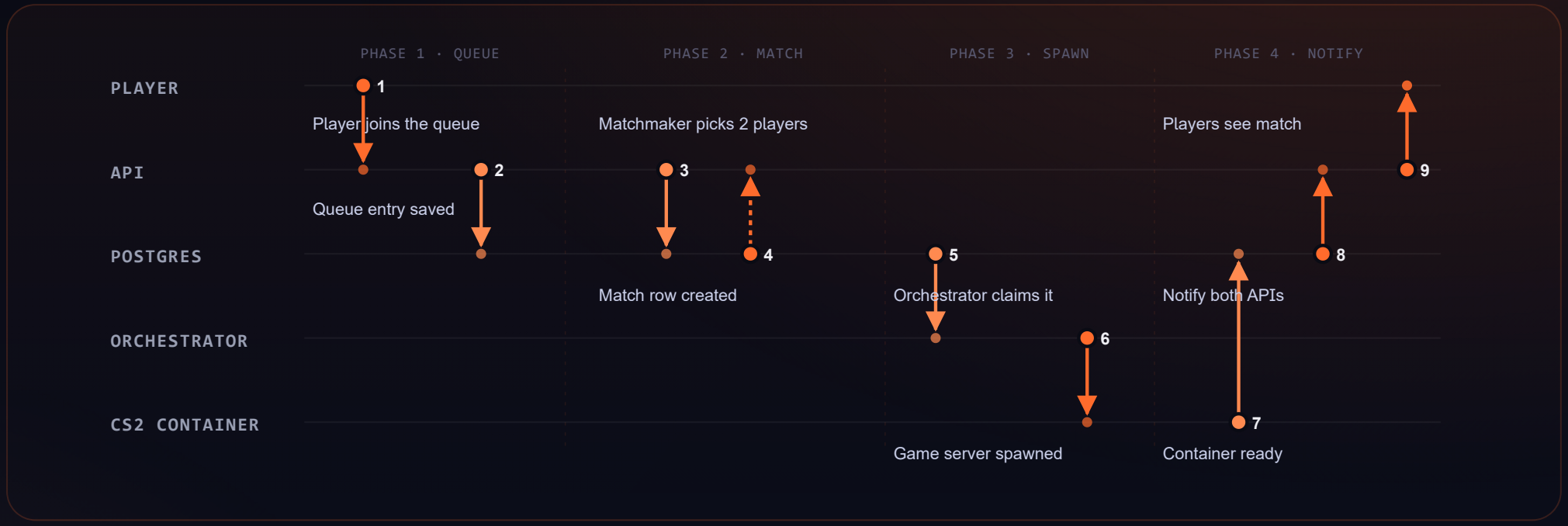
tapper-internal

Private. APIs talk to Postgres here. No host port exposure, the database is unreachable from outside Docker.

tapper-gameservers

Game tier. Orchestrator spawns CS2 here. APIs cannot reach game containers. Players connect via raw UDP, bypassing Traefik.

How a match comes alive.



PHASE 1 · QUEUE

Players hit the queue endpoint, Traefik routes to either API instance, the entry lands in Postgres.

PHASE 2 · MATCH

Matchmaker ticks every 5s in both APIs, picks rows via `SKIP LOCKED`, no conflict.

PHASE 3 · SPAWN

Orchestrator polls matched rows, allocates port + GSLT, calls Docker SDK to spawn the server.

PHASE 4 · NOTIFY

Container ready, Postgres `NOTIFY` fans out, both APIs push SSE, both players see the match.

Four distributed-systems calls behind tapper.

Stateless workers, state in Postgres

01

Every API request carries an access token. No session in api memory. Either instance can answer any request, mid-queue or mid-history. Failover becomes a routing change, not a state migration. Same property makes horizontal scaling trivial.

Database as the coordinator

02

No etcd, no Redis lock manager. Both apis and the orchestrator share rows via `SELECT FOR UPDATE SKIP LOCKED`: queue picks, port allocation, GSLT allocation, all use the same primitive. The DB we already need does the coordination we would otherwise add a service for.

Asymmetric crypto for cross-node trust

03

One RS256 keypair generated at first boot, mounted read-only into every api instance. An access token issued by api-1 verifies on api-2 because they share the public key. Refresh tokens are opaque database-backed secrets, rotated on use and capped at 24 hours.

Failover at the proxy, not in the app

04

Traefik owns the healthcheck loop on `/api/health`. The api code never asks "am I alive?". Adding or removing an instance is a Traefik state change, not application logic. Single responsibility: the app serves traffic, the proxy decides who is alive.

SECURITY NOTES

RS256 + bcrypt cost 12
OWASP-aligned signing and password hashing

Opaque refresh tokens
Rotated on use, stored hashed, capped at 24h

Purpose-scoped SSE tokens
Access JWT never enters a URL

The next five minutes.

- **0:00 to 2:00**, **matchmaking demo**: queue two users, watch the orchestrator spawn a server.
- **2:00 to 3:00**, **admin tour**: owner logs in, brief walk through user management.
- **3:00 to 5:00**, **live failover**: kill api-2 mid-load-test, watch the recovery, recap the numbers.

Stack health, one glance

```
$ docker ps
NAME                    STATUS
tapper-api-1-1          Up (healthy)
tapper-api-2-1          Up (healthy)
tapper-orchestrator-1 Up (healthy)
tapper-postgres-1       Up (healthy)
tapper-traefik-1        Up (healthy)

$ curl -s localhost/api/health
{ "status": "ok", "instance": "a8f3c1..." }
```

Traefik dashboard at `localhost:8080` shows both apis as live backends.

Kill one instance. Keep serving. In 20 seconds.

Live commands

```
# terminal 1, 20s sustained load
$ docker run --rm --network tapper-web \
    -v $PWD/loadtest:/scripts \
    -e BASE_URL=http://traefik \
    grafana/k6:latest run \
    /scripts/quick-failover.js

# terminal 2, at t+5s
$ docker stop tapper-api-2-1
tapper-api-2-1

# confirm api-1 is still serving
$ curl -s http://localhost/api/health
{
  "status": "ok",
  "instance": "a8f3c1...",
  "time": "2026-05-17T11:27:28Z"
}

# reset for the next demo run
$ docker start tapper-api-2-1
```

How much can it take?

SCENARIO	PEAK VUS	TOTAL REQS	RPS	HTTP P95	HTTP P99	PROFILE P95	5XX
Smoke 2 VUs, 30s	2	91	2.8/s	183ms	,	<5ms	0
Load ramp to 100 VUs, 4m	100	28,834	110.9/s	171ms	~200ms	5.31ms	0
Stress ramp to 250 VUs, 3m	250	84,113	433.6/s	656ms	935ms	160ms	0
Failover 50 VUs, 3m, kill at 45s	50	18,162	93.7/s	166ms	~250ms	~6.5ms	0
ApacheBench 1000 auth profile reads	50	1,000	1,471/s	83ms	177ms	83ms	0

Where it breaks

bcrypt cost 12 on register and login. Under stress, auth p95 climbs from 185ms to 969ms while non-bcrypt reads stay below 200ms p95. The wall is CPU on password hashing, not the database, not the proxy.

What stays flat

access-token-protected **reads**: `/api/profile`, `/api/leaderboard`, `/api/matches`. All under 10ms p95 at 100 VUs, under 200ms p95 at 250 VUs. No Postgres pool saturation. No 5xx anywhere.

Fresh clone to running system, one command.

```
# 1. Clone
$ git clone <repo> tapper && cd tapper

# 2. Secrets
$ cp .env.example .env
$ $EDITOR .env

# 3. Up
$ docker compose up --build
[+] Building 4 images
[+] Network tapper-web        Created
[+] Network tapper-internal   Created
[+] Volume jwt-keys           Created
[+] Volume pgdata             Created
[+] Container jwt-keygen      Generated RSA keys
[+] Container postgres        Healthy
[+] Container api-1           Healthy
[+] Container api-2           Healthy
[+] Container traefik         Healthy
[+] Container frontend        Started

# 4. That's it.
```

What runs automatically

- **jwt-keygen** generates a 2048-bit RSA key pair on first boot and persists it to a Docker volume.
- **Postgres** creates its database with a healthcheck that gates the apis.
- **GORM AutoMigrate** runs every table migration on api startup (users, matches, queue, bans, tickets, audit log).
- **Seed** populates the port pool, default Elo config, and platform config.
- **Traefik** discovers api-1 and api-2 by Docker label and starts health-checking immediately.

Zero manual setup beyond secrets. The first user with `OWNER_EMAIL` auto-becomes owner. Subsequent users are normal accounts.

Thank you.

Questions?