

TECH & AI CASE STUDY - ONLINE MARKETPLACE - 2026

Postgres Scaling That Survived 40x Traffic Growth With Zero Downtime

Based on a delivery engagement with Cartful Marketplace

40x

TRAFFIC GROWTH ABSORBED

4s to <300ms

P99 LATENCY AT PEAK

0 in 18 months

MAINTENANCE WINDOWS

+14%

CHECKOUT CONVERSION

A fillable practice edition: the full story, 8 actions,
6 reflections and a 90-day plan - work it with your team.

Free - no email required
emersontechnology.com

MAKE THIS PRACTICE BOOK YOURS

Work it, do not just read it

This practice book turns the Cartful Marketplace case study into a working session for your team. Read the full story first - what happened, what was built and what it changed. Then tick the 8 actions you can commit to, write short answers to the 6 reflections, and finish with a 90-day plan. Every field in this PDF is fillable: open it in Adobe Acrobat Reader, Chrome, Edge or macOS Preview, type directly into the boxes and save.

YOUR NAME

ROLE

COMPANY

TEAM / DEPARTMENT

DATE

EMAIL (OPTIONAL)

MY GOAL FOR THIS PRACTICE SESSION

- 01

The story - What happened at Cartful Marketplace - before, during and after the project.
- 02

The results - The numbers that changed, plus industry benchmarks from public research.
- 03

The 8-action checklist - Tick boxes as your organization completes each step. Owner and date fields included.
- 04

The 6 reflections - Fillable questions that pressure-test your own roadmap.
- 05

My 90-day plan - Three fillable phase cards that turn this case study into your roadmap.

RUN IT AS A 90-MINUTE TEAM SESSION

25 min READ Read the story together. Ask everyone to note the one number that surprised them most.	40 min PRACTICE Work the 8-action checklist and the 6 reflections. Assign an owner and a date to every action you commit to.	25 min COMMIT Fill in the 90-day plan. One goal per phase, one number that tells you it worked. Book a review date before you leave the room.
--	--	---

THE STORY

What happened, and why it worked

01 The Company

Cartful Marketplace is a peer-to-peer platform for secondhand goods — a place where individual sellers list items, buyers browse, search, negotiate and check out, and both sides interact through messaging, watchlists and reviews. Marketplace dynamics give the platform a distinctive load signature: activity is social and event-driven, concentrated around weekend evenings when browsers are relaxed and listings are fresh, and amplified by seller behaviors such as price drops and restocks that trigger waves of notifications. Success in a marketplace is a two-sided trust problem, and the platform's database sits at the center of it: every search result, listing view, watchlist update and checkout touches Postgres.

The company's growth story is the kind founders celebrate and databases survive reluctantly. After a viral category expansion, traffic grew roughly 40x in 18 months — a trajectory that outpaced every capacity assumption baked into the original single-instance architecture. Stack Overflow's annual developer surveys have reportedly ranked PostgreSQL the most-used database for several consecutive years, ahead of MySQL and every commercial alternative, which is precisely why Cartful chose it: a mainstream, deeply understood database with a mature scaling toolbox. The 40x growth did not argue for a different database; it argued for using the one Cartful already ran to its full architectural depth.

02 Where Things Stood

A single PostgreSQL instance began showing every strain symptom at once. p99 latency spiked past four seconds on weekend evenings — the exact hours when marketplace activity, and therefore revenue, peaked. Lock contention on the listings table stalled checkouts as writes queued behind each other. Analysts' ad-hoc queries, run directly against the primary, routinely starved it of resources, so a seller dashboard could degrade checkout performance for every buyer on the platform. Connection exhaustion during email campaigns became a weekly incident: each campaign wave opened fresh connections faster than the instance recycled them, and the platform's own growth marketing repeatedly throttled its transactional traffic.

The operational consequences compounded the technical ones. One Sunday outage in prime trading hours cost an estimated \$240,000 in GMV and pushed sellers to competitors for the rest of the week — a lasting reminder that in a two-sided marketplace, downtime does not merely lose transactions, it shakes the trust that keeps both sides returning. Gartner's widely cited analysis reportedly put the average cost of IT downtime at \$5,600 per minute — roughly \$300,000 per hour for a typical enterprise — and Cartful's Sunday fit that arithmetic uncomfortably well. Schema changes required coordinated downtime windows, so the product team shipped fewer updates precisely when the business needed speed, and the release calendar had become a function of the database's fragility rather than the roadmap's ambition.

03 The Challenge

The challenge was to scale the database through continued hypergrowth without a rewrite, without a maintenance-window habit, and without freezing the product roadmap. Rewrites were explicitly off the table: Cartful's Postgres schema was healthy, its team's expertise was deep, and the failure modes were architectural-usage problems — connection storms, monolithic tables, an overloaded primary — rather than wrong-tool problems. Alibaba has reportedly sustained hundreds of thousands of orders per second during Singles' Day peaks — the benchmark marketplace-scale architecture is measured against — and while Cartful's volumes were far below that, the engineering principle behind the benchmark applies at every scale: scale by removing contention, not by buying a bigger single box.

Four specific bottlenecks had to fall. First, connection storms: thousands of direct connections from application instances during campaign waves exhausted the primary before any query ran. Second, table scale: the listings and events tables had grown past the size where indexes stay small, vacuums stay fast and maintenance stays predictable. Third, workload mixing: search, analytics and seller dashboards competed with checkout writes for the same primary. Fourth, the migration ritual: schema changes demanded downtime windows, which made every product iteration a negotiation with operations. Each of these had a well-understood Postgres-native answer; the challenge was sequencing and operationalizing them so the platform kept trading while the work happened underneath it.

04 Discovery and Diagnosis

Emerson Technology opened with a two-week diagnostic that treated the database as a live system under load rather than as a schema on a whiteboard. Query and connection telemetry captured a full weekly cycle, including the weekend-evening peaks that defined the problem. The analysis quantified each bottleneck precisely: connection counts during campaign waves, lock-wait times on the listings table, the resource share consumed by ad-hoc analyst queries, and the p99 latency curve that degraded from weekday normal to weekend crisis. Slow-query budgets did not exist, so regressions arrived silently and were discovered by user complaints rather than by alerting.

The migration-history review was equally revealing. Past schema changes had required coordinated downtime windows, and at least one had failed mid-window and rolled back — an event that consumed a weekend and taught the organization to fear schema work. Rehearsal discipline was absent: migrations were tested against staging datasets a fraction of production size, so their real-world timings were unknown until they ran against live tables. The diagnosis closed with a four-axis scaling plan — pooling, partitioning, replica routing and cache offloading — plus a fifth, procedural axis: a zero-downtime migration workflow with production-scale rehearsals, so that the platform's growth would never again be held hostage to a maintenance window. Sequence mattered throughout: pooling first because it was cheap and immediate, then read isolation, then the table surgery, with the migration discipline in place before any of it needed a schema change.

05 The Solution

We scaled the database along four axes without a rewrite, then wrapped all future schema change in a zero-downtime workflow so the scaling could continue indefinitely.

Pooling and partitioning

Connection pooling through PgBouncer in transaction mode absorbed the traffic storms, cutting effective connections from thousands to hundreds and eliminating the campaign-wave incidents almost immediately — most database-overload incidents are connection storms, and pooling removes them for near-zero cost in a day. The heavy listings and events tables moved to declarative partitioning by date, keeping indexes small, vacuums fast and maintenance predictable, with `pg_partman` automating the partition lifecycle so the discipline survives without weekend heroics. Partitioning also localized the lock contention that had stalled checkouts: writes now contend within a partition rather than across a monolithic table, which is what returned weekend-evening checkout latency to sanity.

Replicas, cache and queues

Read replicas behind a routing layer now serve search, analytics and seller dashboards, isolated from the write primary on Amazon RDS with Multi-AZ failover — so one misbehaving ad-hoc report can never again stall checkout for customers. Hot rows — counters, watchlists, presence — moved to Redis, absorbing the majority of read bursts with simple cache-aside patterns at low cost, and Kafka queues background work such as image

processing and notifications so slow consumers queue rather than cascade into the request path. The result is a deliberate division of labor: Postgres does transactional truth, Redis does hot reads, Kafka does asynchronous work, and each system operates in the regime where it is strongest.

The zero-downtime migration discipline

Every schema change now runs through a structured workflow: expand, backfill in batches, then contract, validated by rehearsal against production-scale snapshots with timings recorded. Additive steps ship first so old and new code coexist; batched backfills move data without long locks; cleanup passes remove the old structure only after the new one has proven itself in production. The workflow turned migrations from weekend rituals into ordinary deployments — downtime windows became optional engineering choices rather than mandatory rituals. Slow-query budgets and p99 alerting keep regressions visible before users feel them, and a monthly database-cost-per-order metric keeps the scaling honest, proving that efficiency is improving rather than merely being relocated.

06 The First 90 Days

The first month delivered the fastest wins with the lowest risk. PgBouncer went in front of the primary in week two, in transaction mode, with the application connection strings updated behind a config release — and the next email campaign passed without a connection incident for the first time in months. Read replicas were provisioned on RDS with Multi-AZ failover in weeks three and four, the routing layer went live, and analytics and seller dashboards moved off the primary; the difference was felt within days, as weekend-evening p99 began retreating from its four-second spikes.

Weeks five through eight executed the table surgery. The listings and events tables were partitioned using a carefully sequenced online migration — new partitioned structures built alongside the originals, data backfilled in batches, reads validated, then traffic cut over — all through the expand-contract workflow that had been formalized in week four with its first production-scale rehearsal. The rehearsal mattered: it surfaced a backfill that ran hours longer at production scale than on staging data, and the batching strategy was adjusted before the real run rather than during it. Weeks nine through twelve moved hot rows to Redis, queued background work through Kafka, and tuned the p99 alerting and slow-query budgets with thresholds agreed between engineering and product. By day 90, the four scaling axes were live, the platform had passed two peak trading weekends without a database page, and the migration workflow had shipped three schema changes — none of which anyone outside engineering noticed, which was precisely the point.

07 The Results

The platform has absorbed continued growth past 40x baseline traffic with p99 latency under 300 milliseconds and zero maintenance windows in 18 months. Checkout conversion improved 14% once weekend latency spikes disappeared — the direct revenue echo of a database that no longer stalls at the exact hours customers trade. Analysts stopped taking the primary down, literally, and failed-migration weekends are gone: every schema change ships behind the expand-contract workflow, rehearsed at production scale. Infrastructure cost per order fell 27% as replicas replaced brute-force vertical scaling, and peak trading evenings that once required war rooms now pass without a single database page.

METRIC	BEFORE	AFTER
Traffic growth absorbed	-	40x
p99 latency at peak	4 s	Under 300 ms
Maintenance windows	-	0 in 18 months

METRIC	BEFORE	AFTER
Checkout conversion	Baseline	+14%

The product organization felt the change as keenly as engineering. With downtime windows eliminated, schema work stopped being a negotiation, and the team shipped updates faster during the growth surge than it had before it — the opposite of the old pattern, where speed was scarcest exactly when the business needed it most. Trust recovered on both sides of the marketplace: sellers stopped experiencing listing lag during peak hours, buyers stopped abandoning slow checkouts, and the Sunday-outage story faded from sales conversations into engineering history. The scaling architecture also changed the growth conversation internally — leadership now plans category expansions with confidence, because the database layer has demonstrated headroom and a playbook rather than a ceiling.

08 What Your Business Should Do

Cartful's experience compresses into a scaling sequence that most Postgres-backed platforms can follow in order. Pool connections before adding hardware: deploy PgBouncer or equivalent transaction-mode pooling before buying bigger servers, because most database-overload incidents are connection storms that pooling eliminates for near-zero cost in a day. Partition your biggest tables, splitting high-volume tables by date or tenant so indexes stay small, vacuums stay fast and maintenance stays predictable, with automated partition tooling keeping the discipline sustainable without weekend heroics. Route reads to replicas — move search, dashboards and analytics onto read replicas isolated from the write primary — so one misbehaving ad-hoc report can never stall checkout for customers. Adopt expand-contract migrations, shipping schema changes as additive steps, batched backfills and later cleanup passes rather than blocking alters; downtime windows become optional engineering choices instead of mandatory rituals.

Then build the operating habits that keep the scaling honest as you grow. Move hot rows to cache, keeping counters, watchlists and session data in Redis rather than hammering Postgres on every request — simple cache-aside patterns absorb the majority of read bursts at low cost. Rehearse migrations at production scale, validating every change against a production-sized snapshot with timings, since a migration that runs in seconds on staging data can run for hours on real volume. Alert on p99 rather than averages, watching 99th-percentile query times, lock waits and connection saturation, because average latency hides exactly the tail spikes that cost customers during peaks. And measure database cost per order, tracking infrastructure cost per transaction month over month — falling unit costs prove your scaling is genuinely efficient rather than merely expensive in new ways. As you assess your own platform, sit with the questions this engagement raised: what your database's p99 latency is on your busiest evening and who is watching it; which report or job could take your primary down tomorrow if a user got impatient; how much warning you would have before connection exhaustion became visible to customers; and what zero-downtime deployments would change about how often, and how boldly, your team ships. Work through these eight steps with your team using the action checklist in this practice book, and revisit the reflection questions each quarter — scaling is not a project that ends, but a discipline that compounds.

REAL-WORLD PROOF & BENCHMARKS

Stack Overflow's annual developer surveys have reportedly ranked PostgreSQL the most-used database for several consecutive years, ahead of MySQL and every commercial alternative. - Stack Overflow, 2024

Gartner's widely cited analysis reportedly put the average cost of IT downtime at \$5,600 per minute — roughly \$300,000 per hour for a typical enterprise. - Gartner

Alibaba has reportedly sustained hundreds of thousands of orders per second during Singles' Day peaks — the benchmark marketplace-scale architecture is measured against. - Alibaba, reported 2020

THE PRACTICE PAGES

The 8-action checklist

Work through the list with your team. Tick each box as your organization completes the step, and fill in an owner and a target date so every action has a name and a deadline.

☐

1. Pool connections before adding hardware

Deploy PgBouncer or equivalent transaction-mode pooling before buying bigger hardware. Most database-overload incidents are connection storms that pooling eliminates for near-zero cost in a day.

OWNER

TARGET DATE

STATUS

--	--	--

☐

2. Partition your biggest tables

Split high-volume tables by date or tenant so indexes stay small, vacuums stay fast and maintenance stays predictable. Automated partition tooling keeps this sustainable without weekend heroics.

OWNER

TARGET DATE

STATUS

--	--	--

☐

3. Route reads to replicas

Move search, seller dashboards and analytics onto read replicas isolated from the write primary. One misbehaving ad-hoc report should never be able to stall checkout for customers.

OWNER

TARGET DATE

STATUS

--	--	--

☐

4. Adopt expand-contract migrations

Ship schema changes as additive steps, batched backfills and later cleanup passes, never as blocking alters. Downtime windows become optional engineering choices rather than mandatory rituals.

OWNER

TARGET DATE

STATUS

--	--	--

☐

5. Move hot rows to cache

Keep counters, watchlists and session data in Redis rather than hammering Postgres on every request. Simple cache-aside patterns absorb the majority of read bursts at low cost.

OWNER

TARGET DATE

STATUS

--	--	--



6. Rehearse migrations at production scale

Validate every change against a production-sized snapshot with timings. A migration that runs in seconds on staging data can run for hours on real volume.

OWNER

TARGET DATE

STATUS

--	--	--



7. Alert on p99, not averages

Average latency hides exactly the tail spikes that cost customers during peaks. Alert on 99th-percentile query times, lock waits and connection saturation instead of averages.

OWNER

TARGET DATE

STATUS

--	--	--



8. Measure database cost per order

Track infrastructure cost per transaction as volume grows, month over month. Falling unit costs prove your scaling is genuinely efficient rather than merely expensive in new ways.

OWNER

TARGET DATE

STATUS

--	--	--

AFTER THE CHECKLIST: OUR TOP 3 PRIORITIES

From the eight actions above, write down the three you will start first - and the single number you expect each one to move. Three priorities with owners beat eight without any.

PRIORITY 1 NUMBER IT SHOULD MOVE	PRIORITY 2 NUMBER IT SHOULD MOVE	PRIORITY 3 NUMBER IT SHOULD MOVE
--	--	--

THE PRACTICE PAGES

Reflections for your team

Use these questions in your next planning session and type short answers into the boxes. The gaps you find are usually the fastest wins.

Q1. What is your database's p99 latency on your busiest evening — and who is watching it?

Q2. Which report or job could take your primary database down tomorrow if a user got impatient?

Q3. How much warning would your team have before connection exhaustion became visible to customers?

Q4. When did you last rehearse a schema migration against a production-sized snapshot of data?

Q5. If traffic tripled next quarter, which component fails first — and is that failure gradual or sudden?

Q6. What would zero-downtime deployments change about how often — and how boldly — your team ships?

THE PRACTICE PAGES

My 90-day plan

Finish the session by committing to a plan. Three phases, one goal each, one number that tells you it worked. Bring this page to your next leadership review - or to a free consultation with Emerson Technology, and we will pressure-test it with you.

Days 1-30 Focus: audit and alignment

PRIMARY GOAL**KEY ACTIONS****HOW WE MEASURE SUCCESS****OWNER**

Days 31-60 Focus: build and prove

PRIMARY GOAL**KEY ACTIONS****HOW WE MEASURE SUCCESS****OWNER**

Days 61-90 Focus: scale and measure

PRIMARY GOAL

KEY ACTIONS

HOW WE MEASURE SUCCESS

OWNER

Tip: save a copy per team so each group keeps its own answers. Revisit the reflections every quarter - the questions stay the same even as the numbers move.

BOOK THE REVIEW BEFORE YOU CLOSE THE BOOK

Plans without a review date drift. Before you close this practice book, agree on when you will come back to these pages, who will run that session, and when you will check the numbers again. If you want an outside pair of eyes, bring this page to a free consultation with Emerson Technology and we will pressure-test the plan with you.

REVIEW DATE	REVIEWED BY	NEXT CHECK-IN
------------------------------------	------------------------------------	--------------------------------------

Ready for a transformation like Cartful Marketplace's?

Emerson Technology designs and builds the data, AI, cloud and automation systems behind results like these. On a free consultation call we will map this exact playbook to your business - your data, your constraints, your numbers.

Book a free consultation

support@emersontechnology.com

or start online: emersontechnology.com/contact