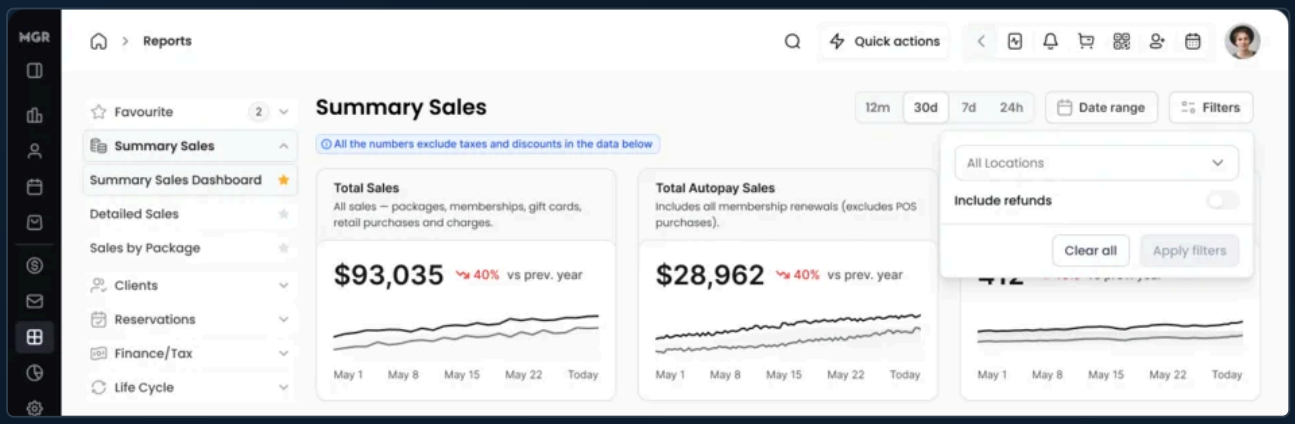
PLATFORM ENGINEERING

# Three screens, three revenue numbers, all correct

Nineteen reports had accreted over five years, each joining live tables at query time and each encoding its own answer to what counts as revenue. We built the thing they had been improvising — one append-only ledger of what money moved and when — then migrated five years of live financial reporting onto it one tenant at a time, without a cutover.

| 1 | 46 | 5 | 2¢ |
|---|---|---|---|
| shared definition of revenue, not 19 | endpoints across 9 report families | years of history replayable on demand | of drift before an engineer is paged |



CLIENT

Fitness studio management SaaS (anonymised)

ENGAGEMENT

Re-platforming inside a long-running platform partnership

INDUSTRY

B2B SaaS — fitness & wellness

TIMELINE

March–July 2026, in four overlapping phases

DELIVERED

Append-only revenue ledger, rebuild tooling, 8 bridge services, 46 V2 endpoints, 3 V1 families retired

## The problem: three screens, three revenue numbers

A studio owner opens their dashboard and sees a revenue tile. They open the sales report for the same period and see a different figure. They ask the in-product assistant and get a third.

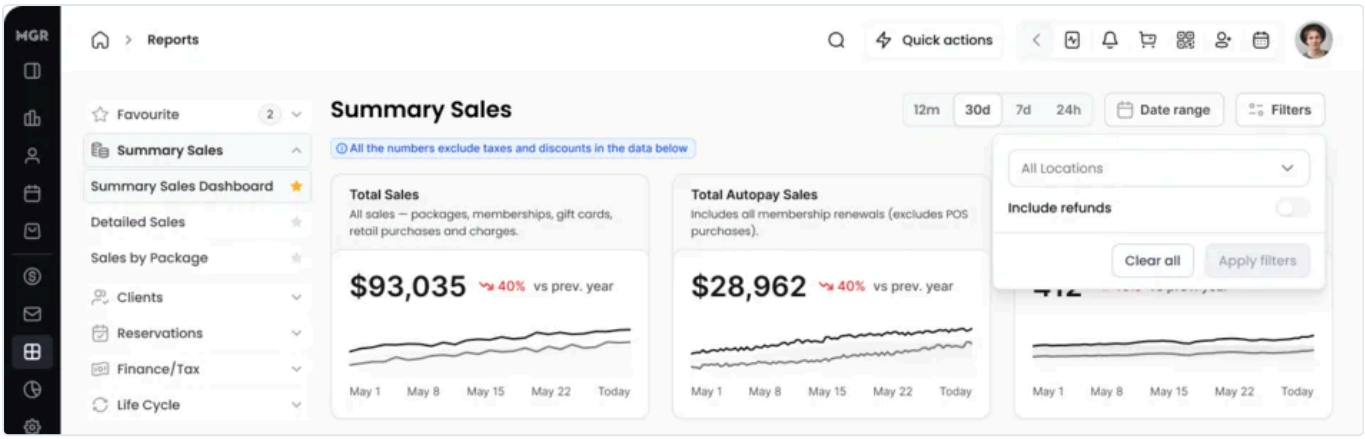
All three are correct. That is what makes it a serious problem.

The dashboard tile summed completed payment transactions by capture date and cached the result for two hours. The sales report summed order totals by purchase date. The assistant used the same query as the tile. Each had made a defensible decision about four things — whether refunds are netted or excluded, whether tax counts as revenue, whether a sale belongs to the day it was *captured* or the day it was *initiated*, and how fresh the number needs to be — and no two had made the same decisions.

This is the predictable end state of a reporting system that grew one report at a time. Nineteen report endpoints had accreted in this platform since April 2021, each a bespoke command joining live operational tables at query time: payments, orders, line items, package instances, profiles, reservations. When a feature shipped and someone needed a number about it, a report was added next to that feature. The routes file still shows the seams, with comment headers grouping reports by the product area that prompted them.

Every one of those reports encoded its own answer to the four questions above, in a hand-written join. There was no shared definition of revenue to disagree with, because there was no shared definition at all.

So the fix was not a faster query or a better chart library. It was to build the thing the reports had been improvising — **one record of what money moved, when, and against what** — and then to migrate five years of live reporting onto it without ever being wrong in front of a customer.




The filter panel is open over the third tile, and it contains the control that used to be an argument: *include refunds*. Notice the line under the title too — the report states its own definition on screen. Both are only possible once one place decides what a sale is.

This is the fifth write-up from this platform, after the workflow automation engine, the AI notification templates, the agentic assistant and the in-app CMS. It is the least glamorous of the five and the one the other four depend on, because every one of them either moves money or reports on it.

---

## Act one: the ledger

### One row per thing that happened

The revenue ledger is an append-only fact table. One row is one entry type, for one payment transaction, for one order line item. A sale of three items on a single payment writes three rows. Refunding one of them appends a fourth row rather than touching the first three.

There are exactly three entry types — sale, refund and gift-card redemption — and the absence of a fourth is the design. There is no *void*, no *adjust*, no *correct*. Every correction is expressed as a new entry of an existing type, because the alternative is editing history.

The reporting dimensions are denormalised onto the row itself: location, staff member, client, product and product type, payment method, card brand, and the channel the sale came through. This is a star-schema fact table living inside a transactional PostgreSQL database, and it is why a question like "sales by location by month" is now a single-table aggregate rather than a five-table join. Composite indexes carry the shape `(business_account_id, <dimension>, event_at)`, which is not a coincidence — it is every report the product offers, expressed as an index.

### The most important column is the one that is easy to miss

Two timestamps, deliberately. `event_at` records when the money moved. `created_at` records when the row was written. They are different columns because they answer different questions, and every report filters and buckets on `event_at`.

Without that separation, everything in the next section is impossible: replaying a two-year-old payment would file its revenue under the day of the replay, and a rebuilt ledger would be worthless. Worth crediting where this came from — the old dashboard metric already carried a comment explaining why it used

capture time rather than creation time, because recurring payments would otherwise all land in their first month. The insight existed before the ledger. The ledger generalised it into schema.

### Immutable, enforced twice

The model raises on update and on destroy. That is the polite version. The real one is a PostgreSQL trigger, because a guarantee that lives only in application code is a convention rather than a guarantee — it does not survive a console session, a raw `UPDATE`, or a future developer with a deadline.

```
RAISE EXCEPTION 'revenue_ledger_entries is append-only:
                 UPDATE and DELETE are prohibited';
```

### The best three minutes in the project's history

Three migrations, all dated 8 March 2026, timestamped one second apart. One working session.

**The first** creates the table with the trigger above, raising on both statements. **The second** adds a second unique index, for reasons in the deep dive. **The third** is named `AllowDeleteOnRevenueLedgerEntries` and relaxes the trigger to block `UPDATE` only.

Read that sequence again, because it is the whole architecture in three files. They built the table as immutably as PostgreSQL allows, and within hours hit the contradiction: **a table you may never clear is a table you can never rebuild.** If the derivation logic is ever wrong, or a new dimension is ever needed, absolute immutability makes the error permanent.

So the position they landed on is more precise than "immutable", and better:

> `UPDATE` is prohibited forever, at the database, for everyone. `DELETE` is permitted, so that the entire table can be discarded and re-derived from the source records.

No row is ever edited. The whole table is disposable. Those are compatible, and together they are stronger than immutability alone — the ledger is not a precious artefact to be protected, it is a projection that can always be regenerated from the operational records that produced it.

The migration is reversible, and its `down` restores the stricter trigger, so both intents stay in the file. The rebuild task's header comment states the reasoning out loud, including that a bulk delete bypasses the model callback and that the trigger deliberately does not cover `DELETE`. The escape hatch is documented, not accidental.

## Engineering deep dive

Why one unique index was not enough, largest-remainder rounding in BigDecimal, a ledger that checks its own arithmetic, rebuild tooling that reuses the live write path, and the retroactive dimension that proved it.

### Two unique indexes, because PostgreSQL treats NULL as unknowable

Ledger entries are written by background workers, and workers retry. Writing the same sale twice would silently overstate revenue — the worst class of bug in a financial system, because nothing

crashes.

```
UNIQUE (user_transaction_id, order_line_item_id, entry_type)
UNIQUE (user_transaction_id, entry_type) WHERE order_line_item_id IS NULL
```

The first index looks sufficient and is not. In PostgreSQL, `NULL` is not equal to `NULL`, so a unique index containing a nullable column does not constrain rows where that column is null — and plenty of entries have no line item, because fees, penalties, account credits and gift card redemptions are not order lines. Under the first index alone, precisely those entries could be duplicated without limit. The partial index added minutes later closes exactly that gap.

The write path then leans on the database rather than second-guessing it: attempt the insert, rescue the uniqueness violation, log and move on. Catching the violation rather than checking first is the right way round under concurrency — a retry that already wrote two of three rows inserts the third and skips the two, with no read-modify-write race to lose.

### The pennies add up exactly

Splitting one payment across several line items is where financial reporting usually acquires its rounding drift. Divide 100 across three items and naive rounding gives you 33.33 three times, which is 99.99, and now the report's line items do not sum to the order total.

The distributor implements largest-remainder rounding in `BigDecimal`: floor every share, count the leftover pennies, then hand them out to the items with the largest discarded fractions.

```
sorted = raw.sort_by { |r| -(r[:raw] - r[:floored]) }
sorted.each do |r|
  break if remainder ⩽ 0
  r[:floored] += BigDecimal("0.01")
  remainder   -= BigDecimal("0.01")
end
```

The parts sum to the whole, exactly, always. Thirty-odd lines that mean a studio's line-item breakdown reconciles to their order total to the cent.

### The ledger checks its own arithmetic

Every write validates itself against the payment that caused it. If the rows a service just derived do not sum back to the transaction they were derived from, within two cents, an engineer gets the transaction id and the exact delta in Sentry.

It deliberately does not roll back. A slightly wrong row someone is actively told about beats a missing row nobody knows to look for. This is also where the platform's definition of revenue now lives, in one expression rather than nineteen: net amount plus tax.

### Rebuildable, and already rebuilt

The rebuild is not a script somebody would write if it were ever needed. It is production infrastructure built on the `maintenance_tasks` gem, so it has an operator UI, a saved cursor, pause

and resume, and throttling. Six documented steps: delete every row in batches, then replay order payments, non-order sale transactions, order refunds, account credits, and gift card redemptions.

Three properties make it trustworthy rather than theatre.

**It reuses the live write path.** The replay steps call the same service classes the live workers call. There is no second implementation of "how do we derive entries from an order" that can drift from the first — which is the failure mode that makes most rebuild tooling untrustworthy after six months.

**It is resumable and timeout-aware.** Work is split across job-iteration ticks explicitly so no single statement hits the statement timeout, and an interrupted run continues from its cursor rather than starting over or double-writing.

**Each source is independently runnable.** The per-source tasks default to filling gaps only, skipping rows that already exist, so they are safe to run against a live table. Only the orchestrator wipes.

### The extensibility claim, demonstrated rather than asserted

"Rebuildable in case it needs extending later" is normally a hypothetical. Here it already happened, and the dates are in the repository.

The channel a sale came through — admin web, admin mobile, consumer web, consumer app, or the system itself — was not in the original table. It was added two months later, and it sits after `created_at` in the schema, the signature of a later `add_column`.

Populating it for every historical row meant that, since `UPDATE` is prohibited at the database, there was exactly one way to do it: delete the entire ledger and re-emit every row through the current derivation logic. That is what the orchestrator exists for. A sixth index was added for the new dimension, and the by-channel breakdown shipped as a V2 endpoint.

New dimension needed, column added, ledger rebuilt from source records, report shipped on it. The architecture's central claim, exercised end to end in production, two months after it was built.

## Act two: migrating five years of reporting without a cutover

A new table is the easy half. The hard half is moving nineteen live reports onto it in a system where being wrong once costs more trust than being late by a quarter.

The obvious approach is to build a new reporting stack on the new table and cut over. That was not the approach.

1. **Build it beside the live system**

   March 2026. The ledger, its write path, its workers and its backfill tasks all landed while every existing report carried on querying exactly what it always had. Nothing observable changed; the ledger simply started accumulating, and the backfills filled in history.

2. **Bridge the old reports onto it, one tenant at a time**

   March–April 2026. The ledger's first consumer was not a new report — it was the **old** ones. Eight services reproduced each existing report's output from the ledger instead of from live joins. Same endpoints, same response shape, same contract. Only the data source changed, and it changed per tenant behind a feature flag, with the legacy query preserved intact underneath.

3. **Build the new surface on a proven substrate**

   May–June 2026. Only once the ledger was serving real reports for real tenants did the V2 API get built: nine report families, 46 endpoints, one controller per family.

4. **Delete the old thing, including the bridge**

   24 July 2026. Around 1,335 lines deleted: three V1 finance reports, their exporters, controller actions, routes and specs — and the three bridge services with them. The bridge was scaffolding from the beginning.

---

Step 2 is the part worth copying. Re-platforming the numbers underneath a financial report is not a deploy, it is a rollout.

Because the API contract never moved, the comparison was apples to apples: the same request, two derivations, one account moved forward at a time, and the old path one boolean away for as long as it took to trust the new one. No customer was the test subject for an architecture change they had not asked for.

Step 4 deserves its own note. Three reports completed the full cycle — built beside, bridged behind a flag, rebuilt on a new surface, then deleted along with their own migration apparatus. Scaffolding that is not removed becomes architecture by default, and the commit that removed it names the bridge services explicitly.

## The shim, and why it is not a redirect

V2's URLs were reorganised mid-build, from flat actions to per-family nesting. Twenty-six aliases keep the old ones working, generated in a loop, with a comment that explains itself: these are real route aliases and *not* 301 redirects, because API clients do not all follow redirects and a 30x reads as an error in most JSON consumers. Each legacy URL invokes the same controller action as its nested counterpart.

The comment also carries its own removal condition — delete this block once all known consumers have migrated — and a rule stopping it becoming a dumping ground: do not add new endpoints here. Small thing, but this is what maintainable looks like in practice.

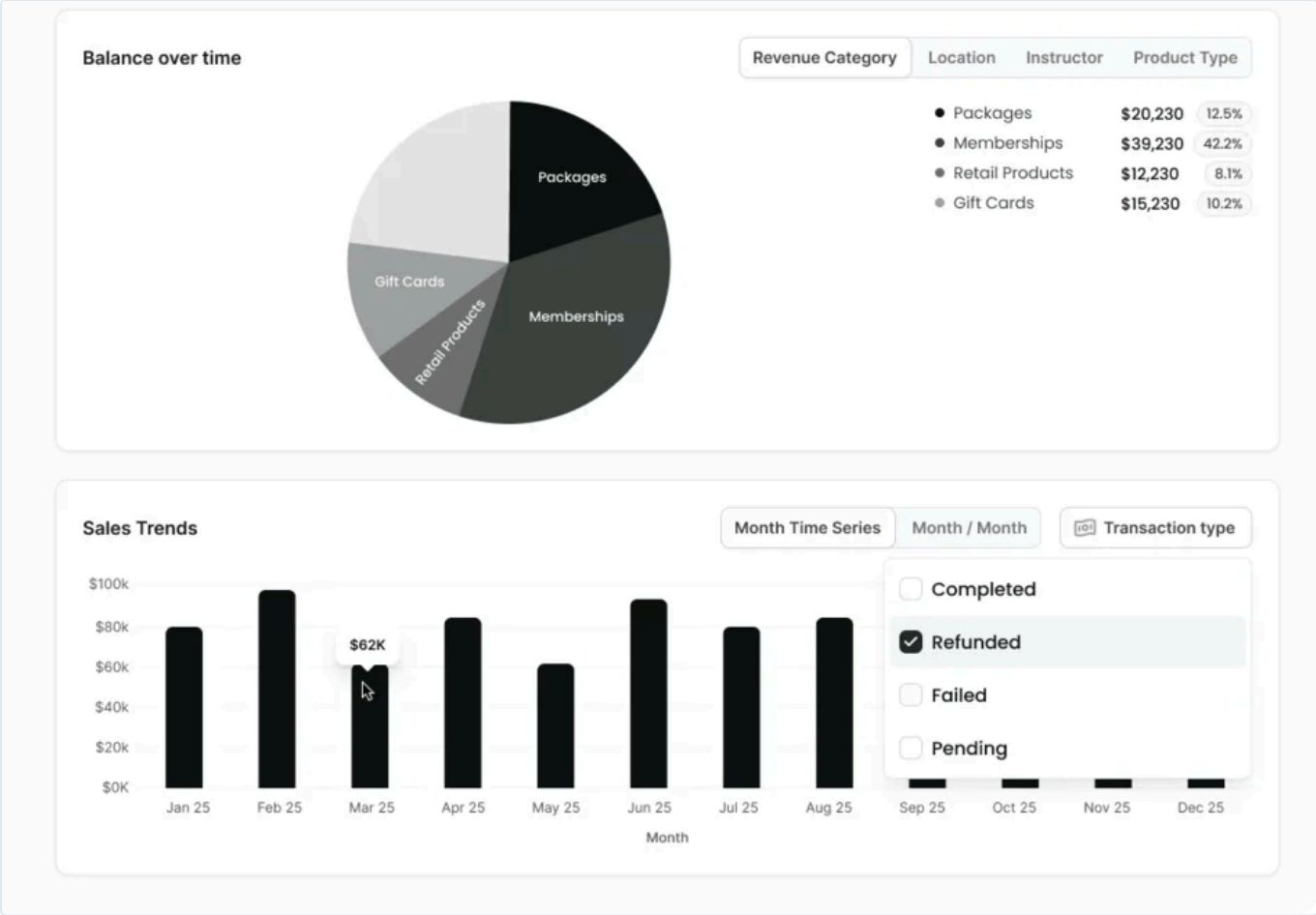
## What the reporting product actually does

V1 was one endpoint per report, returning chart and table and export together. V2 decomposes a report into the widgets it is made of, each fetched independently, and the endpoint names are the product design.

| KIND | ENDPOINTS | WHAT IT FEEDS |
|---|---|---|
| **Summary** | summary, total, total collected | The KPI figures at the top |
| **Breakdown** | by location, by instructor, by channel, distribution, leaderboards, aging | The breakdown charts |
| **Time series** | over time, month series, month-on-month | The trend lines |
| **Raw rows** | table, transactions, clients | The data behind the numbers |

Those raw-row endpoints return the same orders and transactions that the console's search box indexes — reporting aggregates them, search finds them one at a time, and both are downstream of what the payment path wrote.

The practical consequence is that a slow raw-transactions query no longer delays the KPI cards above it — and an export of a widget cannot disagree with the widget, because both are rendered from the same computed payload. The V1 exporters iterated the database independently, which is exactly how an export comes to contradict the screen it came from.

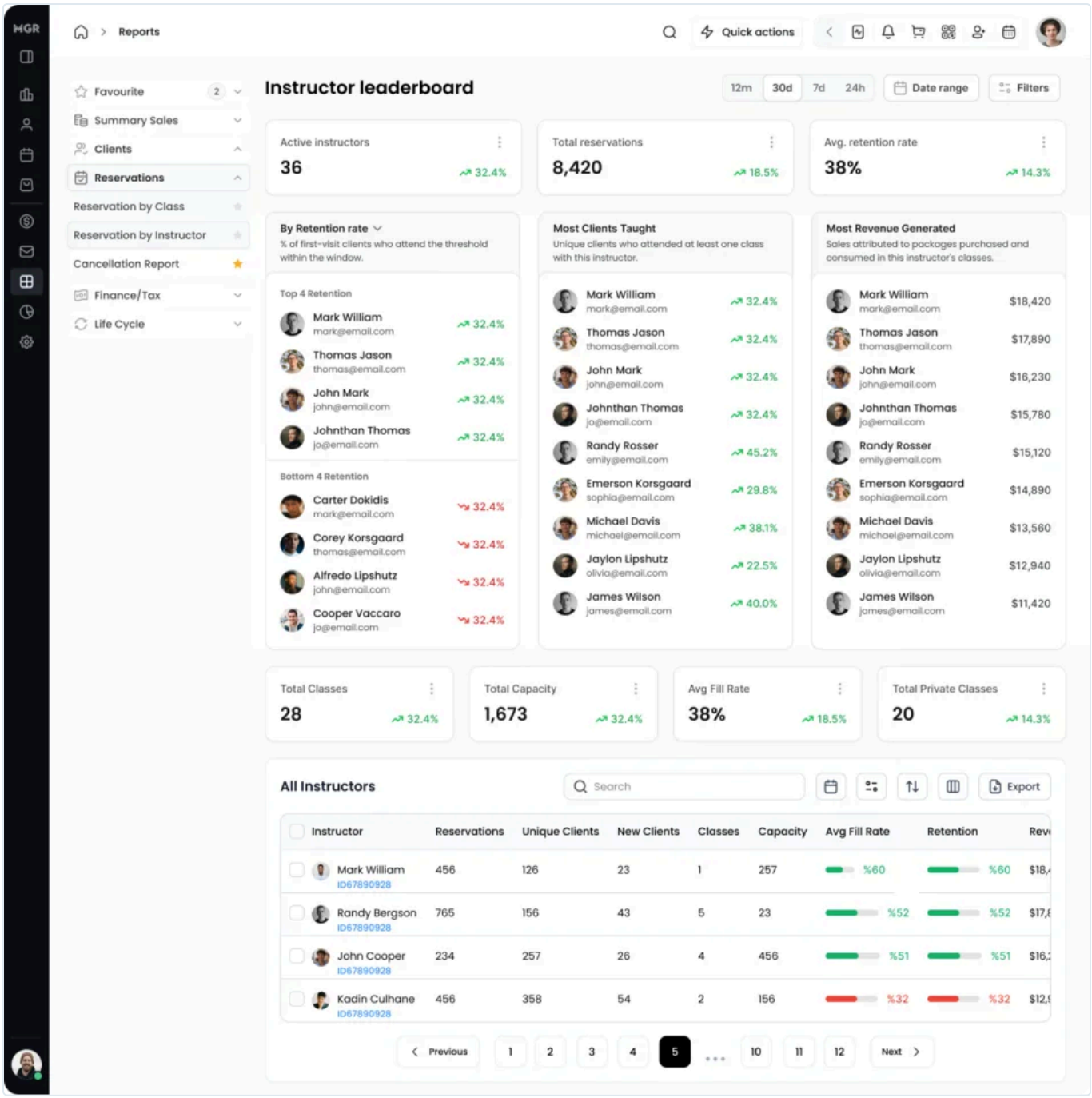



Those four tabs — category, location, instructor, product type — are the ledger's denormalised columns surfaced as a control. Switching between them is one single-table aggregate each, not four hand-written joins. The transaction-type filter below is the refund question again, answered per query rather than per report.

**Charts are bucketed in SQL, in the tenant's timezone.** The frequency presets map onto hourly, daily, weekly and monthly grouping, each passed the tenant's zone; month-on-month comparison buckets three calendar years at the database. Comparison periods are first class — previous equal-length window and year-on-year — with percentage change returning nothing rather than infinity when the prior period is zero.

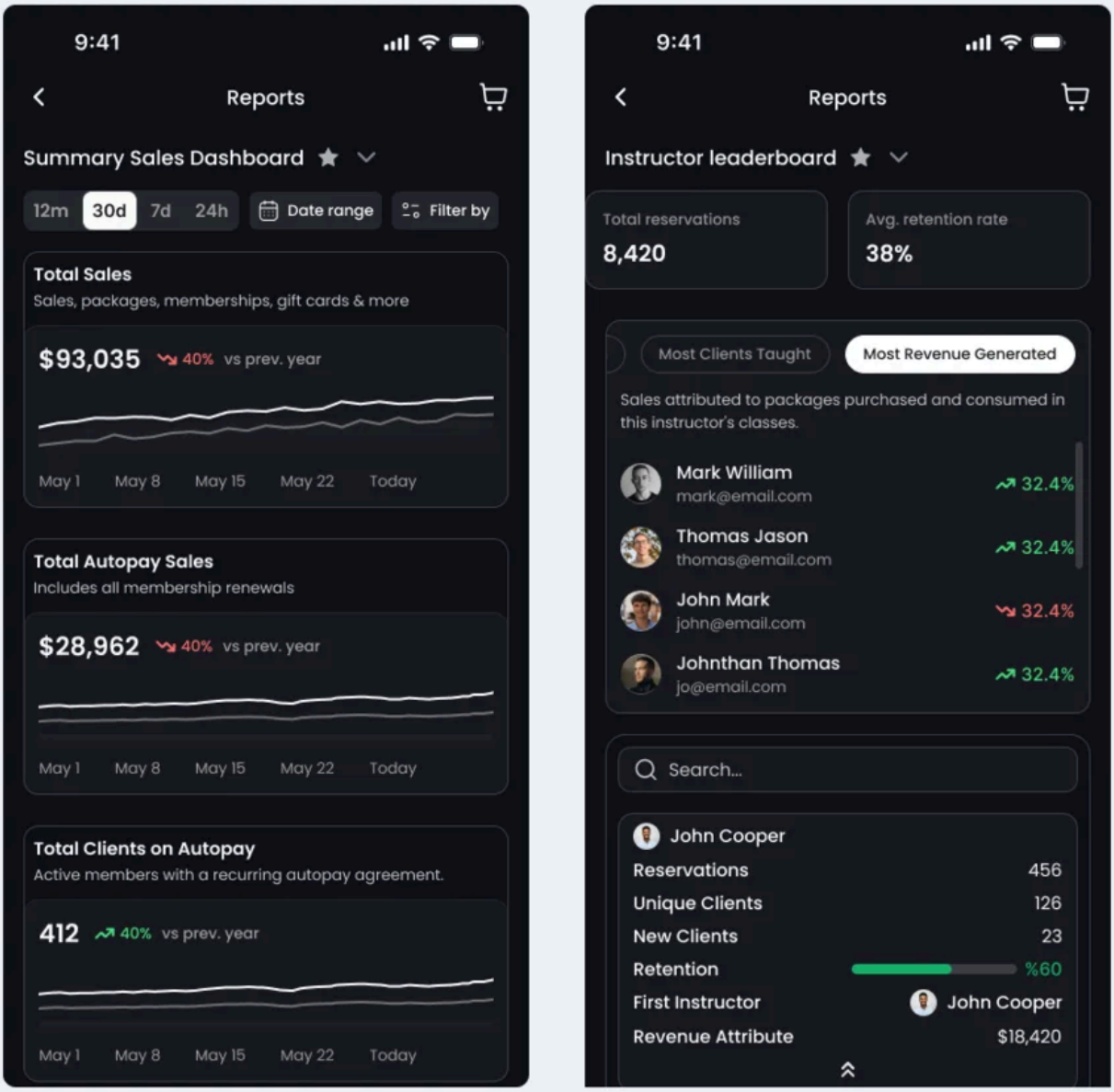
**Timezones are the quiet correctness win.** V1 parsed report dates with a bare conversion and no tenant timezone anywhere. V2 resolves the account's configured zone and threads it through every bucketing call. In a multi-tenant reporting product this is not a detail; it decides which day a 9pm sale belongs to, and therefore whether a month closes correctly.

**Pagination totals are computed across the whole result set**, not the current page — the grouped count runs first and the page is fetched separately. A small thing that is wrong in a lot of reporting code.




One screen, all four kinds of endpoint: summary tiles at the top, ranked leaderboards, breakdown tiles, then the raw rows behind them with pagination and export. Each is a separate request, so the tiles do not wait for the table.

**“Insights” means comparison mathematics and ranking**, and it is worth being precise about that. There are KPI cards with previous-period and year-on-year deltas, top and bottom leaderboards, cohort retention and no-return analysis, an end-of-month revenue pace projection, and membership conversion ratios. There is no forecasting, no anomaly detection and no model-written narrative in the reports UI. Anyone claiming otherwise about a system like this is describing a roadmap.




The same endpoints render on mobile, where a studio owner actually checks their week. Decomposing a report into independently fetched widgets is what makes this practical — a phone can load the three tiles it can show without waiting for a table it cannot.

## How fresh the numbers are, stated precisely

Reports read live, uncached queries. There are no materialised views, no nightly rollup tables, no scheduled aggregation jobs and no cache in front of the report services. Ask for this month's sales and the database aggregates this month's ledger rows, now.

One qualifier, and it should be stated rather than glossed: ledger entries are written by a background worker enqueued after the payment commits, not inside the payment's own transaction. So the ledger *converges on* the payment records rather than moving in lockstep with them. Between a sale completing and its worker finishing, that sale is not yet in a report — normally sub-second, and as long as the queue under backlog.

That was the right trade. Deriving entries inline would put proportional allocation, rounding and a validation pass inside the payment transaction, where a bug declines a customer's card instead of delaying a number on a dashboard. A daily reconciliation job on a dedicated queue re-checks for payment transactions that never produced entries, and any of the backfill tasks can fill a gap without touching the rest of the table.

So: not event sourcing, and not eventually-consistent-by-the-hour. Live queries over a table that trails payments by queue latency, with a scheduled check that the trailing ever finishes.

## Results

| METRIC | BEFORE | AFTER |
|---|---|---|
| Definitions of revenue in the codebase | 19, one per report | **1**, for the V2 financial families |
| How a breakdown by dimension is computed | A hand-written join per report | A single-table aggregate on an indexed fact table |
| Report families on the new surface | — | 9 families, 46 endpoints |
| V1 report families fully retired | — | 3, along with their migration scaffolding |
| Adding a reporting dimension retroactively | Not possible | Column, rebuild, ship — done once in production |
| Tenant timezone in report bucketing | Absent | Resolved per account and threaded through every query |
| Export agreeing with the screen | Separate query, could differ | Serialised from the same computed payload |
| Customer-visible cutover | — | None. Per-tenant flag, unchanged API contract |
| Test coverage on this subsystem | Baseline | Roughly 5× by spec line count, ~3,500 lines on the ledger |

Query latency, ledger row counts and rebuild duration on production data are the client's to publish rather than ours to estimate. What is structural, and does not need a figure attached, is that the platform now has one place where the question "what counts as revenue" is answered — and a table that can be thrown away and rebuilt when the answer needs to improve.

### What we would do next

**Finish the definition.** The V2 financial families read the ledger. The dashboard metric tiles and the older business-analytics endpoint still compute revenue their own way — one from completed payment transactions with a two-hour cache, one from order totals — and the in-product assistant uses the former.

The ledger exists precisely to be the one answer, and until those surfaces move, the platform has one canonical definition and two legacy ones still on screen. This is the highest-value remaining work and it is mostly deletion.

**Make the reconciliation job say something useful.** It runs daily on its own queue and correctly anti-joins payments against entries, but when it finds a discrepancy it emits a static log line — no count, no ids, no tenant, no metric. The good pattern already exists twenty lines away in the write-time validator, which reports the transaction id and the exact delta. The daily check should do the same, and it should compare amounts rather than only presence, because today a derivation bug that produces an entry with the wrong *value* passes reconciliation silently.

**Get the append-only trigger into CI.** Rails does not dump triggers to its Ruby schema file, and this project does not use the SQL schema format, so databases created for tests have the table without the trigger. The strongest guarantee in the design is the one the test suite cannot currently verify.

**Move exports to background generation.** They are synchronous, written to a temporary file and streamed back inside the request, with row caps standing in for streaming. A large export on a large tenant is a request holding a worker. Generating in a job and delivering a signed link is well-understood work and would remove the caps entirely.

**Retire the rest of V1, or decide not to.** Fifteen reports still run on the old architecture with naive date handling, which means they can disagree with V2 about which day a sale belongs to. Either they earn a V2 equivalent or they should be explicitly declared legacy. The current middle state is the one that generates support tickets, and it is the honest reason we would not call this migration finished.

---

## If this sounds like your system

The shape recurs in any product old enough to be useful. Reports accreted next to the features that prompted them, each with its own definition of the number everyone quotes; a dashboard and a report and an export that do not agree; and nobody willing to touch it, because the one thing worse than a slow report is a wrong one.

The transferable decisions are these. Fix the semantics before the performance, because indexes make a disputed number arrive faster. Separate when-it-happened from when-you-recorded-it, or you can never replay anything. Prohibit `UPDATE` and permit `DELETE`, so history is never edited but the projection stays disposable. Make the rebuild call the same code as the live write path, or the two will drift. And migrate the numbers rather than the endpoints — two derivations behind a per-account flag under an unchanged contract is how you re-platform anything whose output people audit.

This is the unglamorous half of platform work, and the half that decides whether the rest of a product can be trusted. If you have a number that three screens disagree about, or a reporting layer nobody dares rewrite, tell us about it — it is a problem with a known shape.

## Tech stack

BACKEND

Ruby 3.3.9 · Rails 7.1 (API) · Sidekiq

DATA

PostgreSQL 16 · Append-only fact table · Postgres trigger enforcement · Six composite indexes · BigDecimal arithmetic

MIGRATION

Flipper per-tenant flags · maintenance_tasks · Resumable batched replay

REPORTING

Groupdate · Timezone-aware SQL bucketing · caxlsx · Prawn

FRONTEND

Next.js · TypeScript · Chart.js

OBSERVABILITY

Sentry · Write-time reconciliation · Daily reconciliation job

## Frequently asked questions

### Why did three screens show three different revenue numbers?

Because each had independently made a defensible decision about four things — whether refunds are netted or excluded, whether tax counts as revenue, whether a sale belongs to the day it was captured or the day it was initiated, and how fresh the figure needs to be — and no two had made the same decisions. Nineteen report endpoints had accreted one at a time over five years, each a hand-written join over live operational tables. There was no shared definition of revenue to disagree with, because there was no shared definition at all. That is the normal end state of a reporting system that grows a report at a time, and it is a modelling problem rather than a query problem.

### Is this event sourcing?

No, and the distinction matters. The ledger is a *projection*, not a source of truth. Payment transactions and their domain records remain authoritative; the ledger is a denormalised, append-only derivation of them, optimised for aggregation and rebuildable from them at any time. There is no event stream, no replay-to-reconstruct-state, and no attempt to make the ledger the system of record. That is precisely what keeps it cheap to own.

### Why not just add indexes and materialised views to the existing tables?

Indexes would have fixed speed. Nothing about a view fixes the actual problem, which was that nineteen reports each defined revenue differently in their own hand-written joins. The ledger is a semantic fix that happens to also be faster: it decides once what a sale is, what counts as revenue, when it happened, and which dimensions describe it. Materialised views would also have reintroduced a staleness window that live queries over an indexed fact table avoid.

### Doesn’t allowing DELETE defeat the point of an immutable ledger?

This was the central design tension and it was resolved deliberately, in a migration named after the decision. Immutability’s purpose is that history is never quietly altered — that is `UPDATE`, and it is prohibited at the database, for everyone, permanently. `DELETE` serves a different purpose: it makes the projection disposable, so a derivation improvement or a new reporting dimension can be applied to all of history by replaying it. A ledger you cannot rebuild is one where your first modelling mistake is permanent. Rows are never edited; the table is regenerable. Both, not either.

### How do you know a rebuild produces the same numbers as the live path?

Because it is the same code. The replay calls the same service classes the live workers call, rather than a parallel implementation of "how do we derive entries from an order". That is the property that makes rebuild tooling still trustworthy a year later, and it is the first thing worth checking in any system that claims to have it — two implementations of one derivation will drift, and the drift will surface as a number nobody can explain.

### What happens if a ledger write fails?

The worker retries. Duplicate inserts are rejected by two unique indexes and caught explicitly, so a retry after partial success completes the remaining rows rather than duplicating the written ones. If the derived rows do not sum back to their source transaction within two cents, an engineer is notified with the transaction id and the exact delta. If the entries never appear at all, a daily reconciliation job finds the gap and a targeted backfill task fills it without touching the rest of the table.

### How fresh are the numbers?

Reports read live, uncached queries — no materialised views, no nightly rollups, no cache in front of the report services. Ask for this month's sales and the database aggregates this month's ledger rows, now. One qualifier we would rather state than gloss: entries are written by a background worker enqueued after the payment commits, not inside the payment's own transaction, so the ledger converges on the payment records rather than moving in lockstep with them. Between a sale completing and its worker finishing, that sale is not yet in a report — normally sub-second, and as long as the queue under backlog.

### Was any of this visible to customers while it was happening?

That was the design constraint. The ledger was built beside the live system without changing behaviour; the switch to ledger-backed queries happened one tenant at a time behind a feature flag, under the existing API contract, with the previous implementation retained underneath; and the old reports were deleted only after their replacements reached parity. Reporting is what people audit their business with. It is the wrong place to ask customers to absorb an architecture change they did not request.