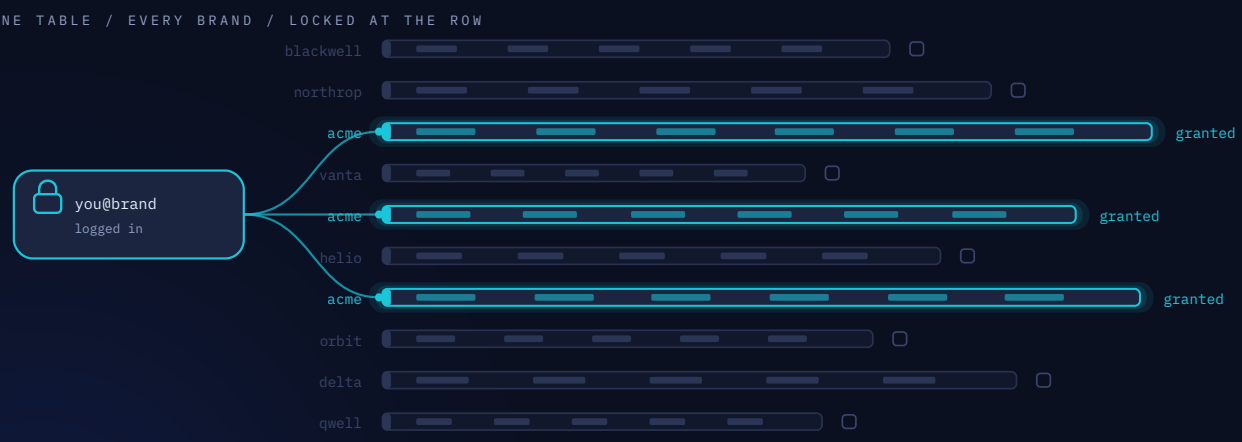
FOR PAID-MEDIA TEAMS WHO WANT TO OWN THEIR DATA

The Creative Analytics Stack

The complete architecture of a multi-tenant dashboard that turns raw Meta, Northbeam, and Triple Whale exports into creative performance insight. One Postgres database. No API server. Security enforced at the row.



WRITTEN BY

Curtis Howland

DTC paid social · Marketing Misfits

curtishowland.com

built for teaching · no client data

START HERE

One database. Every brand. Locked at the row.

This is the full architecture of an analytics application for paid-media teams. It ingests ad-level spend, revenue, and customer data from Meta, Northbeam, and Triple Whale, then runs creative analysis, fatigue detection, and launch tracking on top of it. The entire backend is a single Supabase project, so there is no separate API server to operate.

THE ONE IDEA THAT RUNS EVERYTHING

Security is not in the application code. It is in the database. The browser holds a public key and is allowed to ask for anything. Postgres decides what it actually returns, row by row, based on who is logged in.

Two facts that drive every later decision

FACT 01

All analysis is client-side

There is no server-side aggregation. The browser pulls the rows it is allowed to see and computes everything locally. That keeps the backend simple, but very large single-account datasets, hundreds of thousands of rows, can get slow. So date-range filtering matters.

FACT 02

Tenancy is row-scoped

Every brand shares the same tables. A column called `account_id` on every data table, plus Row Level Security, is the only thing that keeps one brand invisible to another. Not separate databases. Separate rows.

What is inside

THE ARCHITECTURE

01

CHAPTER 01

The shape of the system

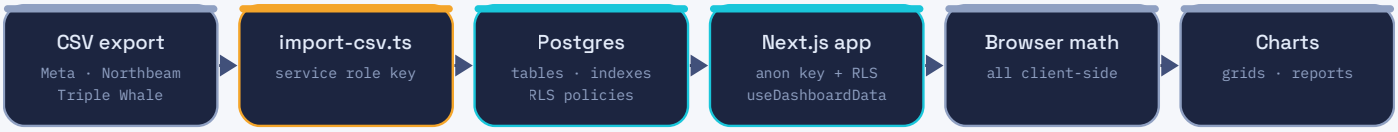
Data moves in one direction. Follow the path once, and every decision later in this guide makes sense.

CHAPTER 01 · ARCHITECTURE

From a CSV export to a chart, in one pass

Understanding this path is the fastest way to understand the whole system. A CSV is exported from an ad platform. A trusted import script writes it into Postgres. The browser reads back only the rows it is allowed to see. Then it does all the math itself and draws the result. Nothing flows the other way.

DATA FLOWS ONE DIRECTION →



no server-side aggregation · the browser only ever receives its own rows

The pipeline runs left to right. The **import script** writes with the service role key. The **browser** reads with the anon key, and Row Level Security trims the result to its own accounts before a single number is computed.

WHY THIS MATTERS

Because the backend does no aggregation, the architecture stays tiny and cheap to run. The trade-off lands entirely on the client, which is why narrowing the date range is the main lever for speed on a large brand.

What you need before you begin

YOU NEED	NOTES
Node.js 20 or newer	Runs the app and the import and admin scripts.
A Supabase project	The free tier is enough to start. Create one at supabase.com.
Three Supabase keys	Project URL, anon (publishable) key, and service role key. All three live in Project Settings, under API.
A CSV export	Optional at first. Load the demo dataset and import real data later.

TRUST MODEL

02

CHAPTER 02

Two keys, two trust levels

The same database answers to two very different callers. The whole security model starts with telling them apart.

One public key, one master key

Supabase hands you two keys, and the difference between them is the difference between safe and catastrophic. One is meant to live in the browser. The other is a skeleton key for trusted server-side work, and it should never leave your server.

PUBLIC LANE

anon key

publishable · in the browser

Safe to expose. Every read passes through Row Level Security, so it can only ever return rows the user has been granted.

`RLS decides → rows in, rows out`

TRUSTED LANE

service role key

master · server-side only

Bypasses RLS entirely. Powers imports, admin scripts, and face scans, the trusted server-side jobs that should not be gated.

`never in client code · never committed`

The **anon key** is gated by Row Level Security on every read. The **service role key** ignores RLS completely, which is exactly why it is reserved for imports and admin scripts.

TREAT THE SERVICE ROLE KEY LIKE A PRODUCTION PASSWORD

It is a master key that ignores all Row Level Security. It belongs only in server-side env files and your host's project settings. Never put it in client code, never commit it, and never paste it into a browser.

Why most writes skip RLS on purpose

Browser reads go through the anon key and are filtered by RLS. Most writes, bulk imports, face scans, admin operations, go through the service role key, which bypasses RLS entirely. That is deliberate. Bulk imports and admin tasks are trusted server-side jobs, so they should not be constrained by per-user visibility. The RLS read policies exist to gate what the browser can pull back, not to police trusted scripts.

WHERE ADMIN POWER COMES FROM

There is no admin flag in the database. Admin status is configured through an environment variable, a comma-separated allowlist of email addresses that unlocks account management and the admin tooling. Use the exact lowercase address stored in Supabase Auth.

THE SPINE

03

CHAPTER 03

Row Level Security

A logged-in user can read a row only if that row's account belongs to them. Postgres enforces it on every single query.

The two tables that control everything

Spend the most time here, because this is the spine of the whole database. Two small tables decide who can see what. Everything else is built on top of them.

TENANT LIST

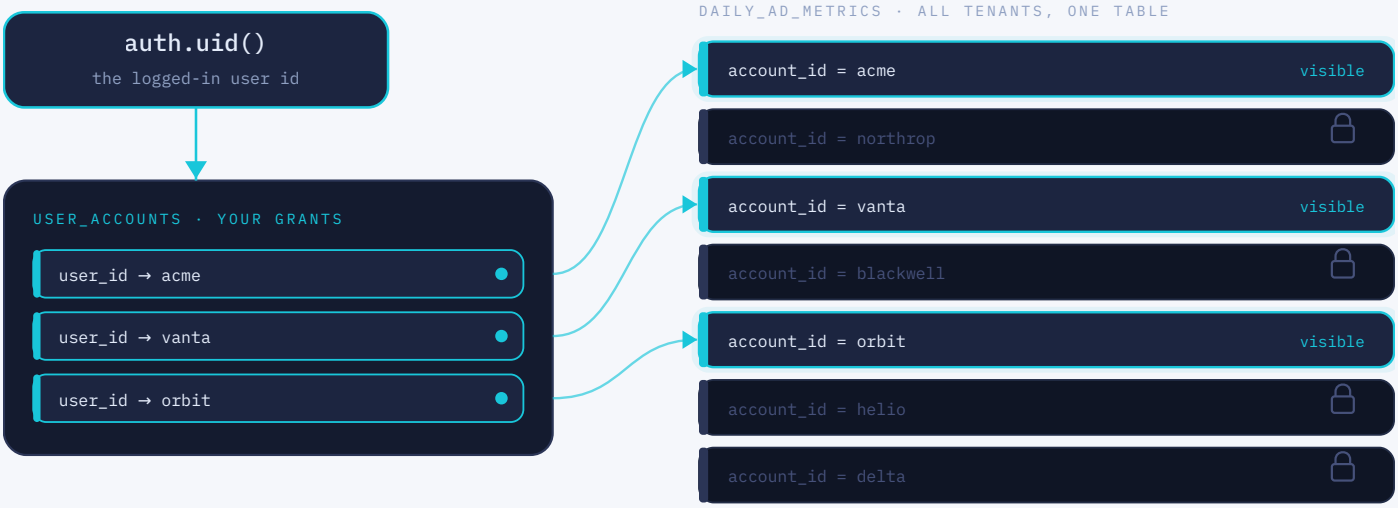
accounts

One row per brand. A short `slug` used in scripts and URLs, plus a `display_name`. This is simply the list of tenants.

ACCESS CONTROL

user_accounts

The junction table, and the heart of access control. One row means one grant: this user may see this account. A user with no rows here sees nothing at all.



A user's grants in **user_accounts** light up the matching account rows in every data table. Rows for accounts they were never granted stay locked, no matter what the browser asks for.

The policy that repeats on every table

Every data table carries an `account_id` column and the same shape of read policy. Read it once and you have read all of them.

```
ALTER TABLE daily_ad_metrics ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users see account metrics" ON daily_ad_metrics
  FOR SELECT USING (
    account_id IN (
      SELECT account_id FROM user_accounts WHERE user_id = auth.uid()
    )
  );
```

In plain language: a logged-in user may read a row only if that row's `account_id` appears in their personal list of granted accounts. `auth.uid()` is the current logged-in user id, supplied by Supabase Auth. Because Postgres enforces this on every query, the browser can ask for the entire table and still only ever receive its own accounts' rows.

Granting access is just adding rows

SETUP-ACCOUNT.TS

A new brand

Creates a brand and automatically grants every configured admin access to it. A new account starts with no tag rules.

CREATE-USER.TS

A new login

Creates one login and grants it one brand. The account has to exist first.

THE MENTAL MODEL

Everything in access control is just more rows in `user_accounts`. Grant, revoke, and share all reduce to inserting or deleting a single row that links a user to an account.

04

CHAPTER 04

The schema, one table at a time

Every data table carries an account_id and the same read policy. Learn one table, and you have learned the pattern for all of them.

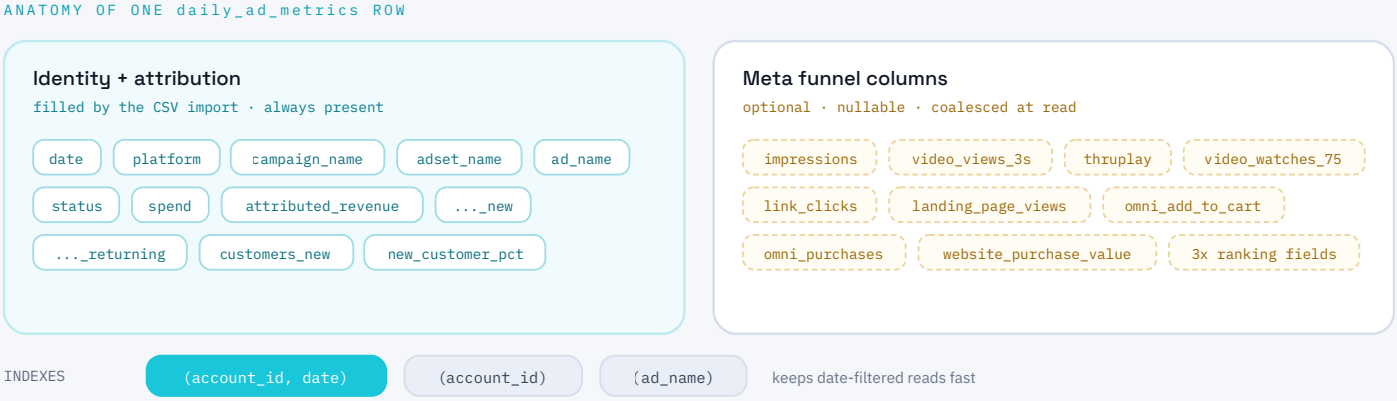
A control plane, and a data plane around it

The schema is one file, split into core tables and optional feature tables. The two control tables sit at the center. Every other table hangs off them through a shared `account_id`, which is what the read policy checks.

The **control plane** (accounts and user_accounts) decides visibility. Every table in the **data plane** carries an `account_id` so the same Row Level Security pattern applies everywhere.

The core dataset

`daily_ad_metrics` is the table every chart and computation reads: one row per ad, per day. It has two groups of columns. The first is filled by the CSV import and is always present. The second is an optional Meta funnel layer that stays null unless a Meta pipeline fills it, and the app coalesces nulls at read time, so a CSV-only setup works fine.



Solid pills are filled by every import. Dashed pills are the **nullable** Meta funnel columns. The composite **(account_id, date)** index is what keeps date-filtered reads fast on a large brand.

The rest of the core, and the optional features

Three more core tables round out the everyday surface, then a set of optional feature tables extend it. Every table below has Row Level Security enabled with the same account-scoped read policy unless noted.

TABLE	WHAT IT HOLDS	ACCESS SHAPE
tag_rules	Pattern-based creative classification, scoped per account. Turns ad names into structured tags.	Account read
monthly_budgets	Drives the production planning tab. One social budget per month.	Read and write for members
creatives	One row per asset: thumbnails, AI tags, ad copy, on-screen text.	Account read
saved_views	Named, team-shared filter presets, stored as a JSON snapshot of the whole filter state.	Team reads · owner edits
saved_reports	Saved report configs, plus global system templates visible to everyone.	System or own · owner edits
brand_content	Vectorized brand and customer text for semantic search, using pgvector.	Account read
login_events	Lightweight usage analytics: logins, tab views, account switches.	Insert own · admin reads
meta_change_log	Imported Meta change history, budget edits and status flips, for the change views.	Account read

THE PERSONA AND FACE-TAGGING TABLES

An optional subsystem tags which creator appears in each ad using AWS Rekognition, backed by seven tables (face_personas, creative_personas, persona_scan_jobs, persona_accounts, persona_merge_dismissals, creative_faces, creator_candidates). If you do not enable face tagging, they simply stay empty. They are written server-side, so their policies only gate browser reads.

ONE THING TO REMEMBER ABOUT BRAND_CONTENT

It needs the vector extension. The schema file runs `CREATE EXTENSION IF NOT EXISTS vector` before creating that table, so a fresh run handles it for you. Embeddings only require an OpenAI key if you actually use semantic search.

INGESTION

05

CHAPTER 05

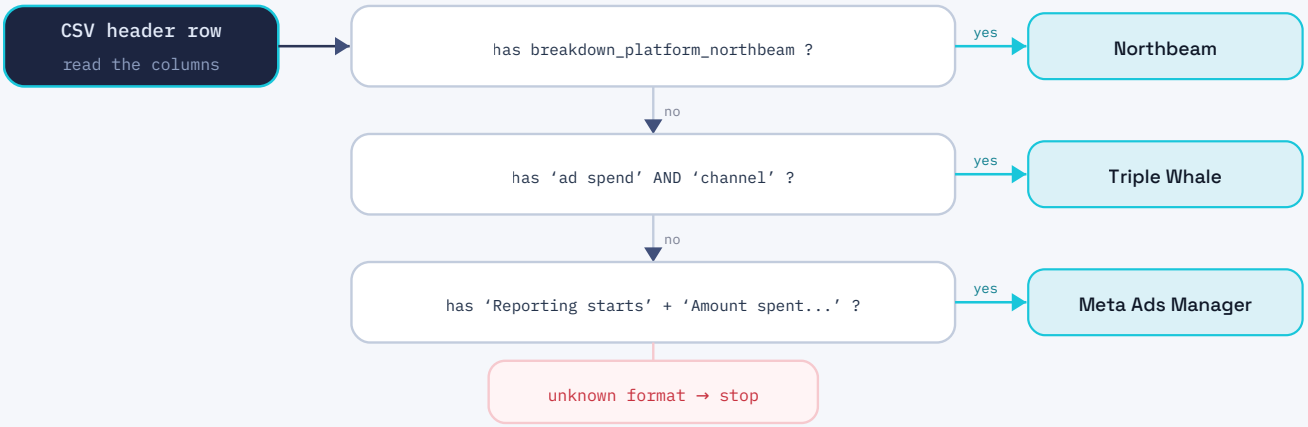
Getting data in

Drop a CSV and name the brand. The importer works out the format, and keeps every import safe to repeat.

The importer detects the format for you

Data enters `daily_ad_metrics` through one script. It reads a CSV, detects the source platform from the header row, transforms the rows into the common schema, and writes them to the account you name. You never pass a format flag.

FORMAT IS DETECTED BY HEADER SIGNATURE · NO FLAG NEEDED

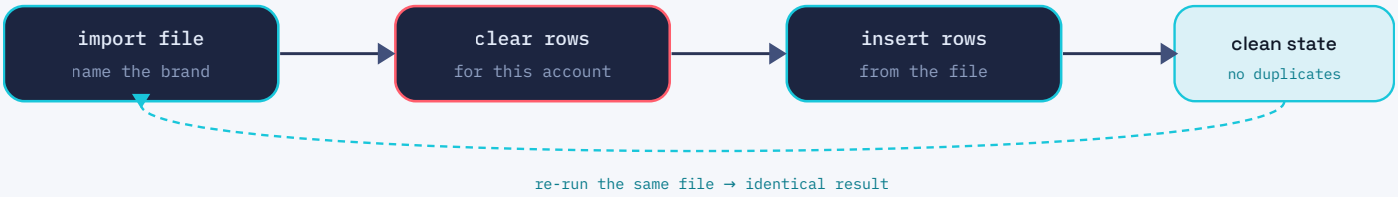


Detection is by **header signature**. The importer checks for each platform's tell-tale columns in turn. If none match, it reports an unknown format and stops rather than guessing.

Re-importing is always safe

The importer does a clear-then-insert per account: it removes the account's existing rows and reinserts from the file. Re-importing the same file, or a corrected one, will never create duplicates. The trade-off is that a partial file replaces a fuller one, so always import the complete export.

IMPORTS ARE IDEMPOTENT · CLEAR, THEN INSERT, PER ACCOUNT



Clear the account's rows, insert from the file, land in a clean state. Run the same file again and you get the **identical result**, with no duplicates.

IF YOU WANT A HARD GUARANTEE

Add a unique index on `daily_ad_metrics` across (account_id, date, platform, campaign_name, adset_name, ad_name). Only do this if your source data never legitimately repeats that combination on a single day.

Tag rules turn ad names into dimensions

Tag rules turn raw ad names into structured dimensions you can slice by, such as format, type, audience, or creator. Each rule has a pattern, a match type (contains, starts with, ends with, exact, or regex), a match field (which name to test), and the tag name and value it assigns when it matches. The app runs every account's rules against its ad names to build a tag map.

TAG RULES TURN AD NAMES INTO DIMENSIONS YOU CAN SLICE

One ad name, run through one rule, produces structured tags. A **priority** field breaks ties when several rules could match the same ad.

A USEFUL CONVENTION

To hide brand-awareness creatives from performance views, add a rule that sets `campaign_type = Brand`. The dashboard's exclude-Brand toggle turns on automatically for any account that has such a rule.

THE BUILD

06

CHAPTER 06

Stand it up: zero to running

Six steps from an empty Supabase project to a dashboard rendering your own data.

Six steps to a live dashboard

This is the whole setup. Each step builds on the one before it, in order.

1 Install dependencies

Pull everything the app and scripts need.

```
npm install
```

2 Create the schema

Run schema.sql in the Supabase SQL editor. It is idempotent, so it is safe to re-run. Add seed-demo.sql for a synthetic brand so the dashboard renders on first launch.

```
-- paste scripts/schema.sql
```

3 Configure environment

Copy the example file and set the four required variables: the Supabase URL, the anon key, the service role key, and the admin emails.

```
cp .env.example .env.local
```

4 Create your first user and account

Add a user in the Supabase dashboard with the email confirmed, then create an account and grant access.

```
npx tsx scripts/setup-account.ts acme "Acme Co"
```

5 Run the app

Start the dev server, sign in, and land on the dashboard.

```
npm run dev
```

6 Import your data

Load the env into your shell, then point the importer at your export. It auto-detects the format.

```
npx tsx scripts/import-csv.ts acme export.csv
```

The environment variables

The first four are required. The rest enable optional features and can be left unset.

VARIABLE	REQ	WHAT IT DOES
NEXT_PUBLIC_SUPABASE_URL	yes	Your Supabase project URL. Safe to expose.
NEXT_PUBLIC_SUPABASE_ANON_KEY	yes	The public anon key. The browser uses this and RLS keeps it safe.
SUPABASE_SERVICE_ROLE_KEY	yes	Master key for import and admin scripts. Bypasses RLS. Keep secret.
NEXT_PUBLIC_ADMIN_EMAILS	yes	Comma-separated emails that get admin access. Exact lowercase address.
ANTHROPIC_API_KEY	opt	Enables AI creative suggestions in the strategy grid.
OPENAI_API_KEY	opt	Enables text embeddings for brand_content semantic search.
REKOGNITION_* (5 vars)	opt	AWS keys, region, collection, and S3 bucket for face tagging.

WHY THE CUSTOM REKOGNITION_ PREFIX

The face-tagging variables use custom names instead of bare AWS_ names on purpose. Vercel and Lambda reserve the AWS_ prefix for their own platform credentials, so a custom prefix avoids a name collision in production.

The setup and admin scripts

SCRIPT	WHAT IT DOES
setup-account.ts	Creates or upserts an account, then grants every admin email access to it.
create-user.ts	Creates a pre-confirmed auth user, then links it to an existing account.
import-csv.ts	Auto-detects the CSV format and loads it into the named account.
register-persona.ts	Registers one creator face into the Rekognition collection. Optional.
scan-personas.ts	Bulk face scan for an account. Images sync, videos run as async jobs. Optional.

DEPLOYING TO PRODUCTION

It is a standard Next.js App Router project. Push it, import it into Vercel, and add the same environment variables. The database itself does not move: Vercel hosts the front end and points at the same Supabase project. Do not forget the service role key and the admin emails.

FIELD NOTES

When something looks wrong

SYMPTOM	CAUSE AND FIX
A user logs in but sees no data	They have no row in `user_accounts`. Grant access with create-user.ts, or by adding a user_accounts row for an existing user.
The dashboard feels slow on one brand	All analysis is client-side. Narrow the date range. The (account_id, date) index keeps filtered reads fast.
Re-importing seems to lose rows	The importer clears then inserts per account, so a partial file replaces a fuller one. Always import the complete export.
Scripts cannot connect	You did not load the env. Run the source command in the same shell first.
Face tagging fails in production	Confirm you used the REKOGNITION_ prefixed names, not AWS_, which the platform reserves.
A brand_content insert fails	The vector extension is missing. Re-run schema.sql, which enables it before creating that table.

Command cheat sheet

```
# app
npm install                                  # install dependencies
npm run dev                                   # local dev server (port 3000)
npm run build                                 # production build

# load env for every script, in the same shell
set -a && source .env.local && set +a

# accounts, users, data
npx tsx scripts/setup-account.ts acme "Acme Co"
npx tsx scripts/create-user.ts user@example.com TempPass123! acme
npx tsx scripts/import-csv.ts acme export.csv
```

REFERENCE

The vocabulary, in one place

`Account`
One brand, or tenant. Every data row belongs to exactly one account.

`anon key`
The public Supabase key the browser uses. Safe to expose, because RLS limits what it returns.

`auth.uid()`
The current logged-in user id, supplied by Supabase Auth and used inside RLS policies.

`Idempotent`
Safe to run more than once with the same result. True of the schema file and of per-account imports.

`Row Level Security`
Postgres rules that decide, per row, what a logged-in user may read.

`service role key`
The secret Supabase key that bypasses RLS. Server-side scripts and admin tasks only.

`Tag map`
The in-app lookup from ad name to its classified dimensions, built from tag_rules.

`Slug`
The short, URL-safe account identifier, such as acme, used in scripts and routes.

YOU HAVE THE MAP

Now build the thing that shows your work.

This is the kind of system I build and write about: data infrastructure for DTC operators who would rather own their numbers than rent a dashboard. If you run paid social and any of this resonated, here is where to go next.

THE NEWSLETTER

Read the breakdowns

Weekly, opinionated analysis for DTC operators and CMOs. Real spend, real creative, shown openly.

THE PODCAST

Marketing Misfits

Conversations on paid social, creative strategy, and building a DTC growth practice in public.

Curtis Howland

DTC paid social · creative strategy · data

curtishowland.com

find me on LinkedIn