FY 25-26

My brief work story of FY 25-26

At the start of the new financial year I reached a turning point. After months of spending my weekends experimenting with AI—building prototypes, stress-testing new tools, and becoming the de facto "AI guy" at the office—I realised it was time to go all-in. I resigned so I could focus full-time on advanced AI work and get back to hands-on coding, determined to understand how these systems run at a fundamental level and refine my **vibe coding** skills.

The quarter wasn't easy. My health took a significant hit, yet I still led several critical projects and pushed to ship them before my notice period ended. Juggling recovery, knowledge-transfer sessions, and final feature launches was exhausting, but it validated my ability to deliver under pressure.

Looking back, those three months were intense but deeply rewarding: I closed out key initiatives, ensured a clean hand-off for the team, and laid the groundwork for my next chapter in AI.

I'll start by detailing the smaller analytics tasks I wrapped up during this period, then move on to the larger, high-impact projects that truly defined this phase of my career.

Data Analysis via Redshift SQL & Mixpanel

For several quarters, I had been actively learning and applying SQL in my work—mainly through the help of GPT. This gave me the ability to run analyses independently, without relying on the data team. It sped up decision-making and gave me more ownership: instead of waiting for someone else to pull numbers, I could directly test hypotheses, validate assumptions, and back my calls with data.

Previously, our team mostly worked with Mixpanel dashboards. They were useful but had limits—especially when it came to more complex, nuanced questions. With Redshift SQL, I could dive into raw data and stitch together exactly the queries I needed. For

example, identifying a thousand new users and then isolating the hundred who were most engaged, tracing back how they were acquired, and what specific early experiences drove that engagement. Was it strong profile views in the first few days? Or something else entirely? SQL allowed me to find these answers in a structured way.

The benefit was clear when looking at match rates. With SQL, I could ask: *What's the match rate for new users from a specific region, acquired through a specific channel, within their first two or seven days? How does that compare over time?* We could then see whether more time on the app improved matches or worsened them—especially in cases where gender imbalances might affect outcomes.

Another area was analyzing user conversations. While Mixpanel showed conversion events, it didn't capture the sentiment or sequence of exchanges. Sendbird captured the sequence of exchanges but didn't capture the sentiment and it could be hard to visualize and draw down insights from it. So the best option here is to write queries that mapped back-and-forth chats—ensuring "A to B" and "B to A" counted as the same conversation—I could build clean datasets for the last 7 or 30 days. These datasets could then be run through LLMs to extract insights on conversation quality, tone, and opportunities to improve user experience.

Working this way built my muscle for handling complex queries: defining logic carefully, cross-checking outputs, and validating data against actual user behavior. I learned quickly that a SQL query can still run even if the logic is wrong—so I made it a habit to validate results by tracking specific users' activities to ensure the data matched reality.

This hands-on work became central to my role in the strategy team. Our day-to-day revolved around building strategies, suggesting new features, and improving existing ones. For that, data was the cornerstone. Redshift SQL combined with Mixpanel gave me the best of both worlds: easy visual dashboards for broad patterns, and deep raw-data analysis for the tougher questions that shaped product direction.

Realistic Image Generation with Character Consistency

In the dating space, one unavoidable reality is that most companies deploy bots—especially in the early stages when demand and supply don't balance. For example, male users often outnumber female users, and acquiring genuine female users can be both costly and limited. If the app can't provide enough engagement early on, users leave. To maintain balance, we explored creating bots—particularly female bots—that could sustain initial interest until real users filled the gap.

To make these bots convincing, we needed realistic images. One image wasn't enough because our platform required every user to upload at least three. If a bot only had one image, it would immediately look suspicious. So I started experimenting with multiple approaches: GPT-4o's image capabilities, FAL, Replicate, and even Kling (which our marketing team was already testing for video generation). While these tools produced good single images, they didn't solve the multi-image consistency problem.

That's when I dove deeper into open-source models like **FluxDev** and **FluxPro**, which excelled at realism. Flux's results were impressive, but again, generating consistent sets of images for the same persona was difficult. The breakthrough came with **seed locking**. I discovered that by fixing the seed in **Google's ImageGen (AI Studio)**, I could regenerate variations of the same character with much better consistency. This wasn't available in India at the time, so I used a VPN to access it via the US, tested different prompts, and finally achieved more controlled outputs.

This approach worked reasonably well for female personas, though male image consistency remained more challenging. Since I had multiple priorities, I trained an intern to handle much of the execution. I prepared detailed prompts—where she only needed to swap locations like "mountains", "beach", "cafe", etc —to generate believable variations of the same character.

I documented the entire method: the seed-locking trick, prompt structures, and a library of generated images. I also shared my learnings with the founders and other teams. Interestingly, they hadn't been aware of the seed technique and were surprised by how effective it was, though the manual nature of the process (and lack of APIs) limited automation.

We also explored other approaches like face-swapping to expand image sets, but those attempts often introduced artifacts and looked visibly AI-generated. Ultimately, I recognized that solving this problem fully would require more dedicated time than I could give. Instead, I handed off my research, experiments, and resources to the team members focusing on it full-time.

This project was a fascinating look at the cutting edge of **realistic, consistent image generation for AI personas**—balancing creativity, technical experimentation, and practical limitations.

FYI: This was done way before models like Flux Krea Dev, Nano Banana, Seedream, etc were released.

Push Notifications for Male Users When a New Female Joins Nearby

One of the challenges in the dating space is maintaining male engagement, especially since male users typically outnumber females. To address this, I proposed an experiment: whenever a new female user joined within a certain radius—say 200 miles of a male's signup location—we would send a push notification to relevant male users.

The logic was straightforward: if a male in California had signed up, and a new female joined within 200 miles, he'd get a notification. But we added conditions to keep the experience relevant: both users' age preferences had to align, the male account had to be active (not deleted), and other dating preference filters needed to match. The idea was to give males an incentive to check the app more often, while also increasing visibility for new female users, ensuring they received more requests and attention early on.

I drafted a full proposal outlining the conditions, potential benefits, and experiment setup. My manager liked the idea, and we aligned with the marketing team, who handled PN campaigns. The tech and product teams also agreed in principle but asked for supporting data before implementation. Specifically, they wanted to know: if we ran this experiment, how many notifications would actually be triggered daily? The concern was avoiding notification fatigue—sending too many PNs to the same user could backfire.

I designed SQL queries to model this. Each condition (location radius, age preferences, activity status, etc.) required mapping specific data points from different tables. This meant writing complex joins and carefully sequencing filters to avoid logic errors. I learned firsthand that the order of filtering matters: applying a condition globally versus at the bottom of a query can completely change outputs. To validate results, I cross-checked with sample users manually—for instance, confirming that if the system said "Rhea" should trigger a notification for "Aman," the distance, preferences, and activity status all matched correctly.

Once the logic was sound, I presented the projected numbers: how many notifications would be sent daily, how caps could be applied (we suggested a ceiling of 5–10 PNs per user across all journeys), and how this would balance engagement without spamming. From there, the tech and product teams implemented the logic into code, building it directly into the PN framework.

The results were encouraging. Users who previously weren't active began opening the app again, at least to "check casually" when notified about a new female nearby. Some engaged further by sending requests or matches, closing the loop we had originally hypothesized.

This experiment showed how data-backed logic, combined with user behavior insights, could directly improve engagement and create a healthier balance on the app.

Personality Trait Model

As FY 24–25 began, some of the foundational research I had done earlier started turning into live features on the app. One of the most visible examples was the integration of MBTI-based personality matching.

We introduced a system where every user received their MBTI type and, based on it, the app highlighted strong matches, neutral matches, and weaker matches. This logic was directly inspired by my earlier research into the **Personality Trait Models like 16 Personality Test, Marriage Pact, MBTI, Boo's Personality Test Algorithm, etc** where I explored how personality traits could be mapped into compatibility patterns. That framework gave us the structure to translate MBTI into meaningful relationship pairings.

Although I wasn't hands-on coding this feature, it was gratifying to see my research guiding product direction. The positioning of our app shifted: instead of being "just another dating app," we could now present ourselves as a platform that leveraged personality science for deeper, more thoughtful matching.

The implementation required blending MBTI pair logic into the broader **matchmaking score formula**. While distance and preferences still mattered, the data team expanded the backend to incorporate behavioral patterns in how users answered MBTI questions. My earlier experiments—where I tested different formulas, hypotheses, and components for scoring—fed directly into this work.

By the time I resigned, MBTI-based polls were live. Users had MBTI tags on their profiles, and matches reflected compatibility beyond surface-level filters. This improved both transparency and trust: users could see there was a structured, personality-driven logic behind why they were matched, not just simple filters like distance or age.

For me, this was a milestone—watching long-term research evolve into a tangible feature that strengthened our product positioning and enhanced the user experience.

Red Flags You Both Are Okay With: Shortvibes for MBTI Polls

Once we decided to roll out MBTI-based polls and assign personality tags to users, the next challenge was how to generate **short vibes** (the one-liner connections shown between users) for these polls. Using our usual formats—first date ideas, media recommendations, or debate topics—posed risks. Since every new user would answer the same fixed set of MBTI questions, most people would converge on the top two options in each poll. We had already seen this pattern in earlier fixed and promoted poll campaigns: typically, one or two options received 80–90% of the votes, while the remaining ones barely reached 5–10%.

If we continued generating short vibes solely from those common options, the experience would quickly become repetitive and stale. To avoid that, I explored alternatives. Options included

showcasing **green flags**, **red flags**, **etc** —but the one that struck the best balance was **"Red flags you both are okay with."**

This worked for two reasons. First, it gave users fresh, engaging one-liners tied to their MBTI answers without overwhelming them with an entirely new experience. Second, it preserved continuity for older users who were already familiar with the short vibe format. By prioritizing red flags while still keeping a mix of other formats, we could reduce repetition while maintaining variety.

To test this, I set up an **automation system** to generate red-flag short vibes. Instead of waiting for the tech team (who had limited resources at the time), I replicated the formula we had used earlier for date ideas and debate topics. I wrote an **App Script** that pulled MBTI poll questions and options from Google Sheets, generated the one-liners row by row, and wrote them back into the sheet. Since App Script wasn't easy for everyone to handle, I also built a version on **Make.com**, which was more accessible for the team.

The process worked like this: each question and its four options were entered into Google Sheets, with each option listed as a separate row. The system prompt then generated the red-flag short vibe for that row. The outputs were stored directly in the sheet, creating a lightweight but reliable workflow. This mirrored the automation we had already used for earlier short vibe categories, making adoption smooth.

After validating the outputs and getting team buy-in, we handed the system to the tech team for final implementation. They didn't need to reinvent the generation logic—only to handle details like randomizing which of the four options appeared and prioritizing red flags within the MBTI polls.

The experiment was a success. By introducing "Red flags you both are okay with," we reduced the repetition issue, kept short vibes interesting, and gave users something more relatable and personal. Click-through rates improved, and users engaged more with their matches compared to previous versions of short vibes.

This feature started as a stopgap solution to prevent repetition but ended up becoming a strong addition to the app's MBTI experience.

ShortVibes based on First Date Ideas, Debate Topics & Media Recommendations

In FY 25-26, I followed through on an initiative to replace our astrology-based shortvibes with something more personal and less repetitive: **first date ideas, debate topics, and media recommendations** (YouTube, Netflix, Prime, Spotify).

The astrology system relied on sun signs and a small repository of prewritten one-liners (e.g., "Taurus × Taurus"), which led to obvious repetition and generic copy. I'd already written the PRD and requirements; this year was about shipping the experiment safely and proving impact.

Because rollback needed to be painless, we **didn't overwrite** the legacy content or events. Instead, I worked with QA and engineering to run a **closed, A/B experiment**. In Mixpanel, rather than creating a new event schema, I **reused the existing astrology shortvibe events** and added a **tag/field** to distinguish variants (legacy astrology vs. new shortvibes). This kept tech effort low and made dashboards immediately comparable.

Execution surfaced some early issues. For example, the new shortvibes displayed correctly in the US but not in India; even though India wasn't our primary market, I pushed for **global parity**—if the app is live, the experience should be consistent everywhere. We also identified mismatches where the wrong **shortvibe type** was being served (e.g., a media recommendation showing up where a first date idea should have been). I flagged and resolved these before rollout.

To generate content at scale without blocking on engineering, I built **lightweight automations:**

- **Google Sheets + App Script** to ingest poll questions/options and generate one-liners row-by-row.
- A parallel flow in **Make.com** so non-technical teammates could operate it.

Once the system was stable, we rolled out the experiment and closely monitored Mixpanel to track adoption, spot anomalies early, and be prepared to act quickly if anything went wrong. The results were strong:

- **View → Wave** (seeing a shortvibe and then sending a wave) **increased ~50%** vs. astrology shortvibes.
- Users sent **~45% more waves overall**.
- Within the experiment cohort, the **new shortvibe types drove ~60% of waves**, while the legacy "common-options" shortvibes' share dropped from ~50% to ~40%.

Beyond the numbers, we saw **qualitative impact**: in early match conversations, users began directly referencing our shortvibes (e.g., continuing a "coffee date" suggestion or chatting about a recommended show). This proved the new shortvibes weren't just increasing clicks—they were actively helping users break the ice and start more natural conversations.

With evidence in hand, we pushed toward full rollout. One remaining question was **debate topics**. I was cautious: generic debates risk irrelevance, and "debate energy" depends heavily on personal interests. Still, to test the boundaries I:

1. Prototyped a **common-option → debate topic** generator (App Script / Make.com + GPT).
2. Produced ~100 topics across ~25–30 questions.
3. Reviewed outputs with the team.

As expected, the **common-option approach wasn't good enough**—too many mismatches. We pivoted to a **curated repository** of broadly appealing, US-relevant debate prompts, tuned to be specific enough to be engaging but general enough to avoid dead ends. In parallel, I refreshed the **media catalogs** (top creators on YouTube, trending movies & shows on Netflix/Prime, current top Spotify artists), then regenerated recommendations via our system prompts. First date ideas got a larger, more diverse set as well.

After QA, we handed the full package to product/engineering: templates, prompts, repositories, and the operating plan to **replace astrology shortvibes** with this new IP. I also shared a

scaling roadmap—how to personalize further by folding in data-science signals once the base experience stabilized.

End-to-end, this work moved us from generic, repetitive astrology snippets to **actionable**, **conversation-starting shortvibes** that users actually used—measurably improving engagement while keeping implementation lightweight for the team.

Voice-Based UXR Agent

By this point, I had built a strong foundation in creating voice-based agents. I had already experimented with onboarding flows and matchmaking assistants, so I understood the nuances of designing conversational systems. The next step was applying this knowledge to help the **UXR (User Experience Research) team**.

The challenge was clear: our UXR team had just one researcher based in India, while most of our users were in the US. Time zone differences and accent gaps limited the number and quality of interviews. Even after attempts to hire in the US, costs were too high and didn't scale. At best, we could conduct two user calls a day, but scaling to 50–100 calls daily for richer insights was impossible with manual work. Each call wasn't just the conversation—it required preparation, follow-ups, transcripts, summaries, and insight documentation.

So, I designed a **voice-based UXR agent** to automate the first layer: conducting the calls, asking pre-defined research questions, handling follow-ups, and summarizing the insights. To build it, I first worked closely with the UXR researcher to understand her scripts—what she asked, how she followed up, and the structure of conversations.

I then wrote a **system prompt** to define the agent's personality and guardrails. It was instructed to gracefully handle unknowns (e.g., "I'll get back to you" or escalating to a human researcher) rather than hallucinate. I also fed it reference material—PRDs, feature descriptions, and past user transcripts—so it could respond knowledgeably about product issues. To keep conversations natural, I designed for small human-like touches: pauses, slower pacing, acknowledging interruptions, and filler words. Since ElevenLabs' most humanized voices weren't out yet (with breathing and laughter

support), I had to mimic these behaviors manually through careful scripting.

On top of that, the agent provided **sentiment tagging at the end of each call**. If the interaction was positive, it could even nudge the user to leave a Play Store/App Store rating—turning research into an opportunity for organic growth.

I iterated multiple times, testing myself and with internal teammates. Feedback guided refinements until the agent felt natural and usable. Once stable, I documented the system prompts, logic, and flows, then handed it off for integration. Behind the scenes ElevenLabs and Twilio were used and the workflow would trigger when users could either schedule a slot or select to speak on the spot to the UXR Agent within the app.

Initial campaigns were intentionally narrow—we didn't want scale yet, we wanted **signal quality**. The first ~10 calls showed promising results: transcripts and summaries flowed into the UXR team, sentiment analysis was accurate, and the researcher could still review recordings to validate findings. The bottleneck of one human researcher doing all the calls was broken.

This agent solved three critical problems at once:

1. **Scalability** – made it realistic to target dozens of calls a day.
2. **Cost** – avoided hiring expensive US-based researchers for basic interviews.
3. **Quality** – supported multiple accents, provided consistent questioning, and freed humans to focus on deeper analysis rather than logistics.

For me, this project was the culmination of months of agent experimentation—turning scattered learnings about prompts, accents, UX, and automation into a **production-ready research tool** that directly improved our user understanding.

Hunch IP: Onboarding to Date → Web Onboarding and Matching on App

This initiative extended from work we had started in FY 24–25, where I first experimented with **voice-based onboarding flows**. At that time, we wanted to test whether voice agents could handle the entire onboarding journey: asking MBTI questions, collecting user details, and completing verification. While the voice prototype worked, we quickly discovered a scale problem: what took **5–10 minutes in text onboarding** stretched to **30 minutes in voice**, even after optimizations. Breaking it into smaller calls risked heavy drop-offs, so we reconsidered the approach.

The breakthrough was to shift from voice onboarding to a **frictionless web onboarding flow**. Users didn't need to download the app or sign up first. Instead, they could answer MBTI questions and basic details directly on a lightweight webpage. At the end, they received:

- Their MBTI type and insights.
- A preview of potential matches (hooks).
- A prompt to unlock full access by purchasing a plan.

Only after payment were they asked to download the app. At that point, their entire profile—answers, details, MBTI tags—was already pre-filled from the backend. This flipped the model: instead of nudging users multiple times after app install to go premium, we collected revenue upfront during onboarding.

My role as Associate PM was to support the Product Manager leading this IP. I handled **planning**, **testing**, **and content creation**:

- **Profile Curation:** I built a pipeline for images shown on the website. Using ImageGen for AI-generated photos and real user data, I applied our **attractiveness score** logic (clarity of face, good lighting/background, no low-quality shots). I combined this with activity filters (active in last 30 days) and match rate signals. Then, I manually reviewed top profiles to validate system accuracy and ensure diversity across four age brackets (18–24, 24–32, 32–42, 42+).

- **Enrichment:** For AI-generated profiles, I automated name and age assignment using AI inference. I also generated testimonials by transcribing **UGC ads from creators**, feeding them into GPT, and converting them into written endorsements that matched the tone of the videos. These were paired with embedded videos to create credible testimonial sections.

- **Implementation Support:** I collaborated with developers to embed testimonials correctly (ensuring the right video mapped to the right text) and to resolve issues with missing or slow-loading images. To fix oversized images, I set up a **local lossless compression workflow** (via GPT-assisted scripts) that reduced all files below 1 MB without losing quality, then re-uploaded them to S3 with consistent naming.

- **Events & QA:** I did extensive event testing. Bugs included MBTI logic not firing correctly, button malfunctions, and missing or incomplete data in triggered events. After every small fix, I re-ran the **full user journey** (onboarding → payment → profile creation) to confirm stability. This required creating dozens of new test accounts with fresh emails.

The feature went live and immediately created a **new revenue stream**. Out of ~$5–6k monthly revenue, ~$1–1.5k came directly from the web onboarding flow. While modest at launch, it was significant proof of a scalable model: conversion rates were higher because users experienced value first (MBTI tag + preview of matches), then were asked to pay.

This was **Version 1**, but it established a strong foundation:

- Reduced onboarding friction.

- Created a direct revenue pathway tied to first-time onboarding.

- Validated that blending **AI-generated + real user content** could power a convincing experience.

By the time I left, both I and my manager had moved on, so I'm unsure how it evolved further. But even at this stage, it had the potential to become a **major revenue driver** for Hunch by shifting the economics of premium subscriptions from reactive nudges inside the app to proactive purchase during onboarding.

Finetuning the Chat Model

This was one of my largest and most important projects at Hunch. The goal: keep users engaged longer by making our in-app conversations feel human. We already had a live chat-bot layer built on a vanilla GPT model, but it suffered obvious flaws:

- It frequently forgot context.
- Replies were over-optimized and robotic.
- In a dating setting, users quickly realized they were talking to AI.

My job was to solve those issues and propose a better version through data-driven fine-tuning.

I began by pulling real user chats. A co-worker's old SQL script supposedly did this, but it captured only one-sided messages (my "hi" was saved, your "hello" wasn't). I rewrote the query so each row contained the full two-way exchange, limited the sample to the past three months, and included only threads with **at least five messages**.

To avoid mid-thread snippets, I required that **at least one account was created within that three-month window**, ensuring the chat started—and often ended—inside our slice. After several GPT-assisted iterations I finally had complete two-way data.

Before anything else, I scrubbed the logs. Real names became *User 1* and *User 2*; phone numbers, handles, and other identifiers were replaced with dummies unless I explicitly needed them later. I did the first pass by hand, then wrote an **Apps Script to validate** & catch anything I missed.

Next I built a **rating pass** using GPT as a judge (with a long system prompt) to score each conversation on:

- **Depth & progression** (do they get beyond small talk; do they angle toward a date?)
- **Flirtation & emotional tone** (playful, supportive, teasing—without being creepy)
- **Naturalness** (slang, shortforms like "u / ur / ttyl", emojis)
- **Healthy back-and-forth** (no walls of text; real turn-taking)
- **Signals of action** (exchanging details; discussing plans; moving off-app—though our bots would later learn to deflect that)

Also told GPT to explain its score and highlight which conversations should be included in our training.

Early runs penalized consensual "explicit" chats too harshly, so I revised the rubric: **adult content is fine if the user initiates it**. That kept things honest to how people really flirt and talk on dating apps.

After several cycles I hand-audited the top-rated threads, merged any GPT oversights, and ended with ~200 high-quality two-way conversations (≈400 messages) for the first model. Further I created ~25 high-quality two-way synthetic conversations (≈50 messages) only where we lacked coverage, making the entire total to ~225 high-quality two-way conversations(≈450 messages).

With that baseline in place, I went for data breadth rather than jumping straight into a model run. We'd ship a **general** bot first (nothing specific like separate for male and female users), but I wanted to test a hypothesis: would gendered assistants (one that "speaks like a woman," one that "speaks like a man") add any realism? So alongside the general dataset, I prepared two persona sets where the **assistant persona** was explicitly male or female. That meant re-threading conversations so the assistant's lines aligned with the intended persona, swapping user/assistant roles where needed to preserve a natural user→assistant turn order, and dropping threads that didn't make sense for that persona. I did this before our first training so we wouldn't have to re-harvest later if the idea proved useful.

Next came formatting. I started by writing an AppScript that turned the cleaned sheets into trainable JSON with strict `role/content` pairs in the order **user → assistant**. I kicked off a first run in the **OpenAI Playground** (kept parameters basically default to save time—no fancy tuning, temperature left at default). That training failed. My exporter had let a few malformed rows slip through; I had checked manually, but a couple of "user→user" or "assistant→assistant" pairs were hiding in the long tail. Only then did I write an AppScript to scan every item and flag lines that broke the structure. With those fixed, the training completed cleanly in the Playground.

I fine-tuned the **general** model first; the male/female variants came later. Within 30 minutes each, I had three custom checkpoints ready to test.

With a working checkpoint in hand, I turned to the prompt that would steer it. I wrote a **long system prompt** that encoded the rules of dating conversation:

- Be flirty when appropriate; small grammar imperfections are okay.
- Never admit you're a bot; stay in character.
- Allowed mild typos and usage of emojis and slangs so replies felt human.
- Respect boundaries: the bot does not initiate explicit topics; if the user does it participates in it.

Now that everything was in place I created a QA checklist so I don't miss at least the important cases to look for and share with my co-workers who were also testing this.

Below are few things we found not working:

- **Flirting skills** & Response to Adult content- The bot's responses weren't as playful as needed. It sometimes

refrained from participating in adult talks.

- **Emoji / slang usage** – It recognized emojis but used them sparingly. Slang was almost absent.
- **Language handling** – It responded to Hindi, Gujarati, even Chinese, when in reality no human user on a dating app would speak 50 languages fluently. This broke immersion.
- **Time awareness** – It had no sense of whether it was morning or night, so replies sometimes clashed with reality.

I logged all of this so that when we do the next training all of these issues are resolved.

The male assistant and female assistant variations performed no better than the general one. In hindsight that made sense: their data wasn't truly distinct—we'd remapped from the same pool. So parked the idea for a future revisit and doubled down on the general model.

For the second pass, I made a structural change: **every training message bundle started with a system role**. That meant our dataset had to become `system → user → assistant` for every training example, not just a single global system instruction. The tech lead also pushed for a **shorter system prompt** (lower tokens = lower cost), and suggested we embed **identity** per conversation (names/ages/locations) in the system message so the bot wouldn't forget who it was and whom it is speaking to. Doing this properly meant reworking the whole data path.

I wrote a new SQL query to pull the fields we needed to parameterize each system prompt: **(initiator/respondent) names, ages, and locations**. Because our earlier filters had already narrowed conversations, I mapped on **conversation_id** and then inferred who sent the first message to determine "user" vs "assistant" in our fine-tuning pairs. Where accounts had been deleted or fields were missing, I generated **consistent,**

fictional replacements for **age, location, and name.**

As raw names should never be used directly. I **shuffled male names only among males and female names only among females** to preserve gendered cues without leaking identities. If a chat mentioned "I live in X" or stated an age, I verified that the **system prompt variables matched the message content.** When they didn't, I updated the synthetic persona values or scrubbed the message. Handles (Instagram/Snapchat) and phone numbers were replaced with **dummy tokens.** This was manual first (to avoid misses).

I converted our long instructions into a **compact system paragraph** that carried all the rules we needed: flirty/casual tone, emojis/slang allowed, **English-only** (politely ask to switch to English if needed), never admit being a bot, okay to continue adult topics **only if the user initiates**, and keep things realistic (not overly perfect). On top of those rules, **each system message injected the persona variables** (Name, Age, City) so the assistant wouldn't drift when history got truncated during deployment.

From the first pass we knew flirting, slang, and emoji rhythm needed more coverage. I added **high-quality real snippets** where available and supplemented with **synthetic but human-sounding conversations** that demonstrated natural flirting, contemporary shorthand ("wyd", "fr", "😭"), and subtle grammar quirks. Add a few Adult conversations too so that it can now respond and co-operate whenever the user initiates it.

I rewrote my Apps Script that generates the training JSON:

- Prepend **role: "system"** with the templated prompt for **every** example.
- Then **user** and **assistant** turns.

- Ensure strict alternation (system → user → assistant) and maintain message pairing.

I also rewrote AppScript which checked user → assistant pairs. I extended it to verify **each example begins with a system role** and that **every trio is complete**. On failure it returned **line-accurate errors** so I could fix the exact rows instead of scanning the whole file by eye.

With this rebuilt pipeline, I exported the new training set and ran the fine-tuning again on Playground with **defaults** for training parameters. The Playground test runs now reflected the per-message system instruction, so the model's tone and boundaries were much more consistent across examples.

Along the way to understand cost vs. quality, I trained the one with a long **system** prompt variant too but this time with **the persona variables** (Name, Age, City) **injected on the top** so the assistant wouldn't drift when history got truncated during deployment. In the Playground, both short and long system prompts behaved properly because the full context was present. The **long** variant was naturally richer—more nuance, better humor. The **short** variant held its own: flirting improved, emojis and slang showed up, co-operated in adult talks and it reliably redirected other languages back to English. So far, so good.

I invited other co-workers, my manager and folks from the tech and product team to test it out and they were quite satisfied with this one. So we went ahead and got it integrated on the app in the dev environment to further test if things are working properly there or not.

When we integrated the fine-tuned bot into the app, we encountered several challenges. Some of these were expected and planned for, while others surfaced only after live testing. Below is a breakdown of the key issues:

1. Lack of Time Awareness

The bot was not aware of the current time context (morning, afternoon, evening). As a result:

- It sometimes generated replies that didn't align with reality, e.g., *"Look at the sunset, how beautiful it is"* in the early morning.

- In other cases, it would say *"I'm going to sleep. Good night"* even when the user was chatting in the afternoon.

This mismatch reduced the naturalness and believability of conversations.

2. Unrealistic Response Speed

The bot responded almost instantly, which felt unnatural compared to human behavior.

- Human conversations usually have natural pauses or delays.

- Without these delays, the interaction clearly felt like "chatting with a bot," breaking immersion.

3. Risk of Over-Engagement with Bots

Because the conversations were engaging and responsive, there is a risk that:

- Users might prefer spending time chatting with bots instead of engaging with real users.

- This goes against our core objective: the bot should enhance the environment, not replace genuine user-to-user interaction.

The bot should act as an engagement booster, especially for onboarding and habit-formation, but not dominate the user's time.

4. Limited Awareness of Current Events

The bot lacked awareness of recent or real-world events. For example, it could not reference:

- What happened today or yesterday.
- What's happening tomorrow.

While this could make conversations feel outdated or generic, we decided **not to prioritize solving this in version 1**, since enabling live web search would significantly increase costs. For now, this limitation is acceptable, and we can revisit it in later versions if needed.

5. Forgetting Conversation Context

On the OpenAI Playground the entire conversation history is automatically preserved in one chat session, but as we set up the finetuned model on our app it required us to **supply conversation history** for the bot to maintain context. This created several challenges:

- For example, if a user mentioned their occupation as *"CA in XYZ company"* ten messages earlier, the bot might later contradict itself and say it works as a *waiter* or *gardener*.
- While **basic details** like name, age, and location could be set in the system prompt, it wasn't feasible to hardcode every possible fact since conversations can go in endless directions.

6. Handling Threaded Replies

Another challenge arose when users replied to older messages within a thread rather than continuing linearly in the chat. For example:

- The user initially asked: *"What do you think about Trump as a president?"*

- The conversation then shifted in another direction.

- Later, the user returned to that earlier message with a reply like: *"What about this one now?"*

If the bot only received the latest message ("What about this one now?") without the original reference, it would generate a vague or irrelevant response. The reply needed to be grounded in the **original base message** to maintain coherence.

After discussions around possible solutions below are few solutions that we implemented to solve the above issues:

- **Current time injection**: We attached the server's current timestamp to the system context so the assistant could say "good morning/evening" appropriately and avoid time-clashing phrases.

- **Reply delays**: We set a **random 1–10 minute delay** before sending a response, to avoid "insta-reply" bot vibes. Later we planned a smart tweak: if the user went **offline**, we could **trigger the reply earlier** to nudge them back with a notification.

- **Ghosting cap**: To avoid users endlessly chatting with bots, we capped bot replies at **15–20** in v1. After that, the bot would stop responding. This ensured people still had ample human matches and nudged them back to real conversations.
- **Message History**: Initially, we decided to maintain the last **10 messages** as context. However, it became clear that this meant 5 user messages + 5 bot messages only, which was insufficient. To improve continuity, we expanded the window to **20 messages (10 from the user, 10 from the bot)**. This ensured the bot remembered more of the conversation and reduced inconsistencies in persona or details.
- **Context for Reply in Thread**: We updated the logic so that whenever a message is identified as a **reply in a thread**, the system passes the base/original message too in the message history. By combining them, the bot could generate a relevant and accurate response tied to the intended context.

Finally when everything seemed to work as expected we asked the founders to test and they said the chat had many issues and was nowhere close to what we had described.

This was odd as we all had tested it as I set with the developer to understand what's going on in the code and asked to see the system prompts, the version of the fine-tuned chatbot version we are using, etc to see if anything of them isn't from the old experiments. And yes it was the same case wherein when he was cleaning up a few things in code he had mistakenly changed the system prompt to an older one thinking that's the latest one.

As this was solved we again asked the founders to test and they were quite happy so we were ready to launch this new chatbot.

I created Mixpanel Boards to track things and also wrote SQL queries to compare how many messages users used to send before the new chatbot and what's that number after the release of this chatbot.

The numbers were quite shocking and how good the bot had become now was something everyone in the company noticed.

Few of the best results were:

- Average messages per bot chat **jumped from ~5 to ~15.**
- Session time length **doubled (~100 %).**
- UXR calls captured comments like "People here actually chat back; conversations flow." Users didn't realize most of those "people" were bots.

Somewhere in the middle of this I also realized two things. First, that careful data curation, privacy discipline, AppScript plumbing, and ruthless iteration can move a product a long way without a huge team. Second, I wanted to go further than this project allowed: agents, MCP patterns, front- and back-end wiring, image/video generation at production quality—the full 0→1→10 path. My health had taken a hit over the past couple of years; most weekends were spent at home learning, testing, or on webinars. The founders and my manager asked me to stay—offered to raise pay and sponsor courses—but I knew if I tried to "do both," I'd do neither well. I wanted a clean block of time to

learn deeply, reset priorities, and recover a bit. The decision wasn't reactive; it was a sum of things: wanting broader ownership, wanting recognition to match scope, and wanting to invest in skills I could compound for years. So with a heavy heart I finally went ahead with resignation to study AI full-time.

Perfect—thanks for the nudge. I've fixed the "validation at the end" bit (you verified SQL against raw numbers as you built each table, not just in the last days), and I've expanded the "how I debugged/learned/solved it" moments with concrete, step-by-step detail. Kept your narration and paragraph flow; only tightened grammar and clarity and added the deeper troubleshooting detail you asked for.

Retool Marketing Dashboard

Okay, so we started the Retool dashboard project—this was the last piece of work I did for the company. The toughest part was that I had to work directly with the founders, and the use case was critical. We needed to monitor various third-party websites like Shortimize. Tools like that let you create analytics reports for social media accounts and track all your videos across TikTok, Instagram, and YouTube. Once you add videos to a tracking sheet, the tool fetches updated numbers every few hours and gives you dashboard views by channel or video: what's trending, when the last video was uploaded, views/likes/comments, how a specific video performs relative to channel averages, and so on.

You get deep overviews for each channel and each video, but what's missing in these third-party solutions is sentiment: there's no built-in analysis of what users are commenting, how often the brand is mentioned, or whether particular creators are talking about you in the comments. We needed a custom, internal dashboard that could: (1) catch viral videos very early, (2) analyze comment sentiment, (3) track how often our app or handles are mentioned, and (4) support creator workflows (e.g., if creators are contracted to engage in comments and mention the app). Building this in-house would save thousands of dollars (Shortimize alone cost us ~$200/month, and we still needed 2–3

more tools) and could generate outsized marketing value. This became my project in my last weeks at Hunch.

Honestly, I didn't feel much energy at the start—I had just spent 8–9 days in the hospital. Coming out and jumping straight into a big, founder-facing project wasn't easy. We split responsibilities: another teammate owned Apify APIs, building the data pipeline (cleaning, storing, and automating updates every few hours into Redshift). I owned the Retool app: building a dynamic dashboard, wiring Redshift to Retool, writing SQL for all views, and handling the integrations. Since it came directly from the founders, I took it on and began.

First, I analyzed how Shortimize works because we wanted to broadly replicate it. I reviewed the dashboards we had set up there: tables, graphs, channel-level overviews, video-level detail, and per-video deep dives. We'd also built "collections," where each collection contains multiple accounts with its own overview and video set. Like any analysis platform, you get filters (last 7 days, by account, by video ID/title, etc.). I took screenshots of the Shortimize dashboards, shared links with GPT, and explained on-click behaviors and page transitions so it could grasp the UX. The goal: recreate something similar in Retool.

I had never used Retool before, so I made a free account, learned what's included vs. paid (found that we didn't need the paid version for now), and started studying. The big challenge: there weren't many good video tutorials, and most resources were text-heavy docs—not ideal when you need to see components in action. Still, I wanted to understand Retool before building. Without that, I'd be flailing.

At one point GPT suggested generating a Retool JSON I could import to scaffold pages instantly. Since we didn't have data yet (the data engineer was still wiring Redshift), I tried supplying dummy data. Even then, imports kept failing with different errors. I spent ~1 day trying, with long GPT threads, but nothing stuck. If that had worked, it would have saved a week. It didn't—so I switched back to fundamentals: learn Retool the normal way.

I found a handful of useful (though mostly year-old) YouTube videos and read some docs. On day three, my founder and I paired for ~6 hours. We built a side panel, a table, connected Retool

to Redshift, tested the connection, set permissions, and finally saw real data in the dashboard. I learned about local storage, actions, where to attach queries, and how to bind SQL outputs to tables. From then on we had daily calls: initially 2–3 hours, then ~1 hour each evening to sync on progress, blockers, and next steps.

It quickly became obvious this wasn't a one-week project. We had to link many moving parts and ensure data correctness. Writing SQL is easy when it's simple—but global filters complicate everything. If you pick "last 7 days," every table and chart must reflect it. If a table depends on another, filters must cascade correctly. We also needed to align on definitions (e.g., engagement rate, virality) and decide whether to use Shortimize's formulas as-is or tweak them. Custom columns required mapping multiple source fields. And for comments, we had no reference: we had to design the schema, decide how to visualize, and define sentiment analysis outputs from scratch. As with any custom build, the "unknowns" reveal themselves only as you build. We extended the timeline to two weeks to do it right and backtest queries thoroughly.

Because video resources on Retool were limited, I bookmarked the best few and avoided getting lost in docs. I started with the simplest tab: **Accounts**. It just listed accounts within a collection (total views, total comments, etc.). I shared screenshots with GPT, explained click behaviors and custom columns, and asked for guidance on how to replicate on Retool. My routine became: design UI components in Retool (drag/drop, hide/show conditions), then write SQL to power them. With GPT, I'd specify my tables, column names, and logic, but I kept ownership of the logic path so I could build my own reasoning muscle. If I was off, I asked GPT to correct me—and to comment on the SQL thoroughly so I could learn from it.

And here's where I was very particular: whenever I created a table or chart, **I immediately validated its SQL result against raw numbers**. I'd run the same logic on the Redshift SQL tab, sample a few accounts/videos, and cross-check totals and spot metrics (views, likes, comments). Only after the numbers matched did I move on. I repeated this pattern for every new component. This habit saved me from carrying silent inaccuracies forward.

We also had to map weird field names from Apify APIs into normal concepts. For example, a column called `diggscount` and `heartscount` both deal with likes wherein one is no. of likes given and another means no. of likes received. Because the data ingestion follows its own conventions, we had to translate those into human terms (views, likes, shares). That mapping fed every tab. For the **Overview** tab (the most complex), I needed playable videos. Embedding TikTok videos was tricky—n't working even after many tries and then as I researched about it I figured out that TikTok is quite strict in letting the videos playable outside it's platform so I rather did a workaround: show the video cover image linked to the real video URL. Clicking opened the video in a new tab. Not perfect, but good enough—and better than just a text link. I also wanted numerical badges on tab buttons (e.g., "Videos (100)", "Accounts (8)"). My charts were correct, but wiring those counts into button labels was tricky. It wasn't critical for correctness, so I deprioritized it to finish core charts and tables. Later, I circled back and implemented it.

Meanwhile, understanding how data was stored in Redshift was crucial. Were updates overwriting old rows or appending? That changes how you query. While cron jobs weren't running, my SQL queries worked fine. Once the every-two-hours pipeline started, results drifted—because I needed to select only the latest snapshot per video for dynamic metrics (views/likes/comments evolve) while treating account IDs and names as static. I updated queries to pick the most recent record per video. I also made sure every global filter (collections, accounts, specific videos, time ranges) flowed all the way into each SQL's `WHERE` and joins. Each time I changed logic, I manually re-validated the numbers against raw data for a handful of test cases.

Another real-world wrinkle: cron jobs can fail or skip some videos. That can make today's totals appear lower than yesterday's simply because a video's new data didn't arrive. I wrote down an operational rule of thumb for myself: whenever an aggregate suddenly dips, **first** check pipeline completeness for that interval before assuming a content trend. I flagged the need for data validations and alerts, plus a small run-book: if a video is missing, check if it was deleted, if the API call failed, or if rate-limit hit.

For seven-day comparisons, there's the off-by-one trap. To show daily deltas properly, you need D-8 to compute the first delta in a 7-day window. So I fetched an extra day for calculations but hid that in the display to keep the "Last 7 Days" honest. I learned to always write that extra-day logic as a comment block right inside the SQL so future me (or someone else) wouldn't "simplify" it and break the math.

There were times I just couldn't get something to reconcile. One example: time filters. Retool and my local queries weren't getting the same output for exact same matching queries. I tried multiple LLMs and asked the data engineer for help. We eventually found Retool was offset by time zone (IST vs. UTC) by ~5 hours. A tiny fix—half a day lost. The lesson: when things don't add up, check the simplest assumptions (time zones, locales, data types) before diving deeper. From then on, I broke problems into smaller pieces: first get UI visible, then wire data, then confirm components match Shortimize behavior or find the closest equivalent if Retool lacks something. For daily/weekly toggle views, I initially used the quick path (hide/show separate charts) and later refactored into conditional logic inside a single chart based on button state. Start with a working baseline; optimize once it's reliable.

Now that **V1** (Shortimize-style marketing dashboard) was largely complete it was time to work on **V2** (comments section with analysis scaffolding). A few columns were pending because I still needed founder alignment on definitions. I explicitly flagged those: if we keep them, here's what to decide; if not, we can drop them.

While most of the Retool work was about replicating Shortimize's metrics dashboards, the **comments analysis tab** was the one place we weren't just copying—we were inventing. This was a feature Shortimize didn't offer, and it quickly became the most original part of the project.

The idea was simple: views and likes tell you how a video performs, but **comments tell you what people actually think**. For Hunch, that mattered. If users were mentioning "Hunch" positively in comments, or comparing us against Tinder or Bumble, that was gold for marketing. If users were negative or mocking, we needed to know early. So this dashboard was designed

to give us visibility into the *qualitative layer* behind the numbers.

The first problem was deciding **what the comments dashboard should even show**. Apify gave us raw comment dumps, but they weren't structured for insights. I worked with the founders to define a schema:

- **Per video**: Show all comments for a specific video (not aggregated per account).
- **Base fields**: Comment text, commenter, likes on the comment, timestamp.
- **Replies**: Captured as child rows, so a parent comment could expand into a thread.
- **Sentiment**: Positive, neutral, negative — placeholder until the LLM pipeline was connected.
- **Mentions**: Count whenever "Hunch," "Tinder," "Bumble," or "Hinge" appeared in comments/replies.

That schema became the backbone. Everything else—filters, charts, sentiment summaries—built on top of it.

The first working version was a **plain table** of comments per video. I connected it to Redshift with a SQL query and cross-checked it against raw JSON dumps to confirm counts matched. Once stable, I added filters:

- **Timeframe** (last 7 days, last 14 days, last 30 days).
- **Keyword search** (find comments mentioning a word/brand).
- **Top-N selector** (show top 5, 10, 50, or all comments).

The Top-N filter was critical. Without it, pulling 500 comments in Retool slowed the UI to a crawl. By default, the table only loaded top 10 comments (ranked by likes), with a "load more" option. This kept the dashboard usable.

Replies were tricky. Apify stored them in a slightly different structure, and my first SQL joins were wrong. Some threads looked empty when they weren't. I manually picked a test video, opened its comments directly on TikTok, and compared. That's how I discovered the parent–child key mismatch. Once fixed, I built a second linked table in Retool: whenever you clicked a comment row, its replies appeared below in a child panel.

That small UX detail—expandable replies—was a first for us. Shortimize never gave that granularity.

I wasn't responsible for wiring the Sentiment tagging directly, but I drafted the **integration plan**:

1. Batch comments per video.
2. Send them to an LLM classifier for sentiment.
3. Use regex for brand mentions.
4. Save results back to Redshift with extra columns: `sentiment`, `brand_mention`.

On the Retool side, I created scaffolding that assumed those columns already existed:

- A pie chart of sentiment distribution per video.
- A counter widget: "Hunch mentioned 42 times."
- Filters: "show only negative comments," or "show comments mentioning competitors."

During testing, I mocked dummy sentiment values (e.g., randomly tagging 20% as negative, 50% neutral, 30% positive) to prove the UI logic. That way, when the LLM integration went live, everything would "just work."

One rule I followed: **validate every step against raw data**. Each new query or table, I ran the corresponding SQL directly in Redshift, exported counts, and cross-checked with what Retool showed. Example: a video with 320 comments in raw data had to

show 320 in the dashboard (even if only 10 were visible by default). This habit prevented surprises later.

I didn't leave validation to my last days—it was part of my daily rhythm.

By the end, this comments tab wasn't just "another dashboard." It was **our differentiator**. Shortimize could tell you which video was going viral. Our Retool dashboard could tell you *why* it was going viral—because users were laughing in comments, or because they were tagging Hunch, or because they were comparing us to Bumble. That qualitative layer made the dashboard not just analytics, but a marketing intelligence tool.

In my final days, I wasn't "finally" verifying SQL—I'd been validating each component as I built it from day one. What I did near the end was **one more sweep**: re-running my existing spot checks on the most business-critical tables after some pipeline changes, re-reading the comments in the queries, and noting optimization ideas I'd uncovered along the way. I also implemented the button-label counts ("Videos (100)", "Accounts (8)") to match Shortimize behavior, cleaned up hidden/unused elements, documented naming conventions (page-scoped prefixes, clear component and query names), and wrote a last to-do list: which columns still needed logic confirmation, and which data issues the data engineer should double-check. I refactored a couple of earlier quick hacks using things I'd learned over the month so whoever picked it up had the best possible base.

As my tenure was ending, I handed this project to a new teammate. I walked him through the project: what we're building, how Retool is set up, the tricks I used, how to break complex problems down, and which Retool concepts had tripped me up (actions, local storage, global vars, reuse patterns). I showed how I build charts (x/y fields, multi-series, query binding). We recorded the session, but one version had noise and the other had good audio but no screen. I then combined sources to make a usable handover. The founders asked me to stay longer to finish sentiment prompts and the LLM wiring, but I was still recovering post-hospital and my AI cohort was starting in a week. My doctor had advised rest, but founder-level projects don't always allow that. I was already working long hours and felt I had to stop. I declined, explained the medical reasons, and prepared the cleanest handover I could.

Before leaving, I documented everything in detail:

- The SQL logic for parent-child joins.
- Why Top-N selectors were added.
- How sentiment/mentions placeholders were set up.
- How to extend filters if new competitor names were added.
- Etc……

I also recorded a walkthrough for the new teammate, showing exactly how to debug comment mismatches or add new charts.

That's where the Retool task wrapped for me.

Handover

With just two days left in my notice period, I knew it was time to stop polishing edge cases and instead focus on what mattered most: leaving behind a **clean**, **structured handover**. I reached out to the founders directly and told them openly that, while I could keep refining the Retool dashboards endlessly, the more valuable thing for the company would be for me to write out and record everything — so the next person wouldn't be left guessing. They agreed, and that's how I switched gears from building to documenting.

I started with the **Retool project**, since it was the most critical and the one where so many small decisions had been made along the way. Even though I had already walked the new hire through the system in calls and shared recordings, I wanted a **written hand-off document** — something that explained not just what was built, but also the reasoning, the edge cases, the shortcuts taken, and the to-do's left behind. I detailed out the two big phases:

- **Phase 1: Replicating Shortimize** – the overview dashboards, the account-level metrics, the video filters, the global logic.
- **Phase 2: Extending with custom features** – especially the comments dashboard, where we had gone beyond Shortimize and

set the groundwork for sentiment analysis.

For every table and every chart, I explained the SQL query logic, the dependencies, and the naming conventions I had used — so the next person could follow a consistent system. I also flagged open items: certain columns where the founders still hadn't aligned, places where the logic could be made more efficient, and polish tasks that weren't strictly necessary for V1 but worth doing later.

After finishing Retool's handoff, I turned to **content**, which had been my other major vertical. Even though MBTI polls were being rolled out globally — reducing the volume of manual content work — I wanted to make sure no one was caught off guard if something backfired and we had to rely on older poll formats. Since my manager had left abruptly a few days earlier, I no longer had anyone to buffer this responsibility; it was on me to leave the team prepared.

So I wrote a **content hand-off document** as well. I included:

- The logic and usage of fixed polls, promoted polls, and MBTI polls.
- The system prompts for *Short Vibe* — including the new updates we had recently rolled out.
- Step-by-step instructions for using the **admin dashboard portal**: how to create campaigns, how to update or delete them, what fields to fill carefully, what fields to avoid touching.
- Notes on the recent changes in polls that required tweaks to *Short Vibe* content, so the next person wouldn't miss them.

To make this stick, I didn't just hand over docs. I also **scheduled a live handover call** with senior management, walking them through the portals, the dashboards, and the logic in real time. I encouraged them to record the call — and I also recorded a local backup myself, just in case. There were technical hiccups (one recording had noise, another lost the screen), but

I even went the extra mile to sync audio and video later, so at least one complete reference would exist.

By the time I was done, I felt confident I had left no gaps. Between the Retool documentation, the content hand-off, the recorded walkthroughs, and the flagged to-dos, the company had everything it needed to keep moving without me.

And then, on my final day, came something I didn't expect. The founder sent a farewell message on the company's main slack channel, describing my journey — from content writer to AI APM, and leading AI-driven projects. They highlighted how, despite being hospitalized for over a week, I had come back and taken on the Retool project, learned the tool from scratch, and delivered a full-fledged marketing dashboard that replaced costly third-party tools. They wrote that this was the first time in the company's history that someone would receive **150% of their promised bonus**, and they rated my leadership "4 out of 4."

What struck me most wasn't the bonus — though that was meaningful — but the recognition. The message closed by saying they would personally write me a reference letter, commending my maturity, technical growth, and the way I had used AI to transform my role. Reading that message on my last day was surreal. After three years of pouring myself into the work — nights, weekends, experiments, and all — it felt like the closure I didn't know I needed.