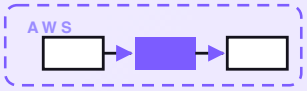
7-PART SERIES · FREE COMPANION



Feedback collector

A serverless system that asks every customer how it went, at the right moment after a visit or purchase, with one simple question. It collects the answer and routes by it: happy customers get a gentle nudge to leave a public review; unhappy ones go straight to the owner privately so the problem gets fixed before it becomes a bad review. Quiet hours respected; it asks once and never nags. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Feedback collector

PART 1 OF 7

JUNE 10, 2026 PART 1 OF 7 · FEEDBACK COLLECTOR SERIES ~5 MIN READ

A feedback collector on AWS for a few dollars a month

A happy customer who walks out the door is worth a public review — if you ask at the right moment. An unhappy one is a bad review waiting to happen — unless you hear about it first and fix it privately. Most small businesses do neither: they never ask, or they blast everyone the same generic “rate us” email and end up sending their grumpiest customers straight to a one-star page. This post walks through the design of a small system that asks each customer one simple question at the right time, reads the answer, and sends the happy ones toward a public review and the unhappy ones straight to the owner.

KEY TAKEAWAYS

- Three sources for customers to ask: a Drive list, a point-of-sale webhook lane, and a booking import lane.
- Every reply lands in one of three buckets: happy, unhappy, or unclear.
- Happy goes to a public-review nudge; unhappy goes to the owner privately; unclear waits for a human.
- Asks respect quiet hours and your holiday calendar. It asks once and never nags.
- Designed on AWS for about $2/month at typical small-business volume.

The whole system on one page

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

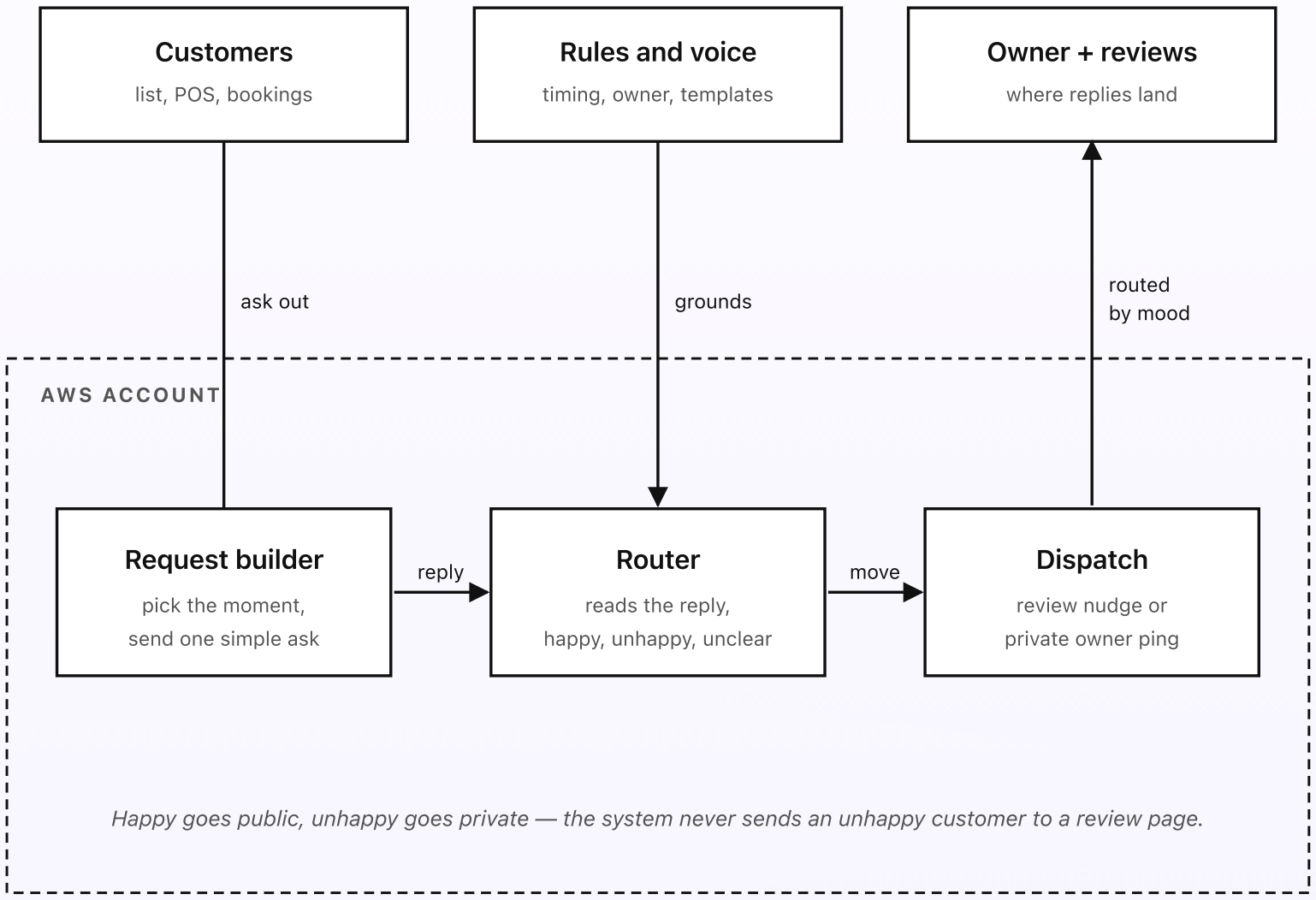

Customers
list, POS, bookings
Rules and voice
timing, owner, templates
Owner + reviews
where replies land
ask out
grounds
routed
by mood
AWS ACCOUNT
Request builder
pick the moment,
send one simple ask
reply
Router
reads the reply,
happy, unhappy, unclear
move
Dispatch
review nudge or
private owner ping
Happy goes public, unhappy goes private — the system never sends an unhappy customer to a review page.

Fig 1. Three sources outside, three pieces inside AWS. Customers flow in from a Drive list, a point-of-sale webhook lane, and a booking import lane. The Router reads each reply and picks one of three buckets. Dispatch sends the right move to the right place.

What you set up once (the outside)

- **Customers.** A Google Sheet in a Drive folder, one row per finished visit or purchase: customer name, contact (email or mobile), what they bought or booked, the type of visit (sit-down meal, takeaway, delivered product, service appointment), the date it happened, and the staff member who served them. You can fill it in by hand, but in real life it fills itself from two other lanes covered in Part 2 — a point-of-sale webhook lane (your till or checkout tool posts a row the moment a sale closes) and a booking import lane (finished appointments get pulled in from your calendar each hour).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the timing for each visit type — how long to wait before asking. A sit-down meal might ask two hours later, while the coffee's still warm in memory; a delivered product waits three days so it has actually arrived; a service appointment asks the next morning. The doc also names the owner who gets unhappy replies, the quiet hours, any holidays to skip, and the public-review link a happy nudge should point to. The *voice* doc holds the ask message and the one gentle follow-up — what the email or text actually says.
- **Owner and review links.** The owner is the person who personally handles an unhappy customer — usually you. The review link is your public Google, Facebook, or industry-site page. Happy nudges point there; unhappy replies never do. They go to the owner with the customer's name, what they bought, their exact words, and a one-tap way to call or message them back.

What runs after every visit (the inside)

- **The request builder.** Three sources feed the customer list. The Drive sheet itself is the canonical store. New rows also arrive via the point-of-sale webhook lane (your checkout tool calls a small endpoint when a sale closes) and the booking import lane (finished appointments get pulled hourly by a small sync Lambda). For each new finished visit, the builder reads the rules doc, works out the right moment to ask, and schedules one simple question to go out then — not now, not three times, just once at the right time.
- **The router.** When a reply comes back, the router reads it. A one-tap star rating is read by plain Python: four or five stars is happy, one or two is unhappy, three is unclear. If the customer wrote a few words instead of tapping, a quick Bedrock Haiku 4.5 read decides the mood from the words. Either way the reply lands in exactly one of three buckets: happy, unhappy, or unclear. The router writes the result to a small table so the same customer is never asked or routed twice in one cycle.
- **Dispatch.** Reads the voice doc and acts on the bucket. *Happy*: send a short, warm note thanking them and offering a one-tap link to leave a public review. *Unhappy*: skip the review entirely and ping the owner privately, right away, with everything they need to make it right. *Unclear*: hold it for a human to glance at in the daily sweep. Every move honors quiet hours and the holiday calendar, and every reply and routing decision is logged. A monthly summary writes a short owner-ready paragraph: how many asked, how many happy, how many unhappy, how many turned into reviews.

In plain words

Dwi finishes dinner at your restaurant at 8pm on a Friday. Your till closes her bill and posts a row. The rules doc says sit-down meals ask two hours later, but two hours later is 10pm — inside quiet hours — so the ask waits until 9am the next morning. At 9am Dwi gets a short text: “Hi Dwi, thanks for dining with us last night! How was it?” with five tappable stars. She taps five. The system reads “happy,” sends a warm follow-up — “So glad! Would you mind leaving a quick review? *[one-tap link]*” — and you get a new five-star review by lunchtime. Meanwhile Budi, who waited 40 minutes for his order the same night, taps two stars and types “food was cold.” He is never shown the review link. Instead your phone buzzes at 9:01am: “Budi — 2 stars — ‘food was cold’ — table 12 last night — *[call] [text]*.” You call, apologize, offer him a free dessert next time. The one-star review that almost happened becomes a regular instead.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is every happy customer who never left the review that would have brought in the next ten, and every unhappy one who skipped you and told the internet instead.

DESIGN RULES THAT SHAPED EVERY DECISION

- Ask at the right moment — timing is in the rules doc, per visit type. A badly timed ask is a wasted one.
- Three buckets, always. Happy, unhappy, unclear. There is no fourth.
- Unhappy never sees a review link. The whole point is to catch it before it becomes a public one.
- Ask once. At most one gentle follow-up, then stop. Quiet hours and holidays are respected.
- The customer list lives in Drive. Adding a customer or changing the timing doesn't need a deploy.
- Every reply and routing decision is logged. You can audit any month's feedback in one place.

Why this shape

Most businesses handle feedback in one of three ways: they never ask, they ask everyone the same way at the wrong time, or they ask only the customers they already know are happy. The first leaves money on the table — happy customers will leave reviews, but only if you ask, and only if asking is easy. The second is worse than nothing: a generic "rate us" blast sends your unhappy customers straight to the public page where they do the most damage. And cherry-picking only the happy ones is both dishonest and a missed chance — the unhappy customer you avoid asking is the one whose problem you could have fixed.

The setup above splits the job in two on purpose. Ask everyone — but read the answer before you decide what to do with it. A good answer earns a public ask, at the moment it's easiest to say yes. A bad answer earns a private, immediate heads-up to the one person who can fix it. The system is invisible to a happy customer (one tap, one thank-you, done) and a quiet early-warning radar to the owner. It never nags, never asks twice, and never points a frustrated customer at the page where frustration costs the most.

The next four posts walk through each piece in turn: how a feedback request goes out, how a reply comes back, how feedback gets routed by mood, and how a review ask or a private fix actually happens. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 10, 2026 PART 2 OF 7 · FEEDBACK COLLECTOR SERIES ~4 MIN READ

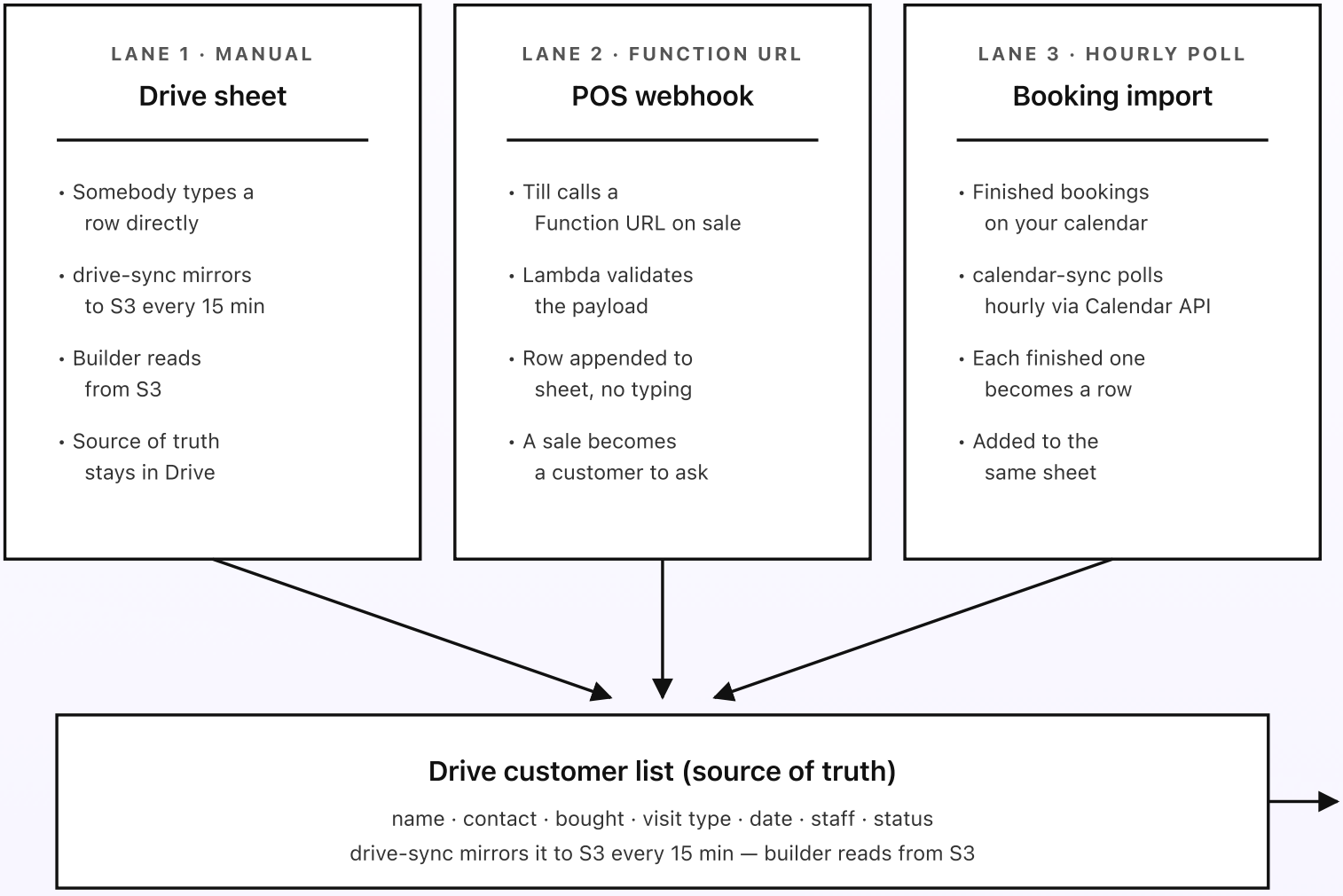
How a feedback request goes out

The system can only ask the people it knows about. So the first job is making sure the customer list actually reflects who walked in and who walked out happy or not. There are three ways a finished visit gets onto the list: somebody types a row in the Drive sheet, your till posts one the moment a sale closes, or a finished appointment gets pulled in from your booking calendar. Once a visit is on the list, a timing rule picks the right moment — and then it asks once.

KEY TAKEAWAYS

- Three intake lanes feed one list: the Drive sheet, a point-of-sale webhook, and a booking import.
- The point-of-sale lane posts a row to a Function URL the moment a sale closes — no typing.
- Finished appointments tagged in your calendar get pulled hourly via the Google Calendar API.
- A timing rule per visit type picks the right send moment — meal two hours later, product three days.
- The list stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one list



The Drive sheet stays the source of truth — the other lanes are conveniences that add rows to it.

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the POS webhook and the booking import are conveniences that add rows. The drive-sync Lambda mirrors the sheet to S3 so the builder can read it without hitting Drive on every run.

Lane 1: the Drive sheet itself

The simplest lane. Open the customer list in Drive, add a row, save. The columns are short: name, contact (email or mobile), what they bought or booked, visit type, the date it happened, and the staff member who served them. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://fc-customers-source/customers.csv` if the sheet has changed since the last sync. The builder reads from S3, not Drive directly. That keeps Drive API calls predictable and gives you S3 versioning for free, so a bad bulk-edit can be rolled back in one click.

This lane covers the cases where you don't have a till that talks to anything — a market stall, a one-off job, a referral you want to follow up. Most businesses use it for the first week and then lean on the other two.

Lane 2: the point-of-sale webhook (the lane most teams actually use)

Set up a small `intake-webhook` Lambda with a Function URL — a plain web address your till or online checkout can call. Most modern point-of-sale and e-commerce tools (Square, Shopify, your booking platform) can fire a webhook when a sale closes. Point it at the Function URL. When a sale completes, the till posts a small bundle of data: who bought, what they bought, when, and which staff member rang it up. The Lambda checks the payload is genuine (a shared

secret in the header, verified against Secrets Manager), drops anything that's a refund or a test, and appends a clean row to the Drive sheet via the Sheets API.

The point of this lane is that nobody has to remember to type anything. The moment Dwi pays for dinner, she's on the list, and the timing clock starts. There's no batch upload at the end of the day, no "oh, I forgot to add the Tuesday customers." A finished sale becomes a finished-visit row in seconds.

One guardrail: the webhook never asks immediately. It only adds the row. The actual ask is scheduled for the right moment, which is the timing rule below. That keeps the "don't pester someone who's still standing at your counter" rule firmly in one place.

Lane 3: booking import

Some businesses run on appointments, not walk-in sales — a salon, a clinic, a repair shop, a consultant. For them the "finished visit" isn't a till transaction, it's an appointment that has happened. A small `calendar-sync` Lambda runs hourly, looks at the configured Google Calendars (using a service-account credential stored in Secrets Manager), and pulls any appointment whose end time has just passed and which is marked as completed (the simplest convention is a `#done` tag the staff add when the client leaves, or a calendar whose events are confirmed bookings). Each finished appointment becomes a row in the same customer list, with the visit type set to whatever the booking was for.

Booking import is the most opt-in of the three lanes. A shop that has no appointments loses nothing by skipping it; a clinic that lives on its calendar avoids retyping every visit by hand.

The timing rule: ask once, at the right moment

A row landing on the list does not send an ask. It schedules one. The request builder reads the visit type and looks up the wait time in the rules doc: “sit-down meal: ask 2 hours later. Takeaway: ask 1 hour later. Delivered product: ask 3 days later. Service appointment: ask the next morning.” The right moment is different for each because the question only works once the experience is fresh and complete — asking about a delivered product the second the order is placed is useless; the box hasn’t even arrived.

The builder computes the send time, checks it against quiet hours (default 8pm to 9am) and the holiday list, nudges it to the next allowed minute if needed, and creates a one-off EventBridge Scheduler rule that fires the ask at exactly that time. One rule, one ask. If the customer doesn’t reply, a daily sweep sends at most one gentle follow-up a few days later, then marks the row done for good. A customer who already replied is never asked again. The next post covers what happens when a reply actually comes back.

PART 3 OF 7

JUNE 10, 2026 PART 3 OF 7 · FEEDBACK COLLECTOR SERIES ~5 MIN READ

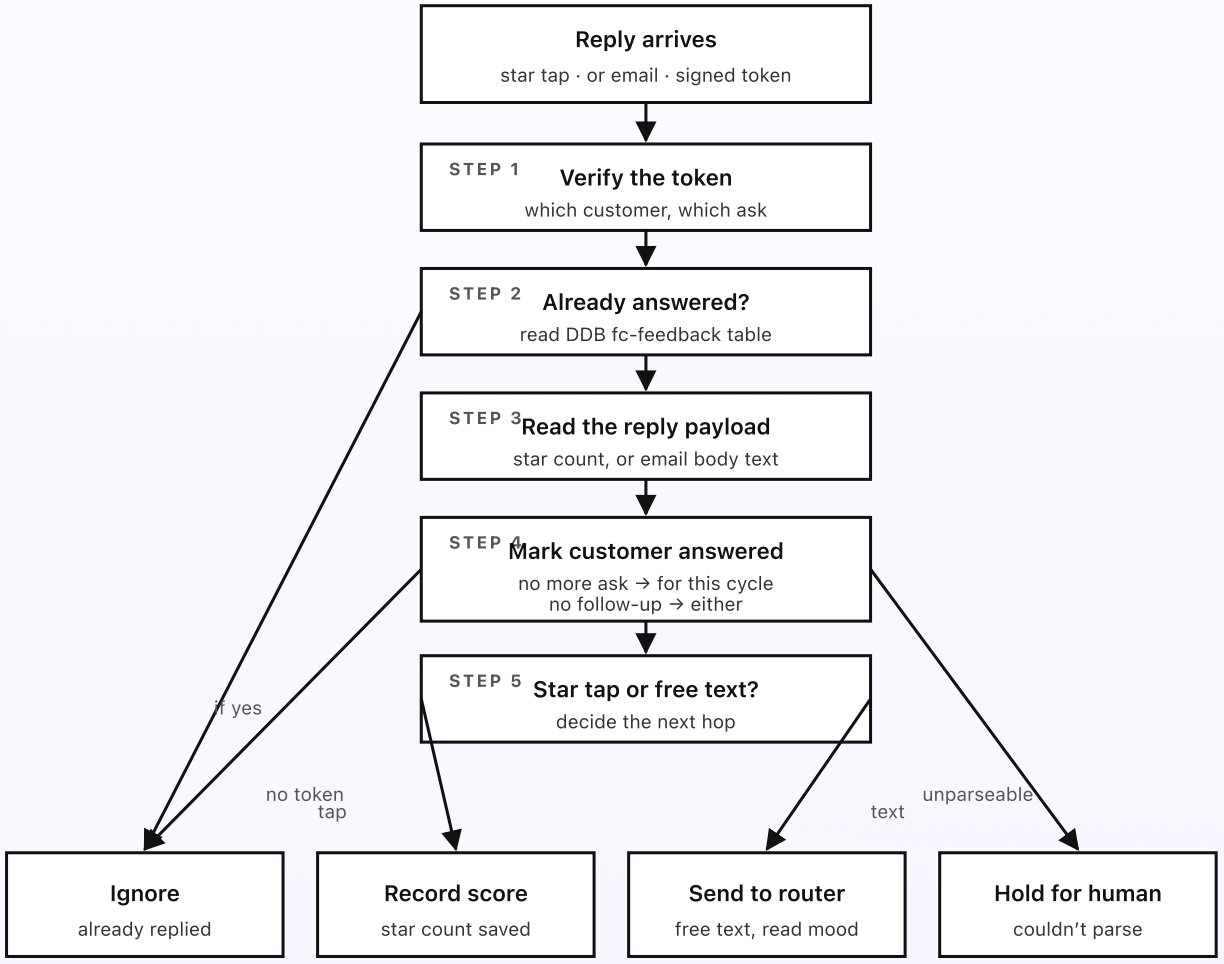
How a feedback reply comes back

The ask went out. Now the customer either taps a star or types a few words — or does nothing. A tapped star arrives as a click on a one-tap link, which hits a Lambda Function URL. A written reply arrives as an email, which SES inbound catches. Either way, the same small Lambda records the reply exactly once, marks the customer as answered so they're never asked again this cycle, and hands the reply to the router. The whole path is plain Python; no model is needed just to receive a reply.

KEY TAKEAWAYS

- A star tap is a one-tap link click that hits a Lambda Function URL — no app, no login.
- A written reply arrives by email and is caught by SES inbound.
- Each reply carries a signed token, so the system knows exactly which ask it answers.
- DynamoDB records the reply once and marks the customer answered, so no ask goes out twice.
- Receiving a reply uses no model. The mood read in the next post is the only AI step.

The reply flow, per customer



A reply is recorded exactly once — the token ties every reply to one ask and one customer.

Fig 3. The reply flow, per customer. Five steps verify, dedupe, record, and route. A star tap goes straight to a recorded score; free text goes to the router for a mood read; anything unparseable waits for a human.

The one-tap star link

The ask message ends with five stars, and each star is a plain web link. There's no app to install and no login — tapping a star simply opens a link. The link points at a Lambda Function URL and carries two things in the address: how many stars the customer tapped, and a signed token. The token is a short string the system generated when it sent the ask, signed with a secret so it can't be faked or guessed. It names exactly which customer and which ask this tap answers.

When a star link is tapped, the `reply-handler` Lambda runs. It checks the signature on the token (a tampered or stale token is rejected and logged), looks up the customer, and reads the star count straight from the link. Five stars and four stars are happy; one and two are unhappy; three is unclear. The handler shows the customer a simple thank-you page, and — for a happy tap — can include the review link right there, so a five-star tapper sees the public-review button immediately while their goodwill is highest. The routing details are Part 5; the point here is the tap is received, verified, and recorded in one quick step.

The written reply

Some customers don't tap. They hit reply on the email and type a sentence: "loved it, the staff were lovely," or "waited far too long and the coffee was cold." That reply lands in a dedicated inbox — `feedback@your-company.com` — handled

by Amazon SES inbound. SES writes the raw message to S3, which triggers the same `reply-handler` Lambda. The Lambda finds the signed token in the email's reply headers (it was tucked into the address the ask was sent from), identifies the customer, and pulls out the body text.

A star tap is unambiguous — a number is a number. A written reply isn't; "it was fine I suppose" could be a quiet five or a polite two. So a written reply doesn't get scored by simple rules. It gets handed to the router, which reads the mood of the words. That step — the one place the system uses a model — is the whole of the next post.

Recorded once, asked once

Both paths converge on one rule: a reply is recorded exactly once, and a customer who has replied is never asked again this cycle. The moment a reply is verified, the handler writes a row to the `fc-feedback` DynamoDB table keyed by the customer and the ask. If a second reply arrives for the same ask — the customer tapped a star and then also wrote an email, or double-tapped — the "already answered?" check catches it and the later reply is ignored (the first one stands; the duplicate is logged for the record).

The same row is what stops the follow-up. The daily sweep that sends gentle reminders skips any customer whose row shows a reply. So the unbreakable promise from Part 1 — ask once, never nag — is enforced by a single row in a single table, written the instant a reply comes in.

Next post: how the router reads a reply — the star rule for taps, the quick Bedrock mood read for free text — and turns every reply into happy, unhappy, or unclear.

PART 4 OF 7

JUNE 10, 2026 PART 4 OF 7 · FEEDBACK COLLECTOR SERIES ~5 MIN READ

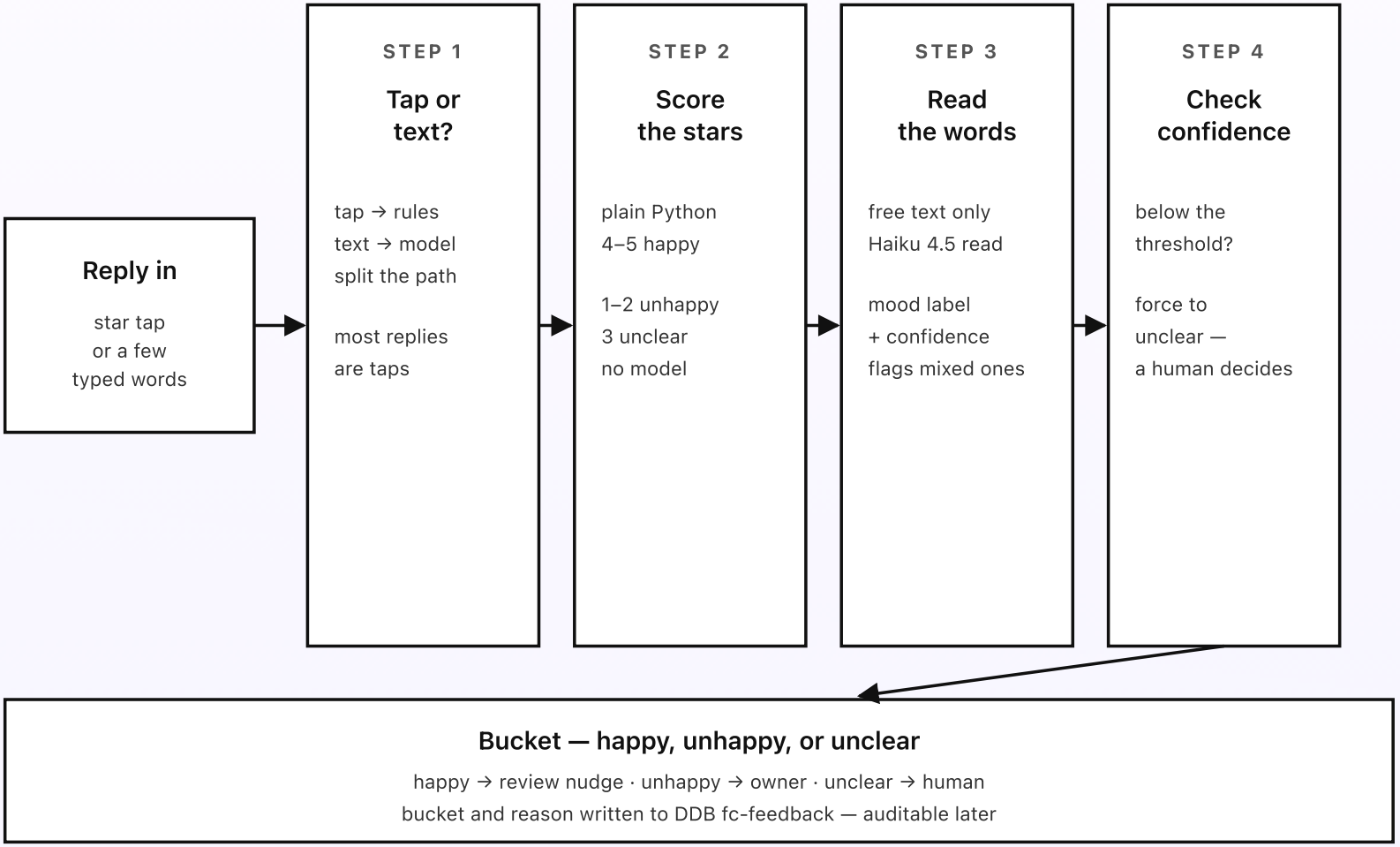
How feedback gets routed by mood

A reply came back — a tapped star or a few typed words. Now the router has to turn it into one of three buckets: happy, unhappy, or unclear. Get this wrong and you nudge a frustrated customer toward a public review, or you bury a delighted one who would have left you five stars. A star tap is settled by plain rules. Free text gets a quick mood read from a model. Four small steps sit between the raw reply and the bucket.

KEY TAKEAWAYS

- A star tap is read by plain Python: 4–5 is happy, 1–2 is unhappy, 3 is unclear. No model.
- Free text gets one short Bedrock Haiku 4.5 read that returns a mood and a confidence score.
- A low-confidence read or a mixed reply is routed to unclear, not guessed.
- The bucket and the reason are written to the log, so any routing decision can be checked later.
- Every gate is a deterministic check or one cheap model call — no surprises.

Four steps to a bucket



A mixed or low-confidence reply becomes unclear, never a guess — the system would rather ask a human.

Fig 4. Four steps between a raw reply and a bucket. Split tap from text. Score the stars by rule. Read the words with one cheap model call. Check confidence and force the doubtful ones to unclear. Then write the bucket and the reason to the log.

Step 1: tap or text

The first thing the router checks is what kind of reply it's holding. A star tap arrives as a number from one to five — clean, unambiguous, no reading required. Free text arrives as a sentence or two. The two need completely different handling, so the router splits the path right away. In practice most replies are taps, because tapping a star is the easiest thing in the world and the ask is built to make it a single touch. That matters for cost: the cheap, no-model path handles the bulk of the traffic, and the model only runs on the minority who actually write something.

Step 2: score the stars

For a tap, the rule is fixed and lives in the rules doc so you can change it without a deploy: four or five stars is *happy*, one or two is *unhappy*, three is *unclear*. That's it. A three is genuinely ambiguous — it's the customer who shrugged — so rather than guess, the system treats it as unclear and lets a human glance at it. There is no model call on this path; a star is a number and a number maps to a bucket. The vast majority of replies settle here, in microseconds, for no model cost at all.

Why put the thresholds in the doc? Because different businesses feel differently about a four. A fine-dining restaurant might decide a four isn't a public-review-worthy "wow" and route fours to unclear; a busy takeaway might be thrilled with

any four or five. Moving the line is a one-word edit in the rules doc, not a code change.

Step 3: read the words

Free text is where a model earns its place. “It was lovely, thank you” is clearly happy. “Waited 40 minutes and the food was cold” is clearly unhappy. But plenty of replies are slippery — “it was fine,” “could’ve been better,” “the food was great but the service was slow.” Simple keyword rules fall over on these. So for free text only, the router makes one short call to Bedrock Haiku 4.5.

The prompt is tight: “Read this customer’s reply. Return JSON only: a mood of happy, unhappy, or unclear, and a confidence from 0 to 1. If the reply is mixed, sarcastic, or too short to be sure, return unclear. Do not invent praise or complaints that aren’t there.” The reply text is a sentence or two, so the call is a few hundred tokens in and a tiny bit out — a fraction of a cent. Haiku is the right model here: it’s the cheap, fast path, and reading the mood of one short sentence is well within what it does reliably. The heavier Sonnet model is reserved for jobs that actually need deeper reasoning, which this isn’t.

Step 4: check confidence, then commit

The model returns a mood and a confidence. The last step is a guardrail: if the confidence is below the threshold set in the rules doc (default 0.7), the reply is forced to *unclear* regardless of which way the model leaned. The same goes for a three-star tap from Step 2. The reason is the asymmetry of the mistake. Routing an unhappy customer to a public review page is the one outcome the whole

system exists to prevent; nudging an ambiguous reply to a human costs a few seconds of someone's attention. When in doubt, the system spends the human's seconds, not the customer's goodwill.

Once the bucket is settled, the router writes it to the `fc-feedback` row along with the reason — "5-star tap," "model: unhappy 0.92," "model: unclear 0.55 → forced unclear." That line is what makes any routing decision auditable later: if a customer ever says "I left you good feedback and you ignored me," you can see exactly what came in and where it went.

Next post: what each bucket actually triggers — the gentle one-tap review nudge for happy, the immediate private ping to the owner for unhappy, and the quiet hold for unclear.

PART 5 OF 7

JUNE 10, 2026 PART 5 OF 7 · FEEDBACK COLLECTOR SERIES ~5 MIN READ

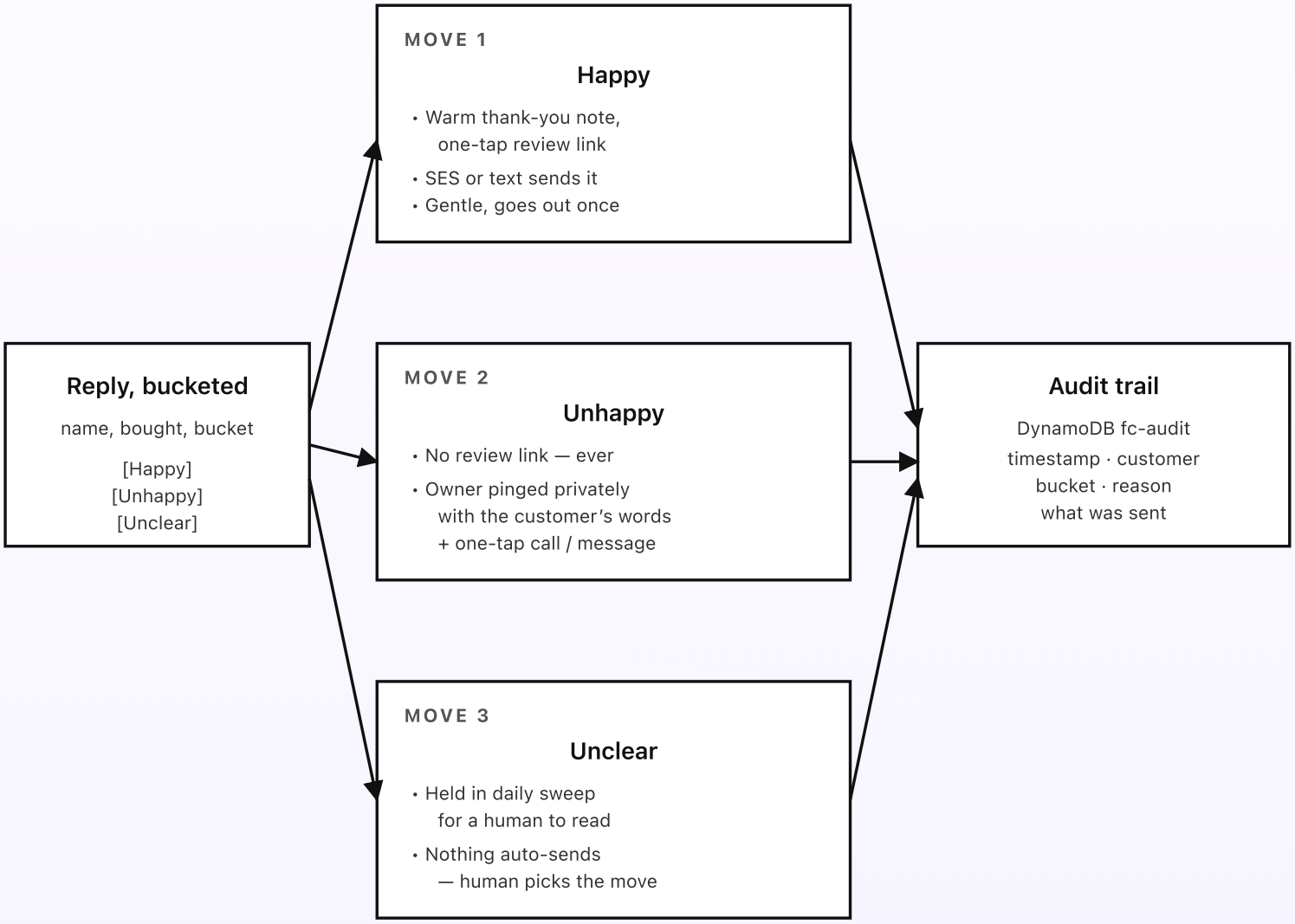
How a review ask or a private fix happens

The router settled on a bucket. Now the dispatch piece acts on it. A happy customer gets a warm, one-tap nudge toward your public review page. An unhappy one is never shown that page — instead the owner gets pinged privately, right away, with everything needed to make it right. An unclear reply waits quietly for a human to glance at. This post walks through all three moves — and how each one is recorded so nothing slips.

KEY TAKEAWAYS

- Three moves from the bucket: *happy* (review nudge), *unhappy* (private owner ping), *unclear* (hold for human).
- The happy nudge is a short note with a one-tap link straight to your public review page.
- The unhappy ping goes to the owner alone — never to a public page — with the customer's words and a way to reach back.
- Unclear replies pile into a daily sweep for a human to read; nothing auto-sends.
- Every move writes a row to the audit trail with the bucket, the reason, and what was sent.

Three moves from the bucket



An unhappy customer is never shown a review link — catch the problem before it becomes a public review.

Fig 5. Three moves, three different effects. Happy gets a one-tap review nudge. Unhappy gets a private, immediate owner ping. Unclear waits for a human. Every move writes to the audit trail.

Move 1: the review nudge (the one that grows your reputation)

Dwi tapped five stars. The dispatch piece reads the voice doc, fills in her name, and sends a short, warm note: “So glad you enjoyed it, Dwi! If you have ten seconds, a quick public review really helps a small place like ours — *[leave a review]*.” The link goes straight to your public Google, Facebook, or industry-site page, named in the rules doc. On a phone, it’s one tap from the thank-you to the review box.

Two design choices keep this from feeling like spam. First, it only goes to genuinely happy customers, so you’re never begging a lukewarm person for a favor. Second, it goes out exactly once. If Dwi doesn’t leave a review, that’s the end of it — no “just checking in!” chase. The nudge is an open door, not a foot in it. And because a five-star tap already showed its goodwill the moment it landed, the nudge can ride on that: it’s often shown on the very thank-you page the tap opened, while the customer is still right there.

Move 2: the private fix (the one that protects your reputation)

Budi tapped two stars and wrote “food was cold.” He is never, under any circumstances, shown the review link. Instead the dispatch piece pings the owner

privately and immediately. The message has everything needed to act: "Budi — 2 stars — 'food was cold' — dinner, table 12, last night, served by Sari — *[call Budi] [text Budi]*." The owner taps call, apologizes, and offers to make it right. The contact lands wherever the owner actually watches — a private Slack DM, a text, or an email, set in the rules doc.

This is the heart of the whole system. A frustrated customer who hears nothing tells the internet. A frustrated customer whose owner calls within the hour, listens, and fixes it usually becomes a regular — and often leaves a good review later, unprompted, because the recovery impressed them more than a smooth visit would have. The system's job is to make sure that call can happen, by getting the complaint to the one person who can act while the problem is still small, fresh, and private.

The owner ping is not auto-resolved. The system doesn't pretend to fix anything itself — it just makes sure the right human knows, fast. The owner can mark the item handled from the ping, which writes the outcome to the log, but the actual making-it-right is a human talking to a human. That's the point.

Move 3: the quiet hold (the one that never guesses)

A three-star tap, or a written reply the model wasn't sure about, lands in *unclear*. Nothing is sent. Instead the reply waits in a daily sweep — a short list the owner or a staff member skims once a day. Each unclear item shows the customer, what they bought, and their exact reply, with three buttons: *treat as happy* (send the review nudge), *treat as unhappy* (open the private fix), or *done* (no action

needed). A human spends two seconds per item and the doubtful replies get a real decision instead of a coin-flip.

Holding unclear replies for a human is the safety valve that lets the rest of the system be confident. Because the doubtful cases have somewhere safe to go, the happy and unhappy paths can act automatically without fear of misrouting — anything they're unsure about simply isn't theirs to handle.

Every move is logged

The `fc-audit` table records every move with the customer, the bucket, the reason the router chose it, and exactly what was sent — the review nudge, the owner ping, or the hold. That trail answers the questions that come up later: how many five-star taps turned into actual public reviews this month, which complaints reached the owner and how fast, whether any unclear items are sitting unread. The monthly summary in the next post reads straight from this table to write the owner a short paragraph — how many asked, how many happy, how many fixed before they went public.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the model barely registers on the bill.

PART 6 OF 7

JUNE 10, 2026 PART 6 OF 7 · FEEDBACK COLLECTOR SERIES ~3 MIN READ

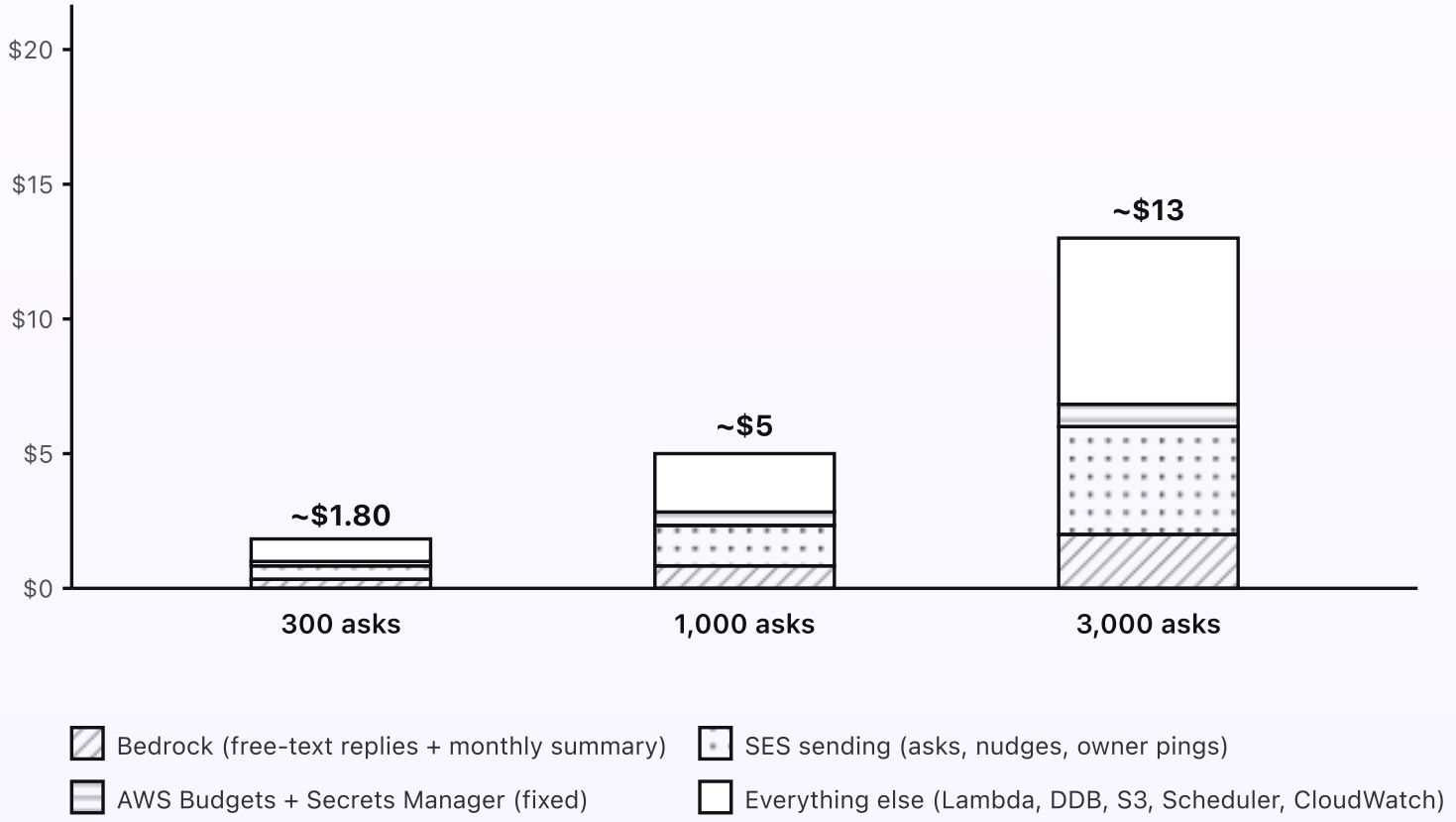
What the feedback collector costs

The feedback collector is one of the cheapest systems in this whole series. It sends a short message per finished visit, reads the star taps with plain date-and-number logic, writes a few rows to DynamoDB, and pings the owner only when something's wrong. It calls no model to receive a reply or to score a star. Bedrock fires only on the minority of replies that are free text, and once a month for the owner summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $1.80/month at typical SMB volume (around 300 asks a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- Sending the asks and reading the taps costs pennies — no model calls on that path.
- Bedrock fires only on free-text replies (a minority) and the monthly summary.
- At 1,000 asks the bill is around $5. At 3,000 asks it's around $13.

Cost at three volumes



Sending the asks is the dominant cost — and even that is a fraction of a cent per ask.

Fig 6. Monthly cost at three ask volumes. Bedrock is a small sliver because it only fires on the free-text minority of replies and the monthly summary. The dominant cost is the everything-else bucket plus SES sending: one message per finished visit.

Where the dollars actually go

Lambda runtime (the bulk of "everything else"). A handful of small functions do all the work: the request builder that schedules each ask, the reply handler that catches taps and emails, the router that buckets each reply, the dispatch piece that sends the right move, plus drive-sync every fifteen minutes and calendar-sync every hour. Each run is a few hundred milliseconds. Even at 3,000 asks a month the Lambda total lands under a dollar — there's no always-on cost, just a tiny burst per event.

DynamoDB on-demand. Three small tables: `fc-feedback`, `fc-state`, `fc-audit`. One write when an ask goes out, one when a reply lands, one when a move fires. Reads during the daily sweep. Pennies a month at any of these volumes.

S3 + Storage. The mirrored customer-list CSV plus the raw inbound emails from written replies. A few hundred KB total at SMB volume. Effectively free.

EventBridge Scheduler. One one-off rule per ask (it fires at the right moment, then self-deletes), plus the recurring drive-sync, calendar-sync, daily sweep, and monthly summary. A few hundred invocations a month. Pennies.

SES (the asks and the replies). Outbound carries the asks, the happy nudges, and the email owner pings: $0.10 per thousand sent. At 300 asks plus a few hundred nudges and pings, that's a few cents a month; at 3,000 it's under a dollar. Inbound catches the written replies: $0.10 per thousand received, so cents. (If you send the ask as a text instead of an email, that part moves to your SMS provider's per-message price, which is usually the single biggest line — budget for it separately.)

Bedrock (only on free text). Reading a star tap uses no Bedrock. The model fires only when a customer writes words instead of tapping — a minority of replies — and the call is tiny: a sentence in, a small JSON mood-and-confidence out, a fraction of a cent each. The monthly summary is one slightly larger call to write the owner paragraph: a couple of cents. Even at 3,000 asks, Bedrock is a small sliver of the bill.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the star-tap and webhook endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The system only runs when a visit finishes or a reply lands.
- **A Knowledge Base.** Reading the mood of one short reply needs no document search — one cheap model call does it. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **A model on the common path.** Star taps are scored by plain Python. Bedrock fires only on free text and the monthly summary.

How the cost scales

Lambda runtime, DynamoDB, and SES sending all grow roughly linearly with the number of asks, because each finished visit gets one ask and at most one reply to handle. Bedrock grows only with the share of customers who write free text instead of tapping — usually a minority — so it stays a small sliver even as volume

climbs. So the bill at 10,000 asks a month is around $40, and most of that is the SMS or email you're sending anyway. Past those volumes the shape doesn't change; you're just sending more messages.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The collector's normal-volume 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 rule set, and EventBridge Scheduler config.

PART 7 OF 7

JUNE 10, 2026 PART 7 OF 7 · FEEDBACK COLLECTOR SERIES ~8 MIN READ

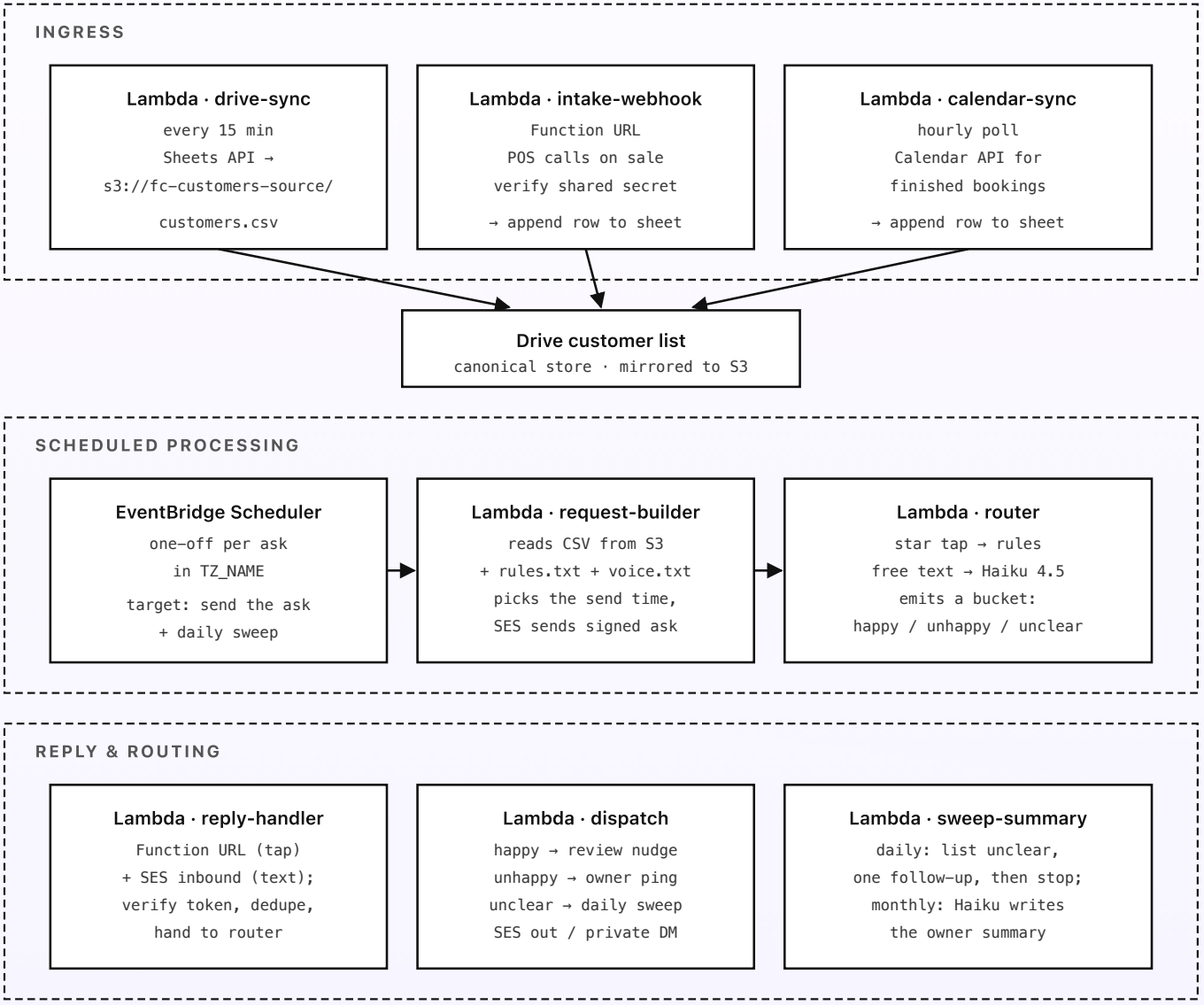
Engineering reference: the feedback collector architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, EventBridge Scheduler config, the DynamoDB schemas, and the star-tap and webhook flow. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound, 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 an SMB is a feedback ask that goes out a few hours late, not a regional outage. One AWS account dedicated to the collector (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. Deploy with GitHub Actions using OIDC and AWS SAM, so there are no long-lived AWS keys anywhere — the CI role is assumed per run.

Topology



Happy goes public, unhappy goes private — and every interaction is logged to fc-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the customer list), scheduled processing (the builder sending asks, the router bucketing replies), reply and routing (the handler receives, dispatch acts, the sweep cleans up). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

All Lambdas use the `arm64` (Graviton) 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.

- **`drive-sync`** — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `fc/drive/sa`) to export the customer-list sheet as CSV and write to `s3://fc-customers-source/customers.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://fc-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`intake-webhook`** — Lambda Function URL, public with `AuthType: NONE`; verifies a shared HMAC secret (`fc/pos/webhook-secret` in Secrets Manager) on each request. Called by the point-of-sale or e-commerce tool when a sale closes. Validates and normalizes the payload, drops refunds and test transactions, and appends a customer-list row via the Sheets API. Memory: 256 MB. Timeout: 15 s.
- **`calendar-sync`** — EventBridge Scheduler target, hourly. Uses the Google Calendar API `events.list` to scan configured calendars for bookings whose

end time has just passed and which are confirmed/done; appends a row per finished booking. For lower-latency setups you can switch to `events.watch` and have Calendar push notifications to a Function URL instead of polling, at the cost of refreshing the channel before its TTL expires. Memory: 256 MB. Timeout: 30 s.

- **`request-builder`** — triggered when a new row appears (a short EventBridge Scheduler poll on the mirrored CSV, or invoked directly by `intake-webhook` / `calendar-sync`). Reads `rules.txt` for the per-visit-type wait, computes the send time, applies quiet hours and the holiday calendar, and creates a one-off EventBridge Scheduler rule (`at(...)` with `--action-after-completion DELETE`) that invokes the send. The ask is rendered from `voice.txt` and sent via SES outbound; each star link and the reply-to address carry a signed token (HMAC over `customer_id|ask_id|exp`). Writes a pending row to `fc-state`. Memory: 256 MB. Timeout: 30 s. *No Bedrock calls.*
- **`reply-handler`** — two triggers: a Lambda Function URL for star-tap link clicks, and an S3 PUT on `s3://fc-raw-mime/` when SES inbound writes a written reply. Verifies the signed token, dedupes against `fc-feedback` (first reply wins), marks the customer answered, and forwards the payload to `router`. Renders a thank-you page for taps; for a happy tap it can show the review link inline. Memory: 256 MB. Timeout: 15 s. *No Bedrock calls.*
- **`router`** — invoked by `reply-handler`. A star tap is scored by plain rules from `rules.txt` (4–5 happy, 1–2 unhappy, 3 unclear). Free text gets one Bedrock Haiku 4.5 call (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) returning `{mood, confidence}`; confidence below the rules-doc threshold is forced to unclear.

Writes the bucket and reason to `fc-feedback` and emits the bucket to `dispatch`. Memory: 256 MB. Timeout: 30 s.

- **`dispatch`** — invoked by `router`. *Happy*: render the review nudge from `voice.txt` and send via SES outbound (or the SMS provider) with the public-review link. *Unhappy*: send a private owner ping (Slack DM via `chat.postMessage`, or SES email) with the customer's words and one-tap call/message links — never the review link. *Unclear*: enqueue for the daily sweep. Writes a row to `fc-audit` after each send. Memory: 256 MB. Timeout: 30 s.
- **`sweep-summary`** — EventBridge Scheduler target. Daily run: lists unclear items for a human, and for asks with no reply past the rules-doc window, sends at most one gentle follow-up via `request-builder`, then marks the row closed. Monthly run (first Monday 9am): reads the past month's `fc-feedback` and `fc-audit`, calls Bedrock Haiku 4.5 to write a one-paragraph owner summary (asked, happy, unhappy, reviews earned, complaints fixed before they went public), emails it via SES. Memory: 512 MB.

Storage

- **DynamoDB · `fc-feedback`** — one row per reply. PK `(customer_id, ask_id)`; attributes: `received_at`, `channel` (tap/email), `raw` (star count or text), `bucket` (happy/unhappy/unclear), `reason`. On-demand. The dedupe and answered-once guarantees both key off this table.
- **DynamoDB · `fc-state`** — one row per pending ask. PK `(customer_id, ask_id)`; attributes: `visit_type`, `send_at`, `sent_at`, `follow_up_sent`,

`status` (pending/sent/answered/closed), `schedule_arn` of the one-off rule. On-demand.

- **DynamoDB · `fc-audit`** — one row per move of any kind. PK `(customer_id, ts)`; attributes: `bucket`, `move` (review_nudge/owner_ping/held), `sent_via`, `by_user` (for human sweep actions). On-demand. No TTL — this is the long-term audit trail the monthly summary reads.
- **S3 · `fc-customers-source`** — mirrored CSV from the Drive customer list. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 2 years.
- **S3 · `fc-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3 · `fc-raw-mime`** — raw inbound MIME from written replies. Lifecycle to Glacier at 30 days; expiry at 2 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`. Two callsites: `router` for the free-text mood read, and `sweep-summary` for the monthly owner narrative. Claude Sonnet 4.6 (`global.anthropic.claude-sonnet-4-6-20250930-v1:0`) is wired as an optional fallback for replies Haiku flags as genuinely ambiguous, but in practice a one-sentence mood read is squarely Haiku's job and the fallback rarely fires.
- **Embeddings.** Not used. Reading the mood of one short reply is a direct classification, not a retrieval problem. No Knowledge Base, no S3 Vectors, no Titan embeddings.

- **Quotas.** Default account quotas are more than enough at SMB volume. The common path (star taps) never calls Bedrock; only the free-text minority and the monthly summary do.

EventBridge Scheduler config

- `fc-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `fc-calendar-sync` — `rate(1 hour)`. Target: `calendar-sync` Lambda.
- `fc-daily-sweep` — `cron(0 17 * * ? *)` in TZ. Target: `sweep-summary` Lambda (daily mode).
- `fc-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `sweep-summary` Lambda (monthly mode).
- **One-off ask rules** — created on the fly by `request-builder`, one per scheduled ask. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions in `TZ_NAME` (e.g. `Asia/Singapore`) with `--action-after-completion DELETE` so the rule self-cleans after firing.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `feedback.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `fc-inbound-rules`: one rule with recipient `feedback@your-company.com` → spam scan → S3 PUT to `s3://fc-raw-mime/<message-id>` → stop. The S3 PUT triggers `reply-handler`. The signed token rides in the unique reply-to address per ask, so the handler can match a written reply to its ask.

- SES outbound for the asks, the happy review nudges, and email owner pings: verify a sender identity at `hello@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request. If you send asks as SMS instead, the request-builder calls your SMS provider (or SNS) rather than SES for that leg.

IAM (least privilege per Lambda)

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

- **request-builder role:** `s3:GetObject` on the customer-list and rules keys; `scheduler:CreateSchedule` for the one-off ask rules; `secretsmanager:GetSecretValue` on the token-signing secret; `ses:SendRawEmail` from the verified sender; `dynamodb:PutItem` on `fc-state`. *No* `bedrock:*`.
- **reply-handler role:** `s3:GetObject` on `fc-raw-mime`; `secretsmanager:GetSecretValue` on the token-signing secret; `dynamodb:GetItem` + `PutItem` on `fc-feedback`; `lambda:InvokeFunction` on `router`. *No* `bedrock:*`.
- **router role:** `bedrock:InvokeModel` on the Haiku (and optional Sonnet) ARNs; `s3:GetObject` on the rules key; `dynamodb:PutItem` on `fc-feedback`; `lambda:InvokeFunction` on `dispatch`.
- **dispatch role:** `ses:SendRawEmail` from the verified sender; `secretsmanager:GetSecretValue` on the Slack bot token (for owner DMs); `dynamodb:PutItem` on `fc-audit`; outbound network access to `hooks.slack.com` / `slack.com` and, if used, the SMS provider.

- **intake-webhook role:** `secretsmanager:GetSecretValue` on the POS webhook secret and the Drive service-account secret; outbound network to `sheets.googleapis.com`.
- **drive-sync and calendar-sync roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the customer-list and rules buckets; outbound network to `www.googleapis.com`.

Signed-token flow

Every ask carries a token: an HMAC-SHA256 over `customer_id|ask_id|expiry`, signed with a secret in Secrets Manager (`fc/token/signing-secret`). For star taps the token is a URL parameter on each star link alongside the star count; for written replies it's embedded in a unique reply-to address. The `reply-handler` recomputes the HMAC and rejects anything tampered, expired, or replayed (a token whose `(customer_id, ask_id)` already has a row in `fc-feedback` is a no-op). This is what lets the star links be plain, login-free URLs while staying un-forgeable — nobody can spoof a five-star tap for a customer they aren't.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch metric for alerting.
- **Alarms:** request-builder failures > 0 in a day (an ask that never schedules is a lost review); reply-handler signature-verification failures > 5/hour (might mean the signing secret rotated); dispatch failure rate > 1% in 24h.

- **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 `fc-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive, Sheets, and Calendar APIs all live in Secrets Manager under `fc/drive/sa` (one service account with scopes for all three APIs). The token-signing secret is `fc/token/signing-secret`; the POS webhook secret is `fc/pos/webhook-secret`; the Slack bot token for owner DMs is `fc/slack/bot-token`. The configured timezone, holiday list reference, quiet-hours window, per-visit-type wait times, the star thresholds, the model confidence threshold, the public-review link, and the owner's private channel all live in Parameter Store under `/fc/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role, AWS SAM for the stack — no long-lived keys. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `fc-customers-source` and `fc-rules-source` so a bad Drive edit can be rolled back in one click, and version the EventBridge Scheduler timezone setting so you don't accidentally start sending asks at 3am after a CI rotation. Total deployable surface: around eight Lambdas, three DDB tables, three S3 buckets, the recurring and one-off Scheduler rules, one SES rule 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.