

Long-form case study · 8 min read

Pictotube

A dark, private, cross-platform social app where the unit is the album — a whole moment of photos and videos tied to an event, a place, and the people in it.

ROLE

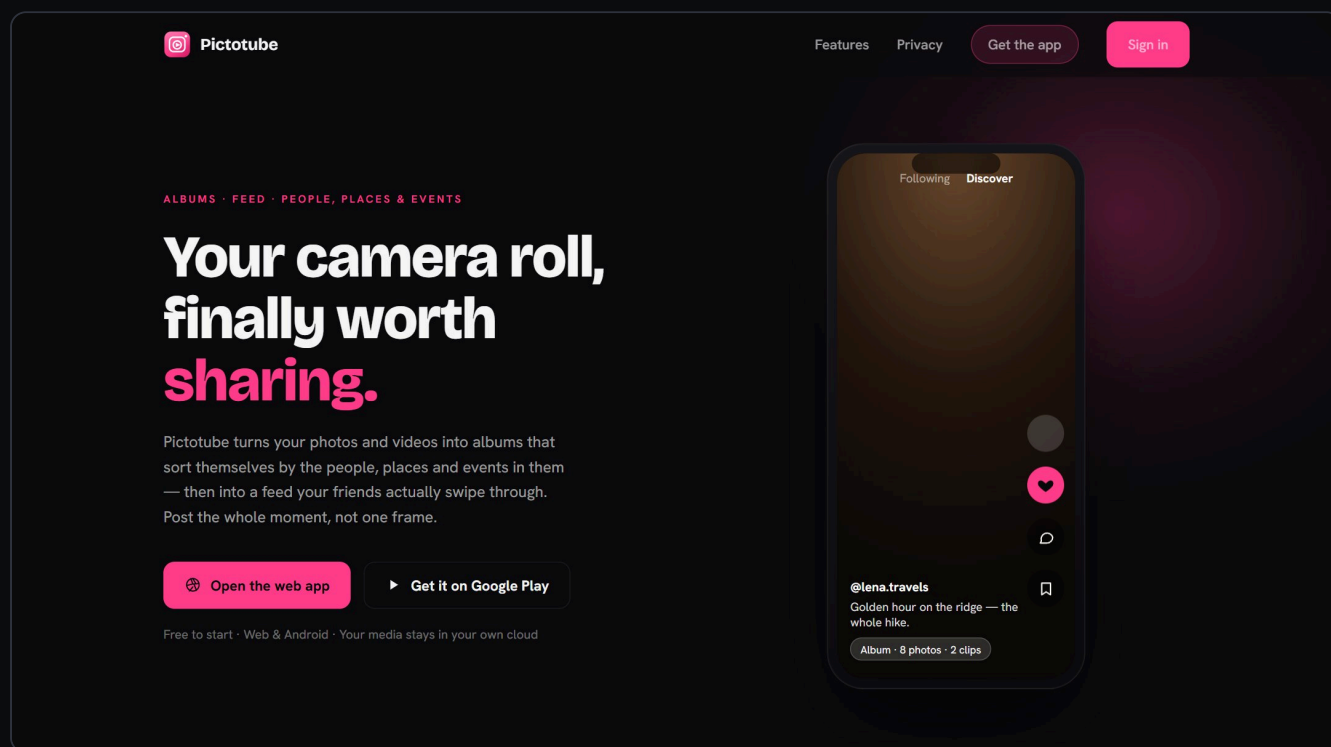
Design · architecture · build

PLATFORM

Android · Web · Cloud

SCOPE

In active development



Pictotube web — landing page

Why Pictotube exists

Photos and videos of the moments that matter end up scattered across camera rolls and feeds. Pictotube is a private, dark, cross-platform social app built around one idea: the unit is the **album** — a whole moment of photos and videos tied to an **event**, a **place** and the **people** in it.

It has two modes on purpose. You **browse** in a Pinterest-style masonry (albums, search, saved, profiles) and you **immerse** in a TikTok-style full-screen feed (home feed, media viewer, album slideshows). A tap moves you from one to the other. Everything is dark so the media is the subject; the pink accent is reserved for actions.

My role

I own the whole product: the product design, the Android app (Kotlin, Jetpack Compose, Hilt, Room, WorkManager, Media3, CameraX, on-device ML Kit), the web app (React, TypeScript, Vite), the Cloudflare Worker and the Firebase security rules. The repository holds about **284 commits** and **63 test files** so far.

Decision: a media pipeline outside Firebase

Pictotube deliberately runs **without Cloud Functions**, so it never needs Firebase's paid Blaze plan. Media lives in two Cloudflare R2 buckets (originals and thumbnails), and a small Worker is the only thing that can hand out access to them.

Clients send a Firebase ID token; the Worker verifies it itself against Google's public keys — RS256 signature, `aud`, `iss`, `exp`, `iat`, `auth_time` and a non-empty `sub` — with no Firebase SDK and no service account. It then signs short-lived URLs: **15 minutes** to upload, **1 hour** to download. Object keys are generated server-side as `<uid>/<uuid>`, so a user can only ever write under their own prefix, and the R2 credentials never leave Cloudflare.

Decision: how the home feed gets built

I wrote the feed's decision record before writing the feed, so the choice would be deliberate rather than whatever the first query made easy. Three shapes were on the table:

Option	Verdict	Why
Fan-out on read	Rejected	Firestore caps <code>in</code> at 30 values: following 200 accounts means 7 merged queries and no cursor that spans them.
Fan-out on write, in the client	Rejected	Every follower's feed would need to be client-writable — exactly the permission a spammer wants — and a killed app leaves a partial fan-out.
Fan-out on write, in the Worker	Chosen	The Worker already verifies tokens; it re-reads the post, confirms the author, and writes one entry per follower with admin authority.

Feed entries are **owner-read, no client write**, so there is no client-writable path at all. Reading the feed becomes one ordered query with a real cursor. I explicitly deferred two things that only pay off at a scale the app doesn't have yet: a hybrid path for accounts with huge follower counts, and ranking.

Measured, then fixed: feed cost and scrolling

One feed page (20 entries) costs **about 61 document reads**, dominated by two reads per post. The obvious optimisation — copying each post into the feed entry — would save around 40 reads a page. I chose **not** to do it: the copy is written by a service account and isn't covered by the post's read rule, so a post that later went followers-only, was deleted, or whose author blocked the reader would still render. Reading the real post is what makes the security rules apply to the feed.

What I did fix: a cold start that loaded the first page twice, cards that downloaded full-resolution originals instead of thumbnails, and a suggestions query that ran even when the feed wasn't empty.

Scrolling a video feed (dumpsys gfxinfo)	Janky frames	Slow draw commands
Before	61.4%	735
After: one pooled ExoPlayer moved between cards	36.5%	130

The fix was to stop building a new player — codec setup and all — every time a card became active. It isn't finished: the median frame is still 23 ms against a 16.7 ms budget while video plays, because a `TextureView` (which respects rounded corners) costs more to composite than a `SurfaceView`. That trade-off is written down, to revisit with more representative content.

Privacy by design

- Face detection runs **on the device** with ML Kit — face data never leaves the phone.
- Media access always goes through short-lived signed URLs; nothing in the buckets is public.
- The feed is built so that visibility is decided by security rules, never by the client.

Where it stands

Pictotube is in active development (v0.1.0) with dev and prod build flavors and real Firebase sign-in. After launch, the plan is **focus spaces** — Birthday, Wedding and Fashion — one app, one account and one feed engine, each space re-theming the create flow over the same album / event / people / place graph.

What I learned

- Write the decision record before the code: the feed's shape was fixed before a single screen depended on it.
- Measure before optimising, and keep the numbers — the 61-read page and the frame timings made the trade-offs concrete.
- A security property can be worth its cost: I kept the extra reads because they are what makes the rules apply.

BENOIT PRESLY NDONG OKE

✉ oke.benoit@gmail.com 🌐 okebenoithub.web.app <> github.com/OkeBenoitHub

🌐 linkedin.com/in/presly-benoit-ndong-oke-27987b117