MUHAMMET ŞAFAK — VISUALS

5 breaking points in data-intensive systems



From the cheapest fix to the costliest, and a Laravel stack with a reason for every part.

Muhammet Şafak • October 3, 2026
muhammetsafak.com

SECTION 01

Five breaking points

From the working set to a separate data tier: from the cheapest intervention to the most expensive.

WHAT SETS THE SCALE

Not requests: the data profile.

A system can take 50 requests per minute and still be drowned by its data; another handles 5,000 requests per second comfortably while its data tier sits untouched.

DATA PROFILE

Five axes

Working set / RAM ratio

Does the frequently read data fit in memory?

Read / write ratio

Which way does the load tilt?

Single table size

How big has a single table grown?

Write throughput

How much does the primary write per second?

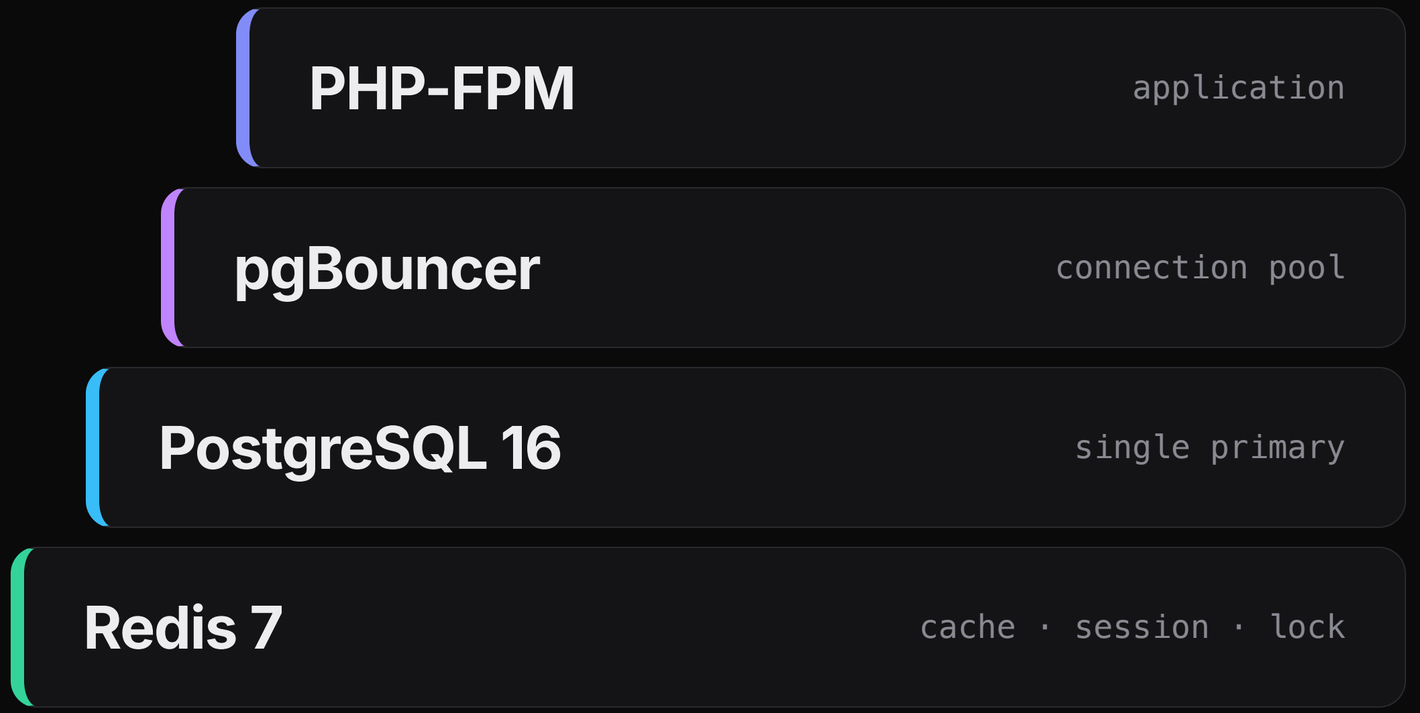
Consistency requirement

Which reads must see the freshest data?

STARTING ARCHITECTURE

Starting with a single primary

PHP-FPM reaches PostgreSQL through pgBouncer and Redis directly. Enough for a few tens of GB of data and thousands of requests per minute, as long as the working set fits in RAM.



BREAKING POINT 01 · WORKING SET

Signal: cache hit ratio

The frequently read data doesn’t fit in RAM. Above 99% is healthy. A drop toward 95%, read together with disk IOPS, shows the working set has spilled out of RAM.

pg_stat_database.sql

```sql
SELECT
sum(blks_hit) * 100
  / nullif(sum(blks_hit) + sum(blks_read), 0)
  AS cache_hit_ratio
FROM pg_stat_database;
```

ORDER OF INTERVENTION

Exhaust the cheap one first

1

Index discipline

`cut unnecessary scans`

2

Separate dead data

`shrink the hot data`

3

Vertical scale

`last, more RAM`

BREAKING POINT 02 · READ LOAD

Reads to the replica, writes to the primary

Signal: CPU is high, SELECT dominates, `cl_waiting` piles up in pgBouncer. `sticky` sends reads that follow a write within the same request to the primary.

config/database.php

```php
'pgsql' => [
  'driver'  => 'pgsql',
  'read'  => ['host' => ['10.0.0.2']],   // replica
  'write' => ['host' => ['10.0.0.1']],   // primary
  'sticky' => true,
  // ...shared settings
],
```

THE COST

The new problem the replica brings

LAG	Replication lag: the replica falls behind; the read-after-write guarantee is lost.
MONITOR IT	In pg_stat_replication, read the replay_lsn difference as lag_bytes.
PRIMARY	Reads that need consistency, such as balance, stock and permissions, stay on the primary.
LIMIT	A read replica adds nothing to write capacity.

BREAKING POINT 03 · SINGLE TABLE

A table split by time

Signal: `n_dead_tup` keeps piling up, `last_autovacuum` falls behind. Instead of creating partitions by hand, you can use `pg_partman`.

events.sql

```sql
CREATE TABLE events (
  id          bigserial,
  occurred_at timestamptz NOT NULL,
  payload     jsonb       NOT NULL
) PARTITION BY RANGE (occurred_at);

CREATE TABLE events_2026_05 PARTITION OF events
  FOR VALUES FROM ('2026-05-01') TO ('2026-06-01');
```

GAIN AND CONDITION

What does partitioning buy you?

Partition pruning

The query looks only at the relevant partition.

Cheap deletion

Old data goes away with DROP TABLE events_2026_01.

Bounded vacuum

Vacuum scans the partition, not the whole table.

Primary key

It must include the partition key: (id, occurred_at).

BRIN

A small, cheap index for time-ordered data.

BREAKING POINT 04 · WRITE LOAD

Signal: I/O wait, checkpoint and WAL pressure

1

Reduce amplification

drop idx_scan = 0 indexes, fillfactor/HOT, batch insert

2

Take out append data

append-heavy data is separated from the primary

3

Split the data domain

last: functional sharding

REALISTIC EXPECTATION

Most systems never reach point 4.

Don't rush: most systems never reach point 4, and most of those that do arrive early because they skipped point 1 (a missing index, a bloated working set).

BREAKING POINT 05

A separate data tier

THE MOVE The data-heavy domain is moved to a separate PostgreSQL instance.

MIGRATION pgBackRest restore, logical replication and a short cutover.

COST A second backup, monitoring and upgrade; no JOIN across the two data sets.

RULE The decision to separate is made by the data profile, not by the org chart.

WRONG DIAGNOSIS

Wrong breaking points

WHAT YOU SEE	WHAT IT REALLY IS	THE RIGHT MOVE
"The database is slow"	N+1: the app fires 200 queries	Eager loading; count queries
"We need a replica"	One query lacks an index	EXPLAIN ANALYZE, add the index
"We need sharding"	Replicas, partitioning untried	Exhaust points 2 and 3 first
"Move to NoSQL"	jsonb, BRIN, partitioning unused	Use PostgreSQL to the fullest
"Cache the writes"	A cache cuts reads, not writes	Look at point 4

THE COMMON ERROR PATTERN

None of them adds write capacity.

A read replica scales reads. A cache reduces reads. Partitioning tames a single table. None of them increases write throughput.

JSONB · LISTEN/NOTIFY · PARTITIONING · BRIN · LOGICAL REPLICATION

You used 20%. That's not a limit.

Most teams use 20% of PostgreSQL and conclude that "PostgreSQL isn't enough".

SUMMARY

From cheapest to most expensive

1

Working set

`index + RAM`

2

Read load

`read replica`

3

Single table

`partitioning`

4

Write load

`amplification + separation`

5

Data-heavy domain

`separate data tier`

SECTION 02

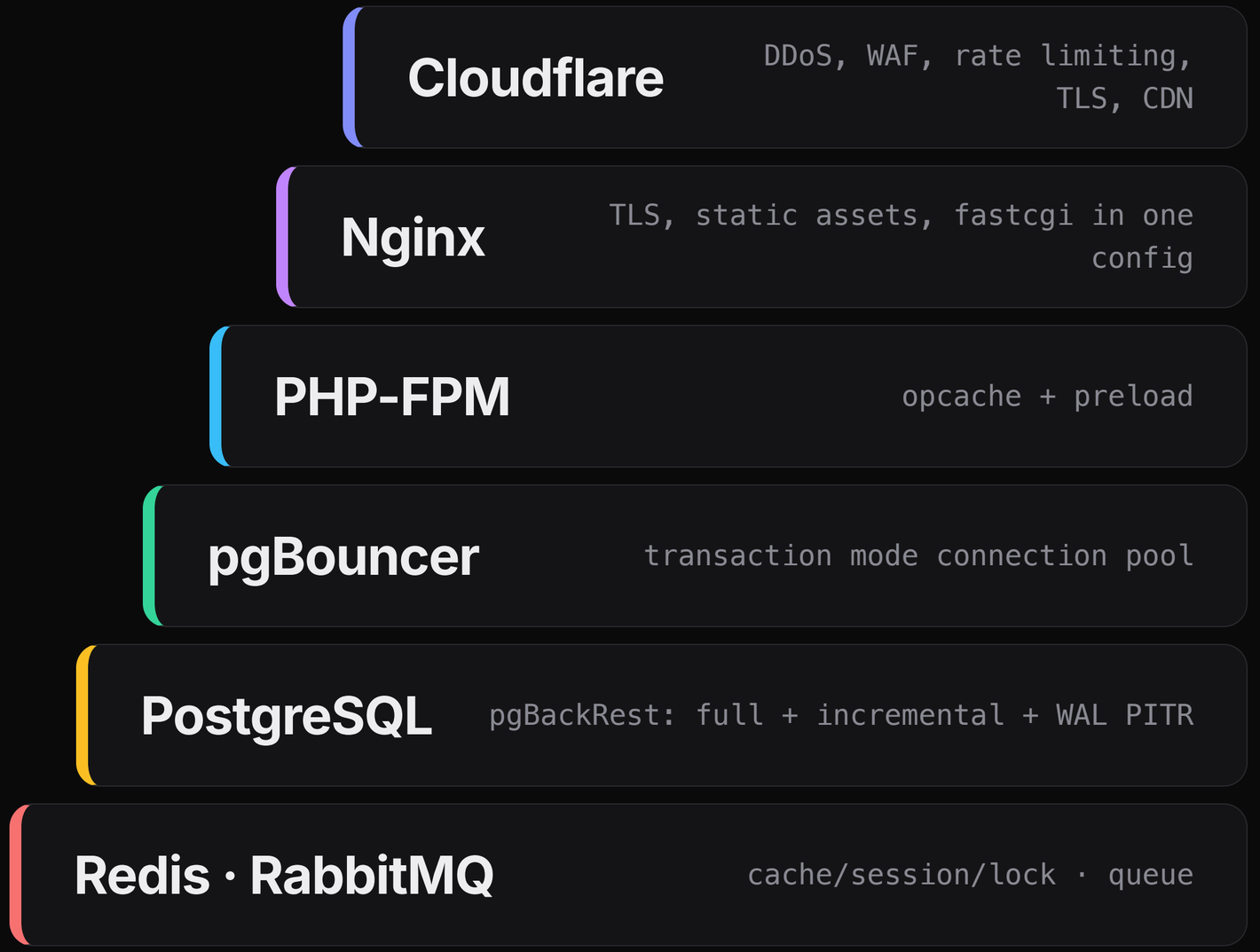
Laravel stack

I'm not saying "everything is necessary" — I'm someone who argues for keeping it minimal until you understand it's needed — but what each component buys you is concrete.

ANATOMY

Why is each component there?

Requests reach PHP-FPM through Cloudflare. PHP-FPM connects to three backing services separately: PostgreSQL through pgBouncer, Redis and RabbitMQ. Workers run with Horizon under Supervisor.



REDIS

One instance, three hats

Cache (`Cache::remember`), session (`SESSION_DRIVER=redis`) and lock (`Cache::lock`) all run safely on a single Redis, as long as these three settings are in place.

redis.conf

```
maxmemory-policy volatile-lru
appendonly no
save 900 1 300 10 60 10000
```

QUEUE (OPTIONAL)

RabbitMQ versus the Redis queue

	Redis	RabbitMQ
PERSISTENCE	Serves from memory; goes to disk via RDB or AOF	✓ Writes persistent messages to disk
REDELIVERY	Depends on the retry_after timeout	✓ Depends on the consumer's connection
ROUTING	The routing decision stays in the application	✓ Exchanges decide
MONITORING	Horizon; limited to what the application dispatches	✓ A broker-level management panel

WHAT HAPPENS WHEN REMOVED

What do you lose if you remove each piece?

COMPONENT	WHAT HAPPENS WHEN REMOVED
pgBouncer	In connection storms PostgreSQL connections run out and requests throw 500
Supervisor	Workers don't come back after a crash, you have to set up alarms
Horizon	Visibility collapses, you answer "why is the queue slow" blind
opcache	Every request reads and parses the PHP file from disk, ~5x slowdown
pgBackRest	Only pg_dump remains: no PITR, a day's backup as your error margin
Redis lock	You're open to race conditions, you're left without a distributed-safe mutex

BORING STACK

Measurably useful, easy to change.

The strength of a boring stack is exactly this: every piece is quantitatively valuable and easy to change.

Thank you

muhammetsafak.com/engineering/

Measure the data profile first, then pick the cheapest intervention.

