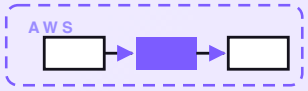
7-PART SERIES · FREE COMPANION



Loyalty tracker

A serverless points program for a small shop. Each purchase earns points by your rules; the system keeps every customer's balance, emails them when they've earned a reward, and lets staff redeem a reward with one tap. It nudges regulars who haven't been back in a while. Every points change is logged and reversible, and staff confirm each redemption so nothing is given away by mistake. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Loyalty tracker

PART 1 OF 7

JUNE 4, 2026 PART 1 OF 7 · LOYALTY TRACKER SERIES ~5 MIN READ

A loyalty tracker on AWS for a few dollars a month

A small shop's regulars are its lifeblood, and a points program is the simplest way to say thank you and keep them coming back. But the paper punch cards get lost, the spreadsheet never gets opened, and nobody at the counter remembers who's close to a free coffee. This post walks through the design of a small system that runs the whole program for you: it earns points on every sale by your rules, keeps each customer's balance exact, tells them the moment they've earned a reward, lets staff redeem with one tap, and gently nudges the regulars who've started slipping away.

KEY TAKEAWAYS

- Three ways a member is created: a sign-up form, a quick add by staff, or an auto-enroll on first sale.
- Every sale ends in one of four moves: skip, add points, add points and announce a reward, or hold for review.
- Your rules live in a short Drive doc: points per dollar, bonus items, and what each reward costs.
- Staff redeem with one tap, but always confirm first — nothing comes off a balance by accident.
- Designed on AWS for about $2 a month at typical small-shop volume.

The whole system on one page

Before any code, here’s the shape of what we’re designing.

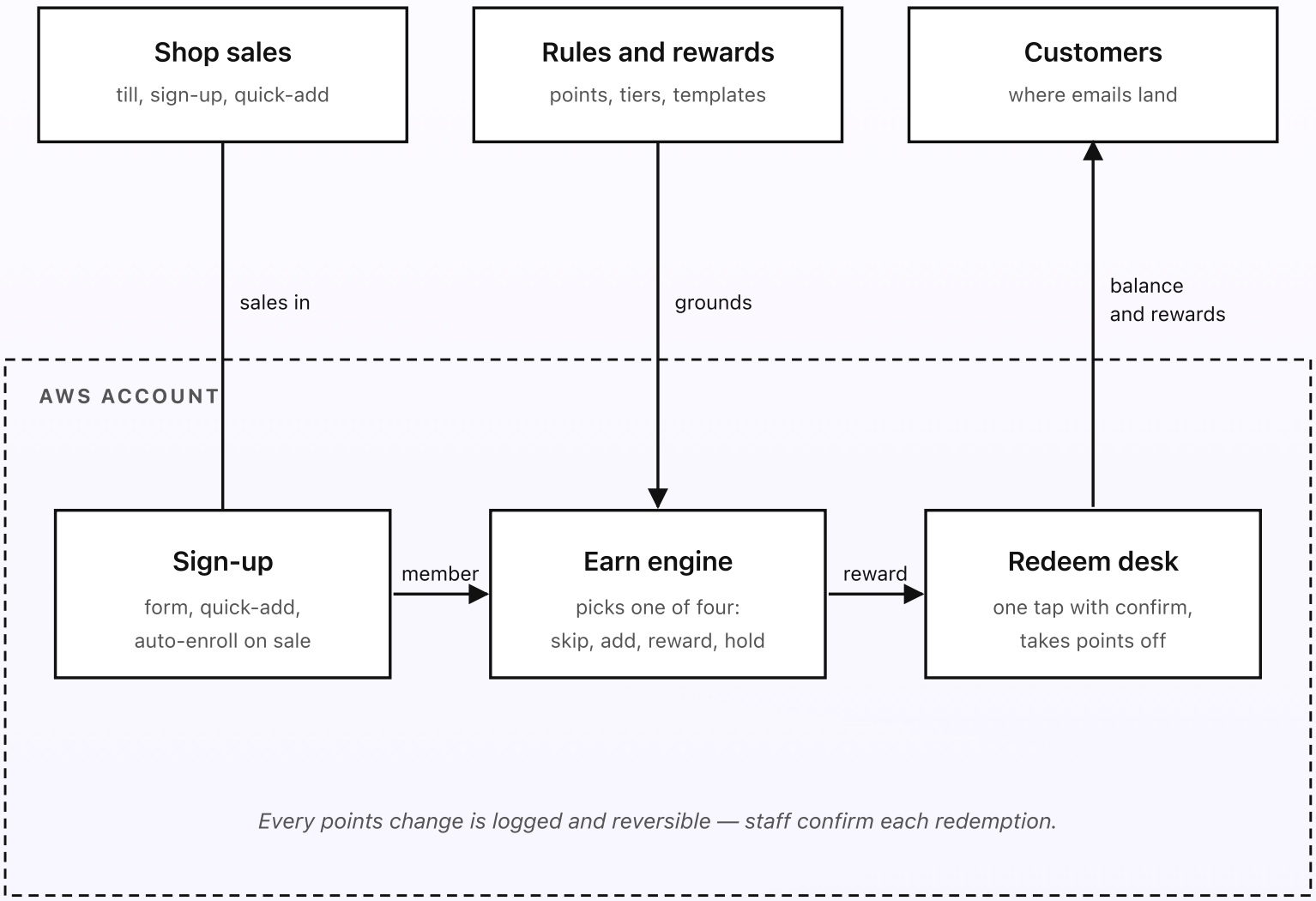

Shop sales
till, sign-up, quick-add
Rules and rewards
points, tiers, templates
Customers
where emails land
sales in
grounds
balance
and rewards
AWS ACCOUNT
Sign-up
form, quick-add,
auto-enroll on sale
member
Earn engine
picks one of four:
skip, add, reward, hold
reward
Redeem desk
one tap with confirm,
takes points off
Every points change is logged and reversible — staff confirm each redemption.

Fig 1. Three inputs outside, three pieces inside AWS. Members come in from a sign-up form, a staff quick-add, and an auto-enroll on first sale. The Earn engine runs on each sale and picks one of four moves. The Redeem desk lets staff give a reward with one tap and a confirm.

What you set up once (the outside)

- **Shop sales.** Your till sends each completed sale to the tracker — the customer's phone number or member id, the basket total, and the items bought. Most modern point-of-sale systems can send this as a small web request when a sale closes; if yours can't, a one-line script reads the day's sales export and posts them. New members can also come in two other ways covered in Part 2 — a sign-up form a shopper fills in, and a quick-add staff do at the counter in a few seconds.
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers how points are earned — how many points per dollar spent, which items earn bonus points ("double points on beans"), and the reward tiers ("100 points = a free coffee, 250 points = a free lunch"). The doc also holds the quiet-hours setting and how long a customer can be away before they count as lapsing. The *voice* doc holds the email templates — what the balance email and the reward-earned email actually say, in your shop's own friendly tone.
- **Customers.** Your members. Each has a phone number (how a sale is matched to them) and, if they gave one, an email address. Emails land with their new balance, how close they are to the next reward, a clear "you've earned a free coffee" note when they cross a tier, and an unsubscribe link. A customer with no email still earns points; they just check their balance at the counter.

What runs on every sale (the inside)

- **The sign-up.** Three ways a member is created. A shopper fills in a short web form (name, phone, email) and lands as a member. Staff add someone at the counter in a few taps. Or a sale comes in for a phone number the system has never seen, and the earn engine creates a member on the spot so the points aren't lost — the customer can fill in their email later. Part 2 walks through all three.
- **The earn engine.** Runs on each sale. Reads the rules doc. Works out the points the sale earns — base points from the total, plus any bonus-item points. Then picks one of four moves. *Skip*: the sale has no member attached and no phone to match — do nothing. *Add points*: the normal case — add the points and update the balance. *Add and announce a reward*: the new balance crosses a reward tier — add the points and email the customer that they've earned a reward. *Hold for review*: something looks off (a basket far larger than this customer ever buys, or a refund that would push points negative) — park it for a staff member to check. The earn engine calls no model; the math is plain Python.
- **The redeem desk.** A simple lookup screen for staff. Type a phone number, see the customer, their balance, and any reward they can claim. Staff tap the reward, the screen shows what it costs in points, staff press *Confirm*, and the points come off. The system checks the balance is high enough first, locks the customer so the same reward can't be redeemed twice in two seconds, and writes the redemption to the ledger. A nightly email goes to anyone whose balance changed that day, and a weekly sweep finds regulars who've gone quiet so they can be nudged. A monthly summary writes the owner a short plain-English read on how the program is doing.

In plain words

Maria buys a coffee and a bag of beans on Tuesday morning, $18 total. She joined the program last month, so her phone number is on file. The till sends the sale; the earn engine reads the rules (1 point per dollar, double points on beans), works out 18 base points plus 6 bonus points for the beans, and adds 24 points. Her balance goes from 84 to 108, which crosses the 100-point tier. The engine emails her: "You've earned a free coffee — show this at the counter next time." On Friday she's back, orders her usual, and mentions the free coffee. The barista types her number, sees the reward, taps it, the screen shows "Free coffee — 100 points," the barista presses Confirm, and 100 points come off. Maria's balance is now 8, the redemption is logged with the barista's name and the time, and everyone moves on.

The cost of running this is about $2 a month at small-shop volume. The cost of *not* running it is the regular who quietly drifts to the cafe across the road because nobody ever made them feel like a regular.

DESIGN RULES THAT SHAPED EVERY DECISION

- Points and money are exact. The earn math and the redeem check are plain Python — no model, no guessing.
- Four moves, always. Skip, add points, add and announce, hold for review. There is no fifth.
- Staff confirm every redemption. One tap is fast; the confirm step is what stops a reward going out by mistake.
- Every points change is logged and reversible. A wrong redemption can be undone in one tap.
- The rules live in Drive. Changing points-per-dollar or a reward tier doesn't need a deploy.
- Emails respect quiet hours and the unsubscribe link. A loyalty email should feel like a treat, not spam.

Why this shape

Most shops run loyalty one of three ways: a paper punch card that customers lose, a spreadsheet nobody opens, or an expensive app with a monthly fee that does ten things you don't need. The punch card works until the customer leaves it in their other coat. The spreadsheet goes stale the first busy week. And the big app charges you a subscription whether you have 50 members or 5,000.

The setup above keeps the rules somewhere the shop owner already edits — a plain doc — but adds a small system that watches every sale and acts only when

there's a point to add or a reward to announce. Balances are always exact because they live in the system's own store, not a doc that two people might edit at once. Rewards can't go out by accident because staff confirm each one. And nothing is ever lost, because every points change is written down and can be undone. The tracker is invisible most of the time; it only speaks up when a customer earns something worth knowing about.

The next four posts walk through each piece in turn: how a shopper joins the program, how a purchase earns points, how a reward gets redeemed, and how a lapsing regular gets nudged back. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 4, 2026 PART 2 OF 7 · LOYALTY TRACKER SERIES ~4 MIN READ

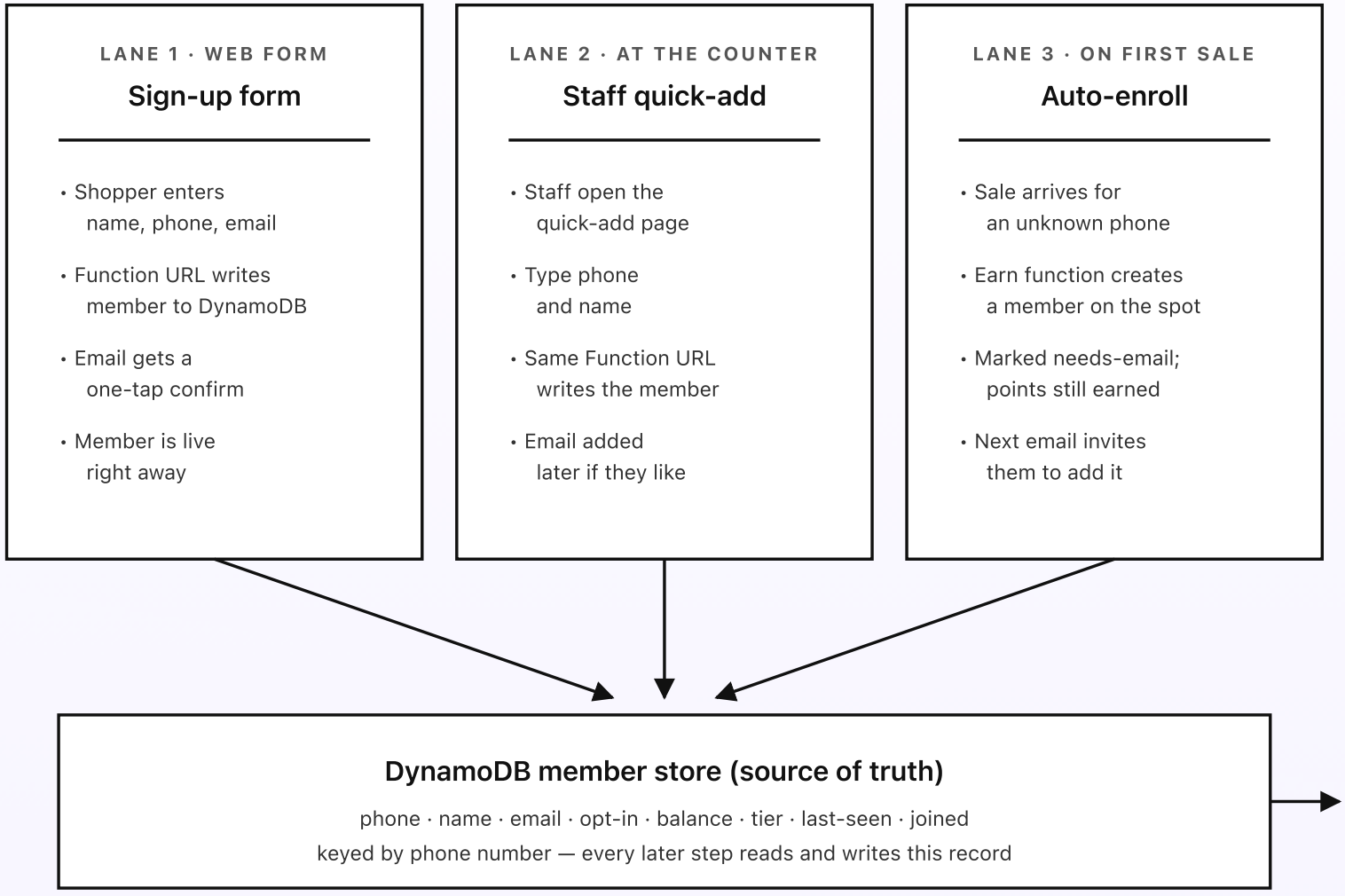
How a shopper joins the loyalty program

The tracker only earns points for people it knows about. So the first job is making it easy to become a member — easy enough that nobody loses points just because signing up was a hassle. There are three ways a shopper joins: they fill in a short form, a staff member adds them at the counter, or a sale comes in for a phone number the system has never seen and it creates the member on the spot. The first two are obvious. The third exists because in real life a customer says “yeah, add my number” while three other people are waiting.

KEY TAKEAWAYS

- Three sign-up lanes feed one member store: a web form, a staff quick-add, and auto-enroll on first sale.
- The member store is DynamoDB, keyed by phone number, because balances must be exact and fast.
- Auto-enroll never loses points — an unknown phone on a sale creates a member before points are added.
- Email is optional and confirmed once, so the shop only ever mails people who asked for it.
- The same member record is used by every later step: earn, redeem, and the lapsing nudge.

Three lanes into one member store



The member store is the source of truth — every later step reads and writes the same record.

Fig 2. Three lanes converge on one member store. The store is the source of truth; the form, the quick-add, and auto-enroll all write into the same DynamoDB record keyed by phone number, so the earn engine, the redeem desk, and the lapsing nudge all read one place.

Lane 1: the sign-up form

The simplest lane for the customer who has a minute. A short web page — a QR code on the counter or a link on the receipt points to it — asks for a name, a phone number, and an optional email. When they submit, the page posts to a Lambda Function URL (a small web address that runs a function directly, with no extra gateway in front). The function writes a new member to the DynamoDB member store keyed by their phone number, sets the balance to zero, and stamps the join date.

If they gave an email, the function sends a one-tap confirm message: “Tap to start getting your balance and rewards by email.” Only after they tap is the email marked confirmed and opted in. This double-check means the shop never emails someone who fat-fingered a stranger’s address into the form, and it keeps the shop on the right side of the rules about marketing email.

Lane 2: staff quick-add (the lane most shops actually use)

Most people sign up because a staff member offered, not because they found a form. So staff get their own small page: type a phone number and a name, tap save, done. It posts to the same Function URL as the form and writes the same kind of record. Email is left blank; if the customer wants their balance by email, staff can add it later or the customer can tap the link in their first balance email.

The quick-add page is deliberately bare — two fields and a save button — because it gets used while a queue is forming. A staff member should be able to enroll a customer in the time it takes to hand over change. Anything that slows that down means fewer people join, which means a weaker program.

Lane 3: auto-enroll on first sale

The most important lane, and the one that quietly does the most work. A sale comes in from the till with a phone number attached — maybe the customer gave it verbally and staff typed it into the till, maybe the till captured it some other way — but the system has never seen that number. Instead of dropping the points on the floor, the earn function creates a member there and then, adds the points, and marks the record as `needs-email`.

The customer is now a member with a real balance, even though they never filled in a form. The next time a balance email would go out, the system has no address yet, so it does the next best thing: it leaves the points safely banked and waits. When the customer does sign up properly later — via the form or a staff quick-add with the same phone number — the existing record is found and updated, not duplicated, so the points they already earned are right there waiting. No customer ever loses points because the paperwork came after the purchase.

Why the member store stays the source of truth

Three lanes in, but only one place where a member actually lives. That's a deliberate constraint. Balances change on every sale and have to be exact to the point, so they live in DynamoDB — the system's own fast, reliable store — not in a doc that two people might edit at the same time. Keying by phone number means

a sale, a redemption, and a nudge all find the same person without anyone having to remember a member id. The rules and the email wording live in Drive where the owner can edit them freely; the member balances live where exactness matters. Each thing lives where it belongs.

Next post: how a purchase actually earns points — how the earn function reads the rules, does the math, and picks one of four moves.

PART 3 OF 7

JUNE 4, 2026 PART 3 OF 7 · LOYALTY TRACKER SERIES ~5 MIN READ

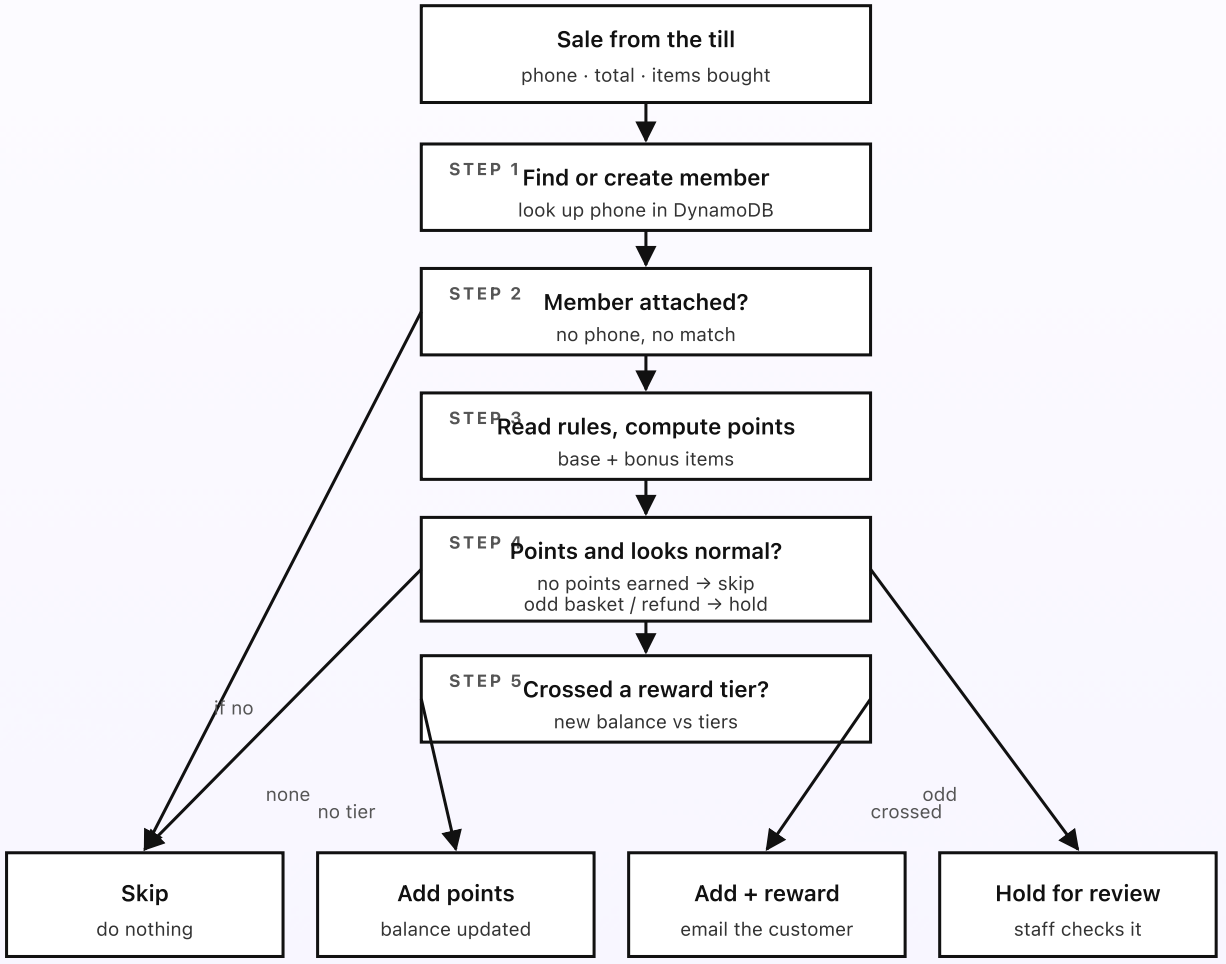
How a purchase earns points

Every time a sale closes, the till sends it to a Lambda Function URL and the earn function takes over. It finds the member, reads the rules, works out how many points the sale earns, and decides whether to do nothing, add the points, add the points and announce a reward, or park the sale for a staff member to check. The whole decision is plain Python. No model. No guessing. Every number lives in the rules doc, where the owner can change it without a deploy.

KEY TAKEAWAYS

- The earn function runs on each sale, posted from the till to a Lambda Function URL.
- The rules doc holds every number — points per dollar, bonus items, and the reward tiers.
- Four moves per sale: skip, add points, add and announce a reward, or hold for review.
- The member's balance is updated with a safe one-at-a-time write so two sales can't clash.
- The earn function never calls a model. Points are money, so the math is fully predictable.

The decision flow, per sale



The rules doc holds every number — change points-per-dollar and the next sale uses it.

Fig 3. The earn function's decision tree, per sale. Five steps decide which of four moves applies. The rules doc holds every number; the earn function only applies them.

The rules doc: points-per-dollar isn't magic, it's in the doc

The rules doc reads in plain prose: "Earn 1 point per dollar spent. Beans and gift cards earn double points. Reward tiers: 100 points = a free coffee, 250 points = a free lunch, 500 points = $25 off. Members can be away 60 days before they count as lapsing." The earn function reads these numbers from the doc (mirrored to S3 so it doesn't hit Drive on every sale) and does the arithmetic. Nothing is hard-coded; the owner can change a tier or add a bonus item by editing the doc, and the next sale picks up the change.

Per-item overrides are easy too. The rules doc can name specific product codes that earn a flat bonus ("each bag of beans: +6 points") or a multiplier ("all whole-bean products: 2× points"). The till sends the line items, so the earn function can match them against the bonus list and add the extra points. This is how a shop runs a "double points weekend" without a developer — add a line to the doc on Friday, remove it on Monday.

Four moves, always

Every sale, every time, lands in exactly one of four buckets. The names are simple on purpose.

- **Skip.** The sale has no member attached and no phone to match against — a walk-in who isn't in the program. Do nothing. Most anonymous sales are skips, and that's fine.
- **Add points.** The normal case. The sale earns points, the new balance doesn't cross a reward tier, so add the points and update the balance. Write a row to the points ledger so the change is recorded.
- **Add + reward.** The points push the balance over a reward tier. Add the points and email the customer: "You've earned a free coffee — show this at the counter." The reward is now claimable at the redeem desk. Write the ledger row and the reward-earned note.
- **Hold for review.** Something looks off. The basket is many times bigger than this member ever buys (a possible mistyped total), or the sale is a refund that would push the balance below zero. Park it on a small review list for a staff member to approve or reject. Nothing is added until a human looks. This is the rare case, but it's the one that keeps a typo from handing someone 5,000 points.

Balance writes that can't clash

On a busy morning the same customer might somehow trigger two sales seconds apart, or a sale and a redemption might land at the same moment. If both read the old balance and both wrote, one change would be lost. The earn function avoids this by updating the balance with a safe one-at-a-time write — it tells DynamoDB "add 24 points to whatever the balance is right now," rather than reading the number and writing back a total it computed. DynamoDB applies these one at a time, so no points go missing even when two things happen at once. Redemptions in Part 4 use the same trick in reverse.

Every move also writes a row to the `loy-ledger` table: `(member, sale_id, points_change, reason, timestamp)`. The balance is the running total; the ledger is the history. If a balance ever looks wrong, you can replay the ledger and see exactly how it got there — which sale, which bonus, which redemption. That's what makes every change reversible.

Why the earn path uses no model

The earn function could call a model to, say, write a more charming reward email or guess whether a basket is suspicious. It doesn't. Two reasons. First, points are money — a customer who earns 24 points must always earn exactly 24 points, and a model in that loop introduces variance nobody wants near a balance. Second, this runs on every single sale, so a model call would cost money thousands of times a month for no gain. The math is a few lines of plain Python and zero surprises.

Bedrock fires elsewhere — once a month, to write the owner the plain-English program summary covered in Part 6. Not on the earn path. The earn function reads a doc, does arithmetic, updates a balance, and writes a ledger row.

Next post: how a reward gets redeemed — how staff look a customer up, what the confirm step does, and the guardrails that stop a reward going out by mistake.

PART 4 OF 7

JUNE 4, 2026 PART 4 OF 7 · LOYALTY TRACKER SERIES ~5 MIN READ

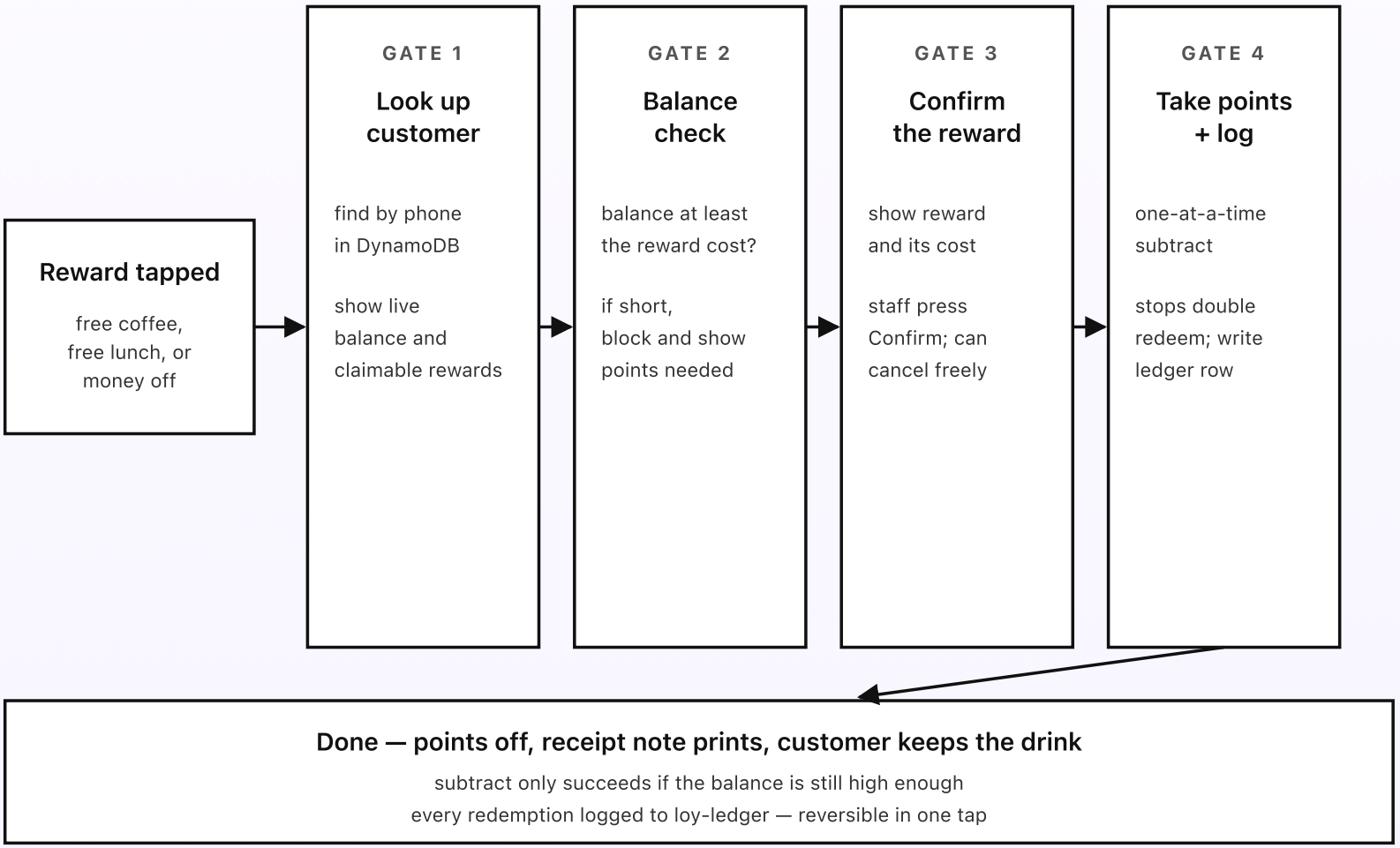
How a reward gets redeemed

A customer comes back and wants to claim their free coffee. A staff member looks them up, taps the reward, and the points come off. That has to be fast — nobody wants to fiddle with a screen while a queue forms — but it also has to be safe, because the points are worth real money. Give a reward away by mistake, redeem the same one twice, or let a balance go negative, and the program quietly leaks money. Four small guardrails sit between the staff tap and the points actually coming off.

KEY TAKEAWAYS

- Staff look the customer up by phone; the screen shows the balance and the rewards they can claim.
- The balance is checked first, so a reward can never be redeemed on too few points.
- A confirm step sits between the tap and the points coming off — nothing happens by accident.
- A one-at-a-time lock stops the same reward being redeemed twice in two seconds.
- Every redemption is logged with the staff member and is reversible in one tap.

Four guardrails on every redemption



Every gate is a deterministic check — no model calls, no reward given away by accident.

Fig 4. Four guardrails between the tap and the points coming off. Look the customer up. Check the balance. Confirm the reward. Take the points with a safe write and log it. Then the points are off and the redemption is reversible.

Gate 1: look up the customer

Staff open the redeem page and type the customer's phone number. The page calls a Lambda Function URL that reads the member from DynamoDB and returns the live balance plus the list of rewards that balance currently allows. The key word is *live* — the screen always shows the balance as it is this second, not a number that was right when the page loaded five minutes ago. If the customer just earned points on the sale they're paying for right now, those points are already counted.

The page shows only rewards the customer can actually claim. If their balance is 108 and the tiers are 100 and 250, they see the free coffee but not the free lunch. There's no way to tap a reward they can't afford, which removes a whole class of mistake before it can happen.

Gate 2: balance check

Even though the screen only offers affordable rewards, the function checks the balance again on the server before doing anything. This matters because two things could have happened between the screen loading and the tap — another till could have redeemed a reward, or a refund could have lowered the balance. So the redeem function re-reads the balance and confirms it's at least the reward's cost. If it's short, the redeem is blocked and the screen says exactly how many

more points are needed. The customer is never told “done” for a reward they couldn’t afford.

Gate 3: confirm the reward

This is the human guardrail. The screen shows the reward name and its point cost in plain text — “Free coffee — 100 points” — and a single *Confirm* button. Nothing comes off the balance until the staff member presses it. They can cancel with no change at all. It sounds small, but this one deliberate tap is the difference between “I meant to do that” and “oops.” Speed comes from the lookup being instant; safety comes from the redeem itself never being a single accidental tap.

The confirm screen is also where a manager override would live if you want one — for example, letting a manager redeem a reward the customer is a few points short of, as a goodwill gesture. That override is itself logged with the manager’s name, so a kindness is recorded just like everything else.

Gate 4: take the points with a safe write, then log

On Confirm, the function subtracts the reward’s cost from the balance with a one-at-a-time write that only succeeds if the balance is still at least that high. This does two jobs at once. It makes the final balance check airtight — the subtraction can’t run on a balance that dropped in the last instant. And it stops a double redeem: if two staff somehow tap Confirm for the same reward at the same moment, only the first subtraction succeeds and the second is rejected, so the reward is given once and the points come off once.

Then the function writes a redemption row to `loy-ledger`: `(member, reward, points_spent, by_staff, timestamp)`. The balance is the running total; the ledger is the history. Because the cost is recorded as a negative points change, reversing a wrong redemption is just adding the same points back and writing a matching “reversed” row — one tap on an admin screen, fully logged. The customer who was charged for a reward they didn’t get can be made whole in seconds, and the trail shows exactly what happened.

Why the guardrails exist

None of these gates are clever. They’re the care a careful person would take if they were doing this by hand — check who this is, make sure they’ve actually got the points, say out loud what you’re about to do before you do it, and write it down so you can undo it if you got it wrong. Putting them in code as four small steps makes them part of the design, not something you’re trusting a busy barista to remember at the height of the morning rush.

Next post: how a lapsing regular gets nudged — how a weekly sweep finds the regulars who’ve gone quiet, and the three things the shop can do about each one.

PART 5 OF 7

JUNE 4, 2026 PART 5 OF 7 · LOYALTY TRACKER SERIES ~5 MIN READ

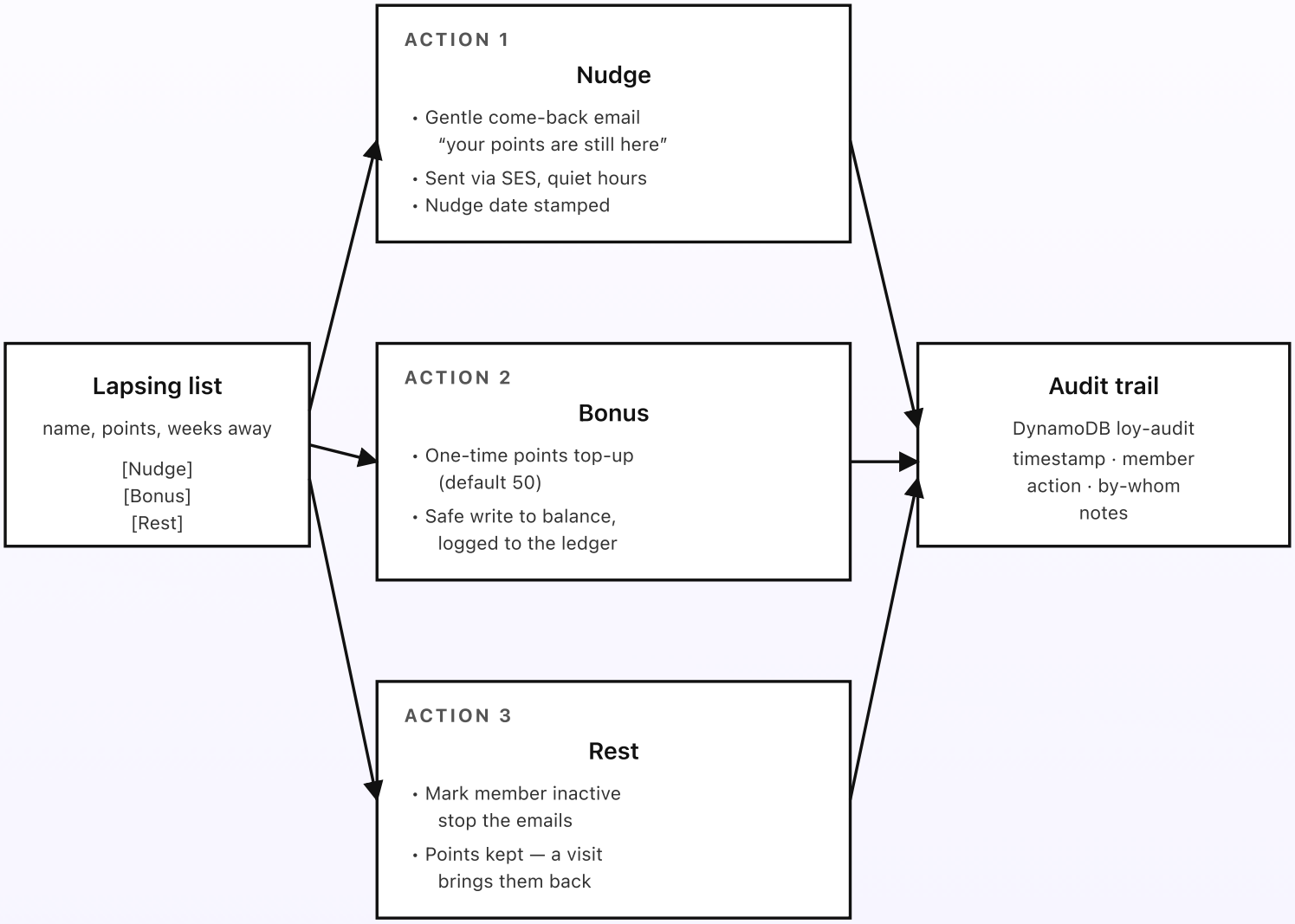
How a lapsing regular gets nudged

Maria used to come in three times a week. It's been five weeks since her last visit. Maybe she's on holiday, maybe she's found another cafe, maybe life just got busy. A weekly sweep notices regulars like Maria and puts them on a short list. The question is what to do about each one — and the honest answer is "it depends." This post walks through the three things the shop can do about a lapsing regular — nudge, bonus, rest — and how the member record, the email log, and the audit trail all stay in sync.

KEY TAKEAWAYS

- A weekly sweep finds members whose last visit is older than the lapsing window in the rules doc.
- Three actions per lapsing member: *nudge* (a gentle email), *bonus* (a points top-up), *rest* (stop the emails).
- Each action updates the member record and writes an audit row.
- Nudges are bounded — a member only gets so many before they're rested automatically.
- Every email respects quiet hours and the unsubscribe link.

Three actions on a lapsing regular



A rested member keeps their points — resting only stops the emails. A visit brings them back.

Fig 5. Three actions per lapsing regular, three different effects. Nudge sends a gentle email. Bonus adds a one-time points top-up. Rest stops the emails without touching the points. Every action writes to the audit trail.

Finding the lapsing regulars

Once a week, an EventBridge Scheduler rule fires a small sweep Lambda. It reads the member store and the rules doc, where the owner has set a lapsing window — say 60 days. For each member, it compares the last-seen date against that window. A member who's past it, and who was a genuine regular (more than a handful of visits, so a one-time tourist doesn't clutter the list), goes onto the lapsing list. The list is the input for everything below.

The sweep itself just builds the list and stamps each member as lapsing; it doesn't email anyone on its own. What happens next is a choice — either the owner reviews the list and picks an action per member, or the rules doc says "auto-nudge once, then wait," and the sweep applies the gentle default. Either way, the three actions are the same.

Action 1: nudge (the gentle reminder)

The most common action. *Nudge* sends a warm, short come-back email from the voice template: "We've missed you — you've still got 108 points waiting, that's a free coffee and then some." It goes out through SES, respects the quiet-hours setting (no marketing email late at night), and always includes the unsubscribe link. The member record is stamped with the nudge date.

Nudges are bounded. The rules doc has a `max_nudges` setting (default two). After that many nudges with no visit, the member is rested automatically — the shop stops emailing someone who's clearly not coming back, which is both kinder and keeps the shop's email reputation clean. A loyalty program that keeps pestering people who've moved on isn't loyalty; it's spam.

Action 2: bonus (the win-back gift)

For a regular worth fighting for — someone whose lifetime spend says they were a real fixture — a gentle reminder might not be enough. *Bonus* adds a one-time points top-up (default 50) and sends the matching email: "Here's 50 points on us — come grab a coffee." The top-up goes onto the balance with the same safe one-at-a-time write the earn engine uses, and it's written to the points ledger as a `reason: win-back-bonus` row, so the gift is recorded just like an earned point.

The bonus is a deliberate cost, so it's the action the owner usually takes by hand rather than letting the sweep do it automatically. The audit row names who granted it, so at the end of the month the owner can see exactly how many points were given away as win-back gifts and whether those customers came back.

Action 3: rest (the "leave them be")

Sometimes the right move is to stop. The member moved out of town. They asked to be left alone. They've been nudged the maximum number of times and didn't bite. *Rest* marks the member inactive and stops the lapsing emails entirely. They keep every point they earned — resting touches the emails, not the balance. If

they ever walk back in and make a purchase, the next sale finds their record, unrests them, and they're a regular again with their points intact.

A rested member also drops off future lapsing lists, so the owner's weekly review stays focused on people who might actually come back. The list never fills up with the same long-gone names week after week.

Every action is logged, every action is reversible

The `loy-audit` table records every nudge, bonus, and rest with who took the action, the timestamp, and a snapshot of the member before and after. If a bonus goes to the wrong person, or someone is rested by mistake, an admin can reverse it — add the points back, or flip the member active again — and the undo is itself an audit row, so the trail stays clean.

This kind of reversibility matters because win-back is the one place the program spends real money on a guess. Being able to look back and say "we gave 1,200 points in win-back bonuses last quarter and 40% of those customers returned" is what turns a nice gesture into a decision you can actually judge.

Next post: the cost breakdown. The whole program above runs in coffee-money territory at small-shop volume; Part 6 explains exactly where the dollars go.

PART 6 OF 7

JUNE 4, 2026 PART 6 OF 7 · LOYALTY TRACKER SERIES ~3 MIN READ

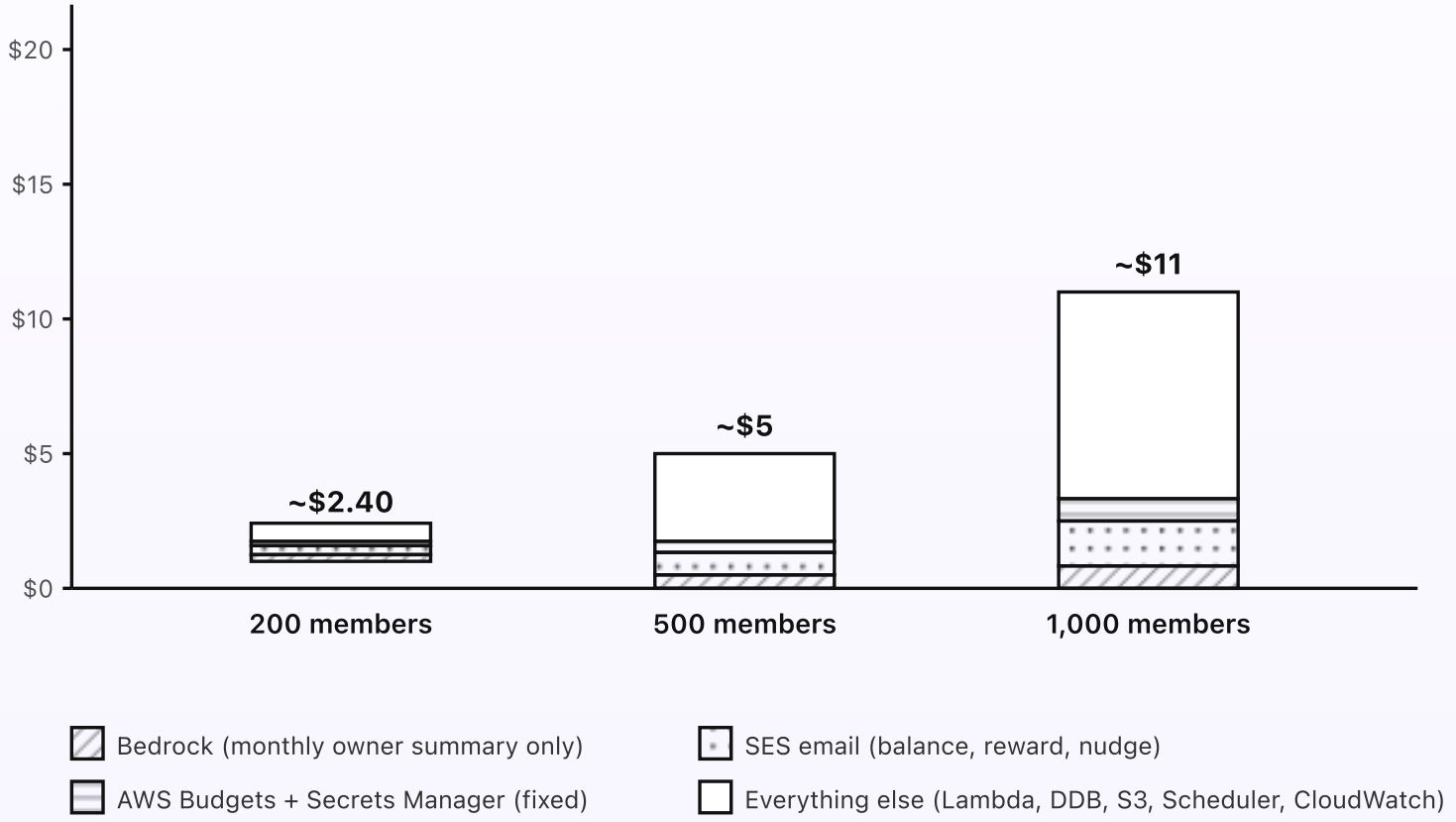
What the loyalty tracker costs

The loyalty tracker is one of the cheapest systems in this whole series. Each sale runs a tiny function that reads a few numbers, does some arithmetic, updates a balance, and writes a ledger row. Redeems are just as small. The only emails are the reward-earned note, a short nightly balance email, and the odd win-back nudge. It calls no models on the busy paths. Bedrock fires only once a month for the owner summary. At typical small-shop volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.40/month at typical small-shop volume (about 200 members, ~1,500 sales a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The earn and redeem paths cost pennies — no model calls.
- Bedrock fires only on the monthly owner summary, so it's a small sliver.
- At 500 members the bill is around $5. At 1,000 members and 8,000 sales it's around $11.

Cost at three volumes



The per-sale earn function is the dominant cost — and even that is fractions of a cent per sale.

Fig 6. Monthly cost at three shop volumes. Bedrock is a small sliver because it only fires once a month for the owner summary. The dominant cost is the everything-else bucket: the earn function running on every sale.

Where the dollars actually go

Lambda runtime (the bulk). The earn function runs once per sale. Each run reads the rules from S3 (cached on a warm function), finds the member, does a little arithmetic, updates the balance, and writes a ledger row — a few dozen milliseconds. At 1,500 sales a month that's pennies; at 8,000 it's still well under a dollar. Add the redeem function for each reward claimed, the sign-up function, the nightly balance-email job, the weekly lapsing sweep, and the drive-sync Lambda every fifteen minutes — the Lambda total lands under a dollar at all three volumes.

DynamoDB on-demand. Three small tables: the member store, `loy-ledger`, and `loy-audit`. Each sale is one read and a couple of writes; each redeem the same. Pennies a month at any of these volumes.

S3 + storage. The mirrored rules and voice docs, plus a daily backup of the member store. A few hundred KB total at small-shop volume. Effectively free.

EventBridge Scheduler. The nightly balance-email rule, the weekly lapsing sweep, the monthly summary, and the fifteen-minute drive-sync. A handful of invocations a day. Pennies.

SES email. Outbound for the reward-earned notes, the nightly balance emails, and the win-back nudges: $0.10 per thousand sent. At a few thousand emails a month that's a few cents to a couple of dollars. The nightly email only goes to members whose balance changed that day, which keeps the count down.

Bedrock (only the monthly summary). The earn and redeem paths use no Bedrock. Once a month, the summary function calls Haiku 4.5 to write the owner a short plain-English read on the program: a few thousand input tokens (the

month's counts) and a few hundred output tokens (the paragraph). A couple of cents a month, every month, regardless of volume.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the POS webhook, the sign-up form, and the redeem desk.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The functions only run when a sale, a redeem, or a scheduled job fires.
- **A Knowledge Base.** Balances are structured numbers, not free text — an exact lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the busy paths.** The earn math and the redeem check are plain Python. Bedrock fires only on the monthly summary.

How the cost scales

Lambda runtime and DynamoDB grow roughly with the number of sales, because the earn function runs once per sale. SES grows gently with the number of emails sent. Bedrock is flat — one call a month no matter how big the shop is. So the bill at 3,000 members and 25,000 sales a month is around $28; at 6,000 members and 50,000 sales it's around $55. Past those volumes you're a chain, not a small shop, and you'd batch the nightly emails and add a read cache — but those are tweaks, not redesigns.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The normal small-shop bill stays well under that ceiling.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, DynamoDB schemas, SES config, and EventBridge Scheduler config.

PART 7 OF 7

JUNE 4, 2026 PART 7 OF 7 · LOYALTY TRACKER SERIES ~8 MIN READ

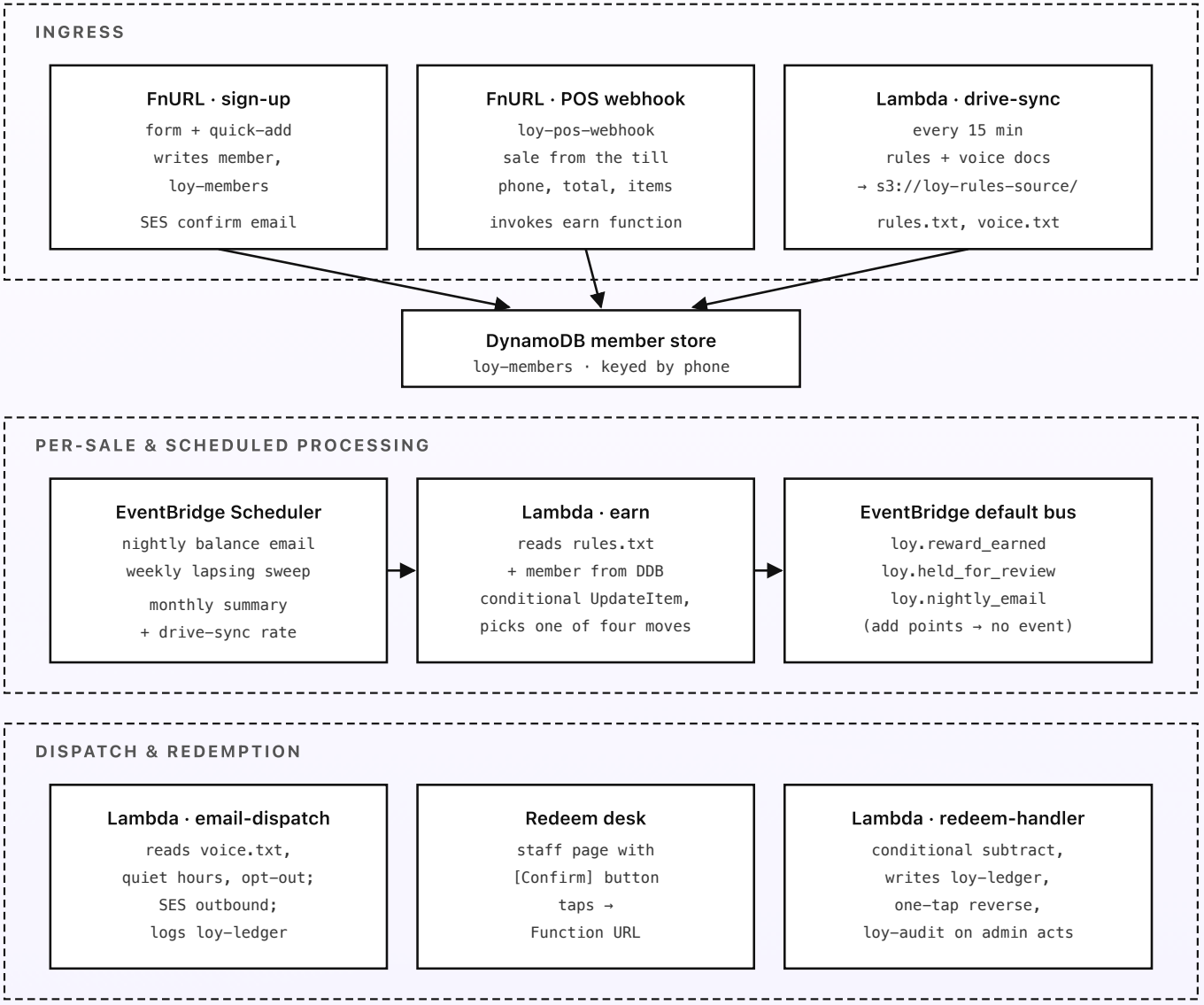
Engineering reference: the loyalty tracker architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES outbound config, EventBridge Scheduler config, the DynamoDB schemas, and the conditional-write pattern that keeps balances exact. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES outbound, Bedrock Global cross-Region inference, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for a shop is a balance email that goes out a few minutes late, not a regional outage. One AWS account dedicated to the tracker (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every points change is logged to loy-ledger — and is reversible.

Fig 7. AWS topology, in three regions of the diagram: ingress (sign-up, the POS webhook, and the rules sync), per-sale and scheduled processing (the earn function and the timed jobs emitting events), dispatch and redemption (emails go out and staff redeem with a confirm). Every Lambda is event-, request-, or schedule-driven; nothing is synchronous-chained.

Lambda functions

All Lambdas use the `arm64` architecture, the smallest memory size that meets latency targets (typically 256 MB), Python 3.14 runtime, and CloudWatch Logs at 7-day retention. Each function has its own least-privilege IAM role. None run inside a VPC.

- **`loy-pos-webhook`** — Lambda Function URL, `AuthType: NONE`; verifies a shared-secret HMAC header that the POS includes on each request. Receives one completed sale (phone or member id, basket total, line items), validates the payload, and invokes `earn` synchronously. If the till can only export a file, a small companion script reads the daily sales export and posts each row to this URL. Memory: 256 MB. Timeout: 15 s.
- **`earn`** — the per-sale points engine. Finds or creates the member in `loy-members`; reads `rules.txt` from `s3://loy-rules-source/` (cached for the lifetime of the execution environment); computes base + bonus points; runs the four-move decision from Part 3. On *add* and *add + reward*, applies a conditional `UpdateItem` with `ADD balance :pts` so concurrent sales never lose points; writes a `loy-ledger` row; and on a tier crossing emits `loy.reward_earned` to the default bus. *Hold for review* writes to `loy-review` and emits `loy.held_for_review`. Memory: 256 MB. Timeout: 15 s. *No Bedrock calls.*

- **`signup`** — Lambda Function URL backing both the web form and the staff quick-add. Writes a member to `loy-members` with `PutItem` + a `attribute_not_exists(phone)` condition so a re-submit of the same phone updates rather than clobbers; on a supplied email, sends an SES double opt-in confirm and only sets `opted_in = true` when the confirm link is tapped (a second tiny Function URL path). Memory: 256 MB. Timeout: 15 s.
- **`redeem-handler`** — Lambda Function URL, `AuthType: NONE`; verifies a staff session token. Backs the redeem desk: `GetItem` the member, return claimable rewards; on Confirm, apply a conditional `UpdateItem` (`ADD balance :neg` with `ConditionExpression balance >= :cost`) so the subtract only succeeds when funds remain, which also blocks double-redeem; write the redemption to `loy-ledger`. A `reverse` action adds the points back and writes a matching `reversed` ledger row plus a `loy-audit` row. Memory: 256 MB. Timeout: 15 s.
- **`email-dispatch`** — EventBridge rule on `loy.reward_earned` and the scheduled email events. Reads the matching template from `voice.txt`, fills placeholders (balance, points to next reward, reward name), checks the quiet-hours window and the opt-out flag, and sends via SES `SendEmail`. Writes a send record. Memory: 256 MB. Timeout: 30 s.
- **`nightly-balance`** — EventBridge Scheduler target, nightly. Queries `loy-ledger` for members whose balance changed today, and for each emits a `loy.nightly_email` event that `email-dispatch` turns into a short balance email. Batches recipients to stay within SES rate limits. Memory: 256 MB.
- **`lapsing-sweep`** — EventBridge Scheduler target, weekly. Scans `loy-members` for active members whose `last_seen` is older than the lapsing window in `rules.txt` and whose visit count clears the regular threshold; writes them to

`loy-lapsing`, stamps each as lapsing, and applies the configured default action (auto-nudge or owner-review). Bounded by `max_nudges`. No Bedrock. Memory: 256 MB.

- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `loy-ledger` and `loy-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph plain-English owner narrative (joins, rewards claimed, win-back results); emails it via SES to the owner. Memory: 512 MB.
- **`drive-sync`** — EventBridge Scheduler target, every 15 minutes. Uses the Google Drive API (service-account credentials in Secrets Manager under `loy/drive/sa`) to export the rules and voice docs as plain text and write to `s3://loy-rules-source/` only if they changed since the last sync. Also writes a daily backup of `loy-members` to S3. Memory: 256 MB. Timeout: 30 s.

Storage

- **DynamoDB · `loy-members`** — one row per member. PK `phone`; attributes: `name`, `email`, `opted_in`, `balance` (number), `tier`, `visit_count`, `last_seen`, `joined`, `status` (active/rested). On-demand. No TTL.
- **DynamoDB · `loy-ledger`** — one row per points change of any kind. PK `(phone, ts)`; attributes: `points_change` (signed), `reason` (earn/bonus-item/reward-redeem/win-back-bonus/reversed), `sale_id` or `reward`, `by_staff`. On-demand. No TTL — this is the long-term audit of every point.
- **DynamoDB · `loy-audit`** — one row per admin or lifecycle action (reverse, rest, bonus-grant, override). PK `(phone, ts)`; attributes: `action`, `by_user`, `before`, `after`. On-demand. No TTL.

- **DynamoDB · `loy-review`** — held-for-review sales awaiting a staff decision. PK `(phone, sale_id)`; attributes: `total`, `computed_points`, `flag_reason`, `status`. On-demand.
- **DynamoDB · `loy-lapsing`** — the current lapsing list. PK `phone`; attributes: `weeks_away`, `nudge_count`, `last_nudge`. On-demand.
- **S3 · `loy-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled, so a bad Drive edit rolls back in one click.
- **S3 · `loy-backups`** — daily export of `loy-members` and `loy-ledger`. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. One callsite: `summary` for the monthly owner narrative. Claude Sonnet 4.6 (`anthropic.claude-sonnet-4-6-20250930-v1:0`) is wired but unused — the summary is a short, low-stakes paragraph that Haiku handles well, so the heavier model isn't justified here.
- **Embeddings.** Not used. Balances are structured numbers; an exact lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The earn and redeem paths don't call Bedrock at all; `summary` fires once a month.

EventBridge Scheduler config

- `loy-nightly-balance` — `cron(0 19 * * ? *)` in the shop's timezone. Target: `nightly-balance` Lambda.
- `loy-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `loy-lapsing-sweep` — `cron(0 9 ? * MON *)` in TZ. Target: `lapsing-sweep` Lambda.
- `loy-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **Quiet hours** — `email-dispatch` enforces the configured window; events that arrive inside it are deferred with a one-off `at(...)` schedule that re-invokes dispatch at the next allowed minute, created with `--action-after-completion DELETE` so the rule self-cleans.

SES outbound

- Verify a sender identity at `rewards@your-shop.com` with DKIM and SPF on the parent domain. Out of the SES sandbox by request before launch.
- One configuration set with event publishing for bounces and complaints to an SNS topic; a small handler flips `opted_in = false` on a hard bounce or a complaint so the shop stops mailing bad or unhappy addresses.
- Every marketing-style email (nightly balance, win-back nudge) carries a one-tap unsubscribe link backed by a tiny Function URL path; the reward-earned email is transactional and always sends.

IAM (least privilege per Lambda)

Each Lambda has its own role with policies scoped to exact ARNs. Sketch:

- **earn role:** `s3:GetObject` on the rules key; `dynamodb:GetItem` + `UpdateItem` + `PutItem` on `loy-members`, `loy-ledger`, and `loy-review`; `events:PutEvents` on the default bus. *No* `bedrock:*`.
- **redeem-handler role:** `dynamodb:GetItem` + `UpdateItem` on `loy-members` (the conditional subtract); `dynamodb:PutItem` on `loy-ledger` and `loy-audit`; `secretsmanager:GetSecretValue` on the staff-session signing key. *No* `ses:*`, *no* `bedrock:*`.
- **email-dispatch role:** `s3:GetObject` on the voice key; `ses:SendEmail` from the verified sender identity; `dynamodb:GetItem` on `loy-members` for the opt-out check; `dynamodb:PutItem` on `loy-ledger`; `scheduler:CreateSchedule` for quiet-hours defers.
- **signup role:** `dynamodb:PutItem` + `UpdateItem` on `loy-members`; `ses:SendEmail` for the double opt-in confirm only.
- **drive-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on `loy-rules-source` and `loy-backups`; `dynamodb:Scan` on `loy-members` for the backup; outbound network to `www.googleapis.com`.

Concurrency and correctness

The one rule that matters most: a balance is never read-modified-written in application code. Every change is a conditional `UpdateItem` with an atomic `ADD` on the `balance` attribute, so two sales, or a sale and a redeem, applied in the same instant both land without losing each other. Redeems carry a

`ConditionExpression balance >= :cost`; if the condition fails, DynamoDB rejects the write and the handler returns "not enough points" rather than overdrawing. The same condition makes double-redeem impossible: the second identical subtract fails its condition. Idempotency keys on the POS webhook (the till's `sale_id`) make a retried sale a no-op, so a flaky network can't double-credit a customer.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"condition_failed"` to a metric for alerting.
- **Alarms:** earn-function errors > 0 in an hour (a dropped sale is a lost point); redeem condition-failures spiking (might mean a stale staff screen or a bug); SES bounce rate > 2% (protect sender reputation).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `loy-cost-alarm` subscribed to the owner's email.

Config and secrets

Service-account credentials for the Drive API live in Secrets Manager under `loy/drive/sa`. The POS shared secret, the staff-session signing key, and the SES configuration live under `loy/*`. The configured timezone, quiet-hours window, lapsing window, `max_nudges`, and the regular threshold all live in Parameter Store under `/loy/config/` — the operational knobs that don't belong

in the owner-editable rules doc. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC (no long-lived keys) and AWS SAM. The opinionated bits: turn on S3 versioning for `loy-rules-source` so a bad Drive edit can be rolled back in one click, version the EventBridge Scheduler timezone setting so you don't accidentally start running the nightly job in UTC after a CI rotation, and keep the POS webhook's shared secret out of the repo (it lives only in Secrets Manager, injected at deploy). Total deployable surface: around nine Lambdas, five DDB tables, two S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SES configuration set, and one Budgets alarm.

That's the full system. Six narrative posts and this engineering reference. If you want to talk about adapting it for your business, see Work with me.