7-PART SERIES · FREE COMPANION

Review request sender

A job finishes and the moment to ask for a review opens and closes fast — ask too soon and it feels grabby, ask too late and the customer has moved on, ask the wrong person and a bad day turns into a public one-star. This is the design of a small serverless system that turns a completed job into one well-timed, personal review request: when an order or job is marked done it schedules a single ask for the right moment — a sensible delay, inside opening hours — then, just before sending, it reads whatever signal exists (a rating, a reply, past sentiment) and splits the road. Happy customers get a warm, personalised note with a direct review link; anyone who looks unhappy is quietly routed to a private feedback form instead, so a poor experience is caught in private rather than aired in public. It asks each customer once, honours opt-out and quiet hours, and never nags. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

Workflow guide $19 · Deployable AWS CDK starter $79 · Bundle $89

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/review-request-sender

CONTENTS

Review request sender

PART 1 OF 7

JULY 3, 2026 PART 1 OF 7 · REVIEW REQUEST SENDER SERIES ~11 MIN READ

A review request sender on AWS for a few dollars a month

The best time to ask for a review is a small window that most small businesses miss entirely: too soon and it feels grabby, too late and the customer has forgotten you, and asking someone who had a bad day just hands them a public megaphone. This post walks through the design of a small serverless system that catches that window — one warm, well-timed ask per finished job — while quietly steering unhappy customers to private feedback instead of a one-star.

KEY TAKEAWAYS

- When a job or order is marked complete, a webhook fires; the system schedules exactly one review request for a sensible later moment.
- Just before it sends, it reads whatever signal the customer left — a rating, a reply, past sentiment — and grades it happy, unhappy, or unclear.
- Happy customers get a warm, personalised ask with a direct review link. Anyone who looks unhappy is diverted to a private feedback form.
- Two small Bedrock calls do the soft work — one grades the signal, one writes the ask. Every real decision is plain Python.
- Designed on AWS for about $1.90/month at roughly 150 completed jobs. It asks each customer once, honours opt-out, and never nags.

The whole system on one page

Before any code, here's the shape of what we're designing. Every small business that finishes work for people is sitting on reviews it will never get — not because customers are unwilling, but because nobody asked at the right moment, in the right words. The valeter drives off, the physio waves the patient out, the boiler engineer packs up the van, and the thought "I should ask them for a review" either never comes or comes three weeks too late. The system below catches the finish of a job and turns it into one warm, well-timed ask — but only for the customers who seem glad, and quietly routes the rest somewhere private.

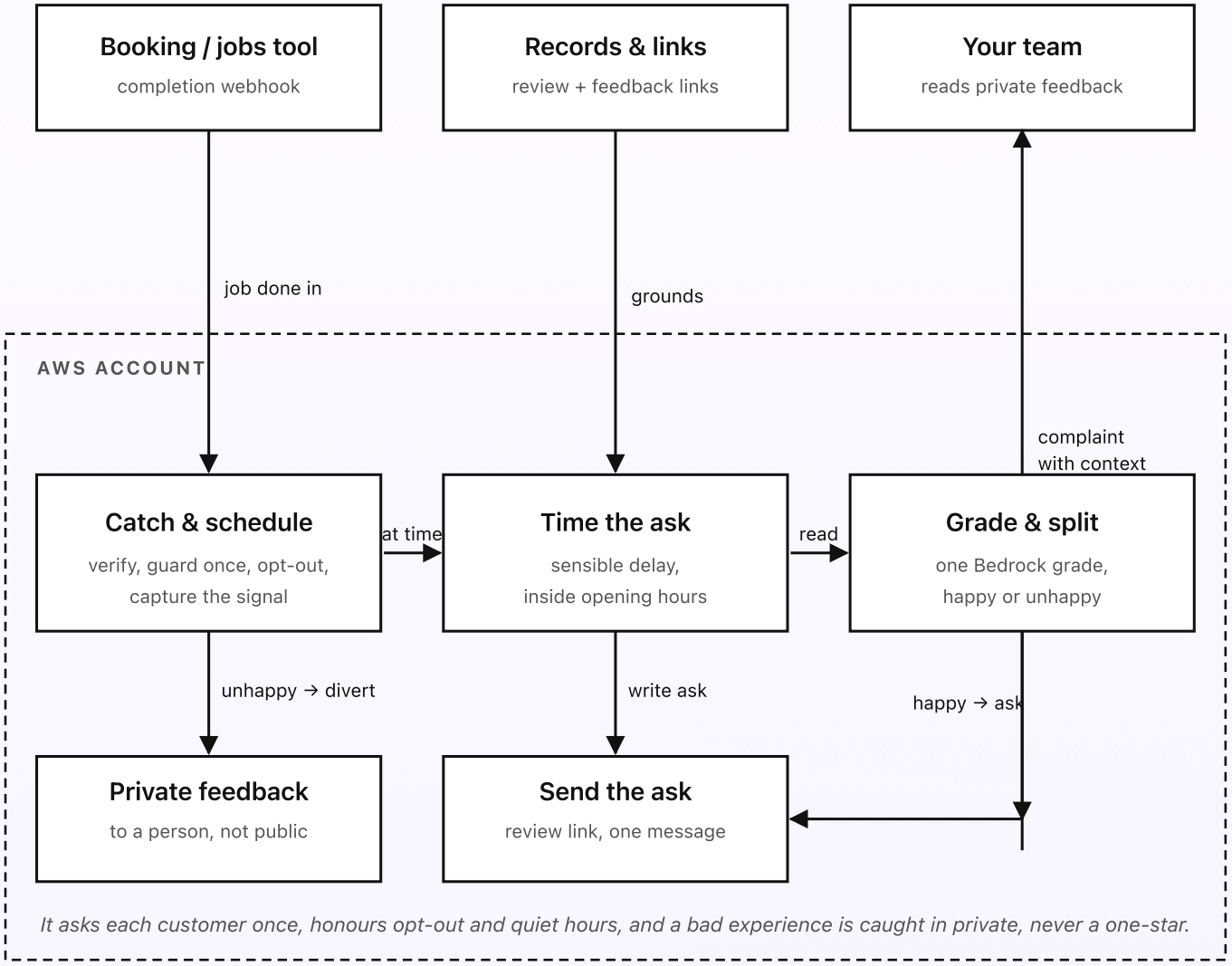

Booking / jobs tool
completion webhook
Records & links
review + feedback links
Your team
reads private feedback
job done in
grounds
AWS ACCOUNT
complaint
with context
Catch & schedule
verify, guard once, opt-out,
capture the signal
at time
Time the ask
sensible delay,
inside opening hours
read
Grade & split
one Bedrock grade,
happy or unhappy
unhappy → divert
write ask
happy → ask
Private feedback
to a person, not public
Send the ask
review link, one message
It asks each customer once, honours opt-out and quiet hours, and a bad experience is caught in private, never a one-star.

Fig 1. Three things outside, four pieces inside AWS. A completed job comes in from the booking tool; Catch & schedule decides an ask is warranted and books it for later, Time the ask fires it inside opening hours, and Grade & split reads the signal — happy customers get a warm ask with a review link, unhappy ones are diverted to private feedback and, if it's a real complaint, to a person.

What you set up once (the outside)

- **Booking or jobs tool.** Whatever already tells you a job is done — a booking system, a field-service app, a shop's order status, or a simple "mark complete" button. It needs to do one thing: fire a webhook when a job or order is marked complete, carrying the customer, the job, and any rating or reply they left. You point that webhook at one AWS URL and store its key in Secrets Manager. This is the trigger for everything, and it's covered in Part 2.
- **Customer records and two links.** A sheet with one row per customer: name, contact (email or mobile), the jobs they've had, and any prior sentiment you've captured. You already keep most of this; the system just mirrors it so an ask can open with the customer's name and know their history. Alongside it sit the two links every ask points to — your public review link (Google, Trustpilot, wherever you want stars) and a private feedback form that comes straight to you — plus a small settings doc for the voice, the delay, the opening hours, and the escalation rules.
- **Your team.** The owner or manager who reads the private feedback and fixes a real problem. When an unhappy customer fills in the private form — or a signal is bad enough to skip the ask entirely — they get a message with the customer, the job, the rating, and what was said, so they can ring and put it right before it festers. The system never argues, never apologises on the business's behalf,

and never publishes anything; it opens a quiet conversation and a human handles it.

What runs on every finished job (the inside)

- **Catch and schedule.** The booking tool posts a completion event to one Lambda Function URL. The function verifies the signature, guards against making a second request for a job it has already handled, checks the customer against the opt-out list, captures whatever signal is already attached, and schedules one ask for a sensible later moment. Only then does a request exist. This is Part 2.
- **Time the ask.** Each request gets its own one-off schedule — a day or so after the job, snapped inside opening hours — so the ask lands when it's welcome, not the second the van pulls away. At fire time the system re-checks that the customer hasn't opted out or already been asked, and reschedules rather than sends if the moment has drifted into quiet hours. This is Part 3.
- **Grade and write the ask.** The first Bedrock Haiku 4.5 call grades the signal — happy, unhappy, or unclear. For a happy customer, a second call writes one short, warm ask in the business's voice, and code injects the direct review link. The model grades and phrases; it never decides whether or when to send, or which link to use. This is Parts 4 and 5.
- **Divert the unhappy.** Anyone the gate reads as unhappy never sees the public review link. They get a gentle note pointing to a private feedback form, and a genuine complaint is handed to a person to put right — so the public star rating is only ever offered to customers who already seem glad. This is Part 5.

In plain words

It's Tuesday and Dan finishes a full valet on a customer's car — Marta, who booked through his online form. He taps "job complete" in the app and drives to the next street. Nothing happens straight away, which is the point. The next afternoon, about the time Marta's back at her desk and the gleaming car is fresh in her mind, her phone buzzes: "Hi Marta — thanks for having Dan's Mobile Valeting round yesterday, hope the car's still looking sharp! If you've two minutes, a quick review really helps a small business: dansvaleting.uk/review." She taps it, leaves five stars, and Dan never had to have the awkward "could you review me?" conversation at the kerb.

Across town a physio clinic finishes a course of treatment for a patient, Tom, who left a lukewarm three-star rating in the booking app on his way out. When his request comes due the next day, the gate reads that signal and does *not* send him to the public review page. Instead he gets a quieter note: "Hi Tom — we'd love to hear how your treatment went; anything we could have done better? A minute here goes straight to the clinic: meadowphysio.uk/feedback." His reply — the exercises were never really explained — lands in the practice manager's inbox, not on Google. She rings him, books a follow-up to walk through them properly, and a three-star drift becomes a fixed problem and a kept patient. One customer earned a public star; the other was caught in private, exactly as designed.

DESIGN RULES THAT SHAPED EVERY DECISION

- One job, one ask. A job marked complete twice, or a retried webhook, still makes a single request — it never double-asks or nags.
- Protect the public rating. Only customers who look happy are ever pointed at the public review link; the unhappy go to private feedback.
- Timing is deliberate. The ask waits a sensible delay and lands inside opening hours — never the instant the job ends, never at 2am.
- The model grades and phrases. It reads the signal and writes the words; whether, when, and which link are all deterministic.
- Opt-out is sacred. Anyone who has opted out is suppressed for good, checked again the instant before every send.
- A real complaint reaches a person. Bad feedback isn't filed and forgotten — it's handed to the owner with the job and the words attached.

Why this shape

Most small businesses handle reviews one of three ways: they don't ask at all, they ask everyone the same blunt way at the wrong moment, or they buy a bulk review-request tool that fires the same template at every customer regardless of how the job went. The first leaves a good reputation invisible — happy customers rarely review unprompted. The second feels grabby and gets ignored. And the third is actively dangerous: blast the same "leave us a review" text at everyone and you hand your unhappy customers a public stage, turning private grumbles

into permanent one-stars. The gap is a single well-timed, well-judged ask that knows the difference between a glad customer and an unhappy one.

The shape above fills exactly that gap and nothing more. It leans on the booking tool you already use as the trigger, keeps the customer sheet you already maintain as the source of names and history, and adds a small system that asks one good question at the right time — but only of the people who seem pleased. The happy case turns into a tap on a review link with no human involved. The unhappy case is pulled aside quietly, sent somewhere private, and, when it matters, put in front of a person with the whole story already gathered.

The next four posts walk through each piece in turn: how a completed job becomes a scheduled request, how that request gets timed, how the ask gets written, and how unhappy customers get filtered to private feedback. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JULY 3, 2026 PART 2 OF 7 · REVIEW REQUEST SENDER SERIES ~8 MIN READ

How a completed job triggers a request

Before a single review request exists, the system has to turn a raw ‘job done’ event into something safe to act on later: prove it’s real, prove it hasn’t already made a request for this job, check the customer hasn’t opted out, and capture the signal that will decide the ask. This post is about that gate — how a completion webhook becomes one scheduled, send-once request.

KEY TAKEAWAYS

- The trigger is a completion webhook from your booking or jobs tool, posted to one Lambda Function URL — no API Gateway.
- The webhook is verified by its signature before anything else runs, so a stranger can’t make the system pester your customers.
- A conditional write keyed on the job guarantees one request per completed job — a re-marked job or a retried webhook never asks twice.
- The opt-out list is checked here, before a request is even scheduled, so a customer who opted out never enters the pipeline.
- Whatever signal the customer already left — a rating, a reply, prior sentiment — is captured now and carried with the request.

From “job done” to a request

Everything starts with one event the business already generates but rarely does anything with: a job marked complete. Booking systems and field-service apps expose this as a webhook — when an order ships, an appointment is closed out, or someone taps “complete”, the tool sends a small HTTP POST with the customer, the job, a completion id, and often a rating or a note. That POST lands on a single Lambda Function URL. There’s no API Gateway in front of it; a Function URL is a plain HTTPS endpoint on the function itself, which is all a webhook needs and the cheapest way to receive one.

The catch function’s job is not to send an ask. It’s to decide whether an ask is warranted at all, and to turn the event into exactly one request scheduled for later. Most of this post is about the checks that sit between “a job finished” and “a request is on the clock”, because that gap is where a review system either earns trust or becomes the thing that texts a customer four times about the same haircut.

Prove it’s real, then prove it’s new

The first check is authenticity. A Function URL is public, so the first thing the function does is verify the tool’s signature — a hash of the request body signed with a shared secret held in Secrets Manager. If the signature doesn’t match, the request is dropped with a `401` and nothing else happens. Without this, anyone who found the URL could fabricate “completed jobs” and make the system message strangers; with it, only your booking tool can start a request.

The second check is duplication, and it's the one that keeps the system from nagging. A job can be marked complete more than once — an engineer re-opens and re-closes a ticket, an order flips to complete then back and forth, or the tool retries a webhook it thinks failed. Either way, the system must make exactly one request per completed job. So the function writes a request record keyed on the completion id, using a conditional write to DynamoDB: the first event for that job wins and creates the request; any repeat sees the existing record and stops. One finished job, one ask, however many times the status was touched.

Should we ask at all?

Two more gates decide whether a request should even be created. **Opt-out** is the suppression list: any customer who has ever replied STOP, unsubscribed, or been marked "do not contact" is recorded as suppressed, and the system will never message that contact again. It's checked here, at the front door, so an opted-out customer never even becomes a scheduled request — and it's checked *again* at send time in Part 3, because someone can opt out during the day the request is waiting. **Eligibility** is the second: some jobs shouldn't generate an ask at all — a warranty callback, a cancelled order, an internal job, or a customer asked for a review in the last few months already. Those are recorded and dropped, because the cheapest ask to handle is the one you correctly decide not to make.

Only once a job is verified, de-duplicated, not opted out, and eligible does the function look at *how the customer felt* — and capture the signal that Part 5 will later grade.

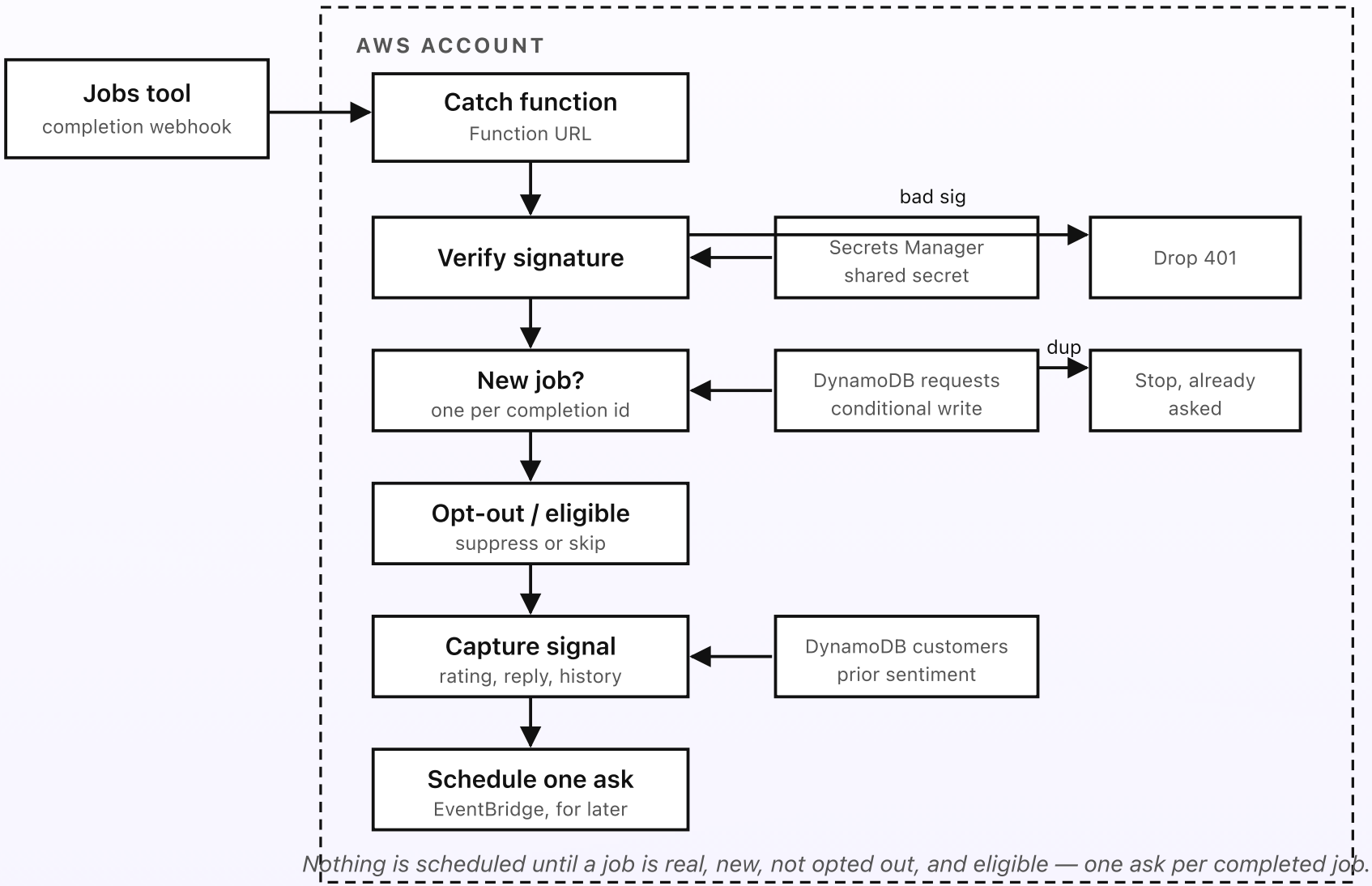

AWS ACCOUNT
Jobs tool
completion webhook
Catch function
Function URL
Verify signature
bad sig
Secrets Manager
shared secret
Drop 401
New job?
one per completion id
DynamoDB requests
conditional write
dup
Stop, already
asked
Opt-out / eligible
suppress or skip
Capture signal
rating, reply, history
DynamoDB customers
prior sentiment
Schedule one ask
EventBridge, for later
Nothing is scheduled until a job is real, new, not opted out, and eligible — one ask per completed job.

Fig 2. The catch gate. A completion webhook is verified by signature, guarded to one request per job with a conditional write, checked against the opt-out list and eligibility rules, and stamped with whatever signal the customer left — only then is a single ask scheduled for later.

Capturing the signal

The last thing the catch function does before scheduling is gather the signal — the raw material Part 5 will grade to decide happy from unhappy. There are three sources, in order of usefulness. Best is a signal the customer just gave: many booking tools collect a quick star rating or a one-line note at the point of completion, and if that's on the event it's the strongest read we'll get. Next is anything the customer said during the job — a reply, a message thread, a note the engineer logged. Last, if there's no fresh signal at all, is prior sentiment: the customer's history in the mirrored records, so a long-standing regular who's always been happy is read differently from someone whose last two visits went sideways.

All of it is captured now and stored on the request, but crucially it is *not* graded here. Grading is a soft judgement that belongs at send time, next to the compose step, and deferring it keeps this front-door function fast and deterministic: it does signature, dedup, opt-out, eligibility, and capture, and hands off. If a job carries no signal whatsoever — a boiler service where the engineer logged nothing and the customer said nothing — the request is still made, marked *unclear*, and Part 5 handles that band deliberately rather than guessing. Everything eligible becomes a clean request in DynamoDB: customer, job, contact, the captured signal, and the completion id, ready for Part 3 to put on the clock.

DESIGN RULES THAT SHAPED THE CATCH STEP

- One public surface. A single Lambda Function URL receives the completion webhook; there is no API Gateway and no other way in.
- Verify before you trust. A bad signature is dropped with a 401 before any work — the URL being public is fine because the secret isn't.
- One job, one request. A conditional write on the completion id collapses re-marks and retries into a single scheduled ask.
- Decide not to ask, cheaply. Opt-out and eligibility are checked before scheduling — a suppressed or ineligible job costs nothing.
- Capture, don't grade. The signal is gathered here and graded later, next to compose, so the front door stays fast and deterministic.
- No signal is still a signal. A job with nothing attached becomes an *unclear* request, handled deliberately rather than guessed about.

PART 3 OF 7

JULY 3, 2026 PART 3 OF 7 · REVIEW REQUEST SENDER SERIES ~7 MIN READ

How the ask gets timed

Timing is most of the craft here. Ask the moment the job ends and you sound like you're fishing; ask a week later and the goodwill has cooled; ask at 2am and you look broken. This post is about the clock: how each request gets its own one-off schedule at the right delay, kept inside opening hours, and re-checked the instant before it fires.

KEY TAKEAWAYS

- The ask never goes out the instant a job ends — that reads as grabby. Each request waits a sensible delay set per business.
- Each request gets its own one-off EventBridge Scheduler schedule at the computed due time — a timer per ask, not a busy polling loop.
- The due time is snapped inside opening hours, so a job finished at 8pm becomes an ask the next morning, never a buzz at 2am.
- At fire time the request is re-checked: opted out since scheduling? already asked? still eligible? If quiet hours have crept in, it reschedules.
- The clock is entirely deterministic. The model is never asked when to send — only, later, how to grade and how to phrase.

The window is narrow

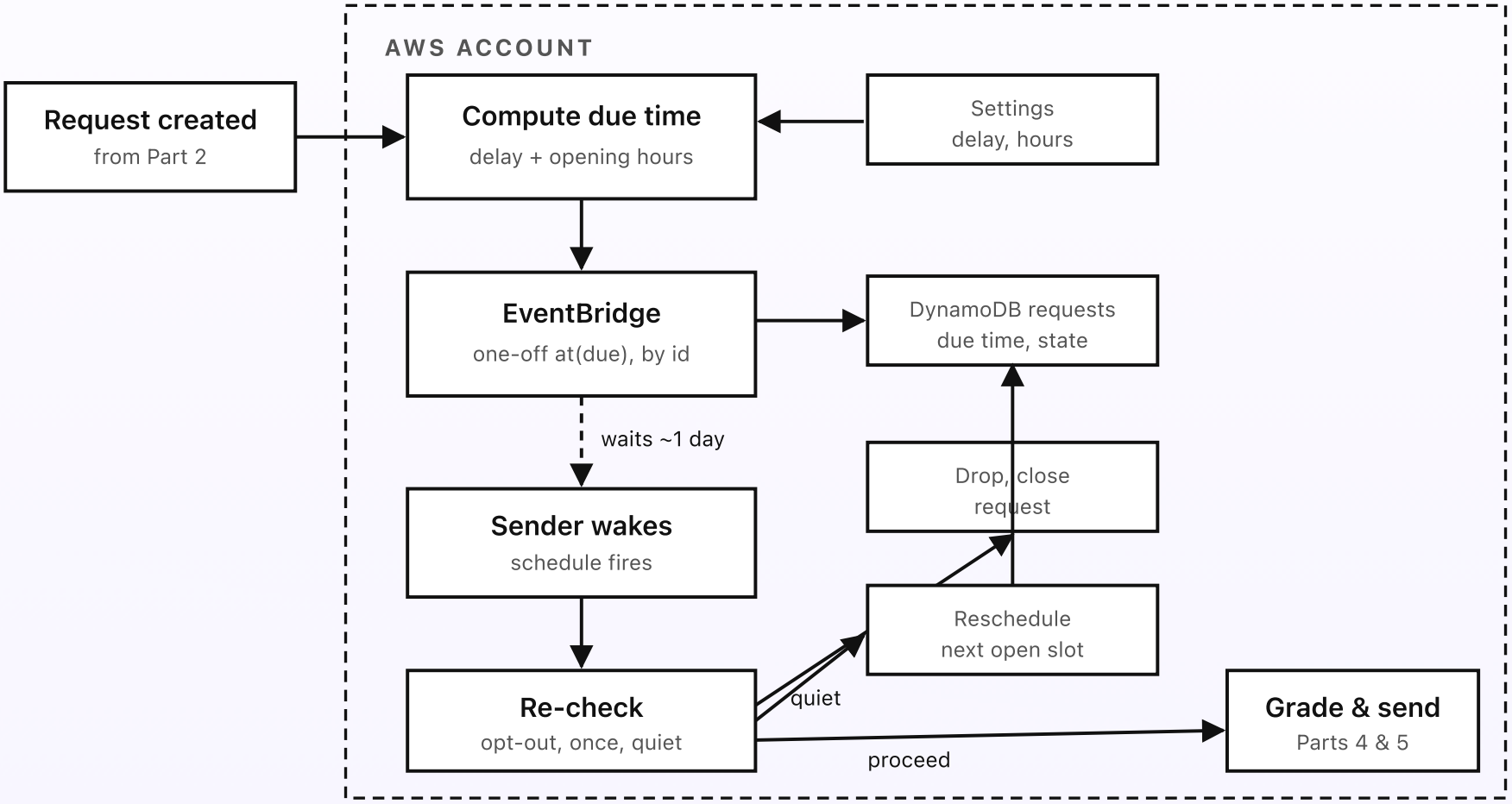
Timing is most of the craft in a review request, and it's the part cheap bulk tools get wrong. Ask the moment the job ends and you sound like you're fishing before the customer has even got home; the valet's still drying, the physio session's barely over, the boiler's not been through a cold evening yet. Ask a week later and the warmth has cooled to indifference. Ask at 2am because that's when the job happened to close in the system, and you look broken. The good window is a day or so out, when the work has proved itself and the customer still feels the benefit — and always at an hour a real business would actually message someone.

So timing gets its own stage, and one firm rule: the delay and the hour are computed by plain code, never by a model. The catch function in Part 2 handed over a request; this step decides *when* it fires and makes sure that when it does, the world hasn't changed underneath it.

A timer per ask

Each request gets its own one-off schedule in EventBridge Scheduler — a managed service built for exactly this: enormous numbers of one-time schedules, each firing a Lambda at a specific moment, with no always-on compute and effectively no cost. When a request is created, the code computes a due time — the configured delay (say 24 hours) added to the completion, then snapped forward into the next opening-hours window — and creates a schedule with an `at()` expression for that exact instant, named deterministically from the request id. Because the name is derived from the request, creating it twice is a harmless no-op, which keeps the whole thing idempotent even if Part 2 retries.

This is the neat inversion of the usual advice. A periodic “sweep every 15 minutes and see what’s due” loop works, but it wakes up constantly to mostly find nothing, and it fires a batch of asks all at the same tick. A schedule per request fires each ask at its own precise, business-appropriate moment and sleeps the rest of the time — and modern EventBridge Scheduler makes one-off schedules cheap enough that the old “don’t create a timer per item” worry no longer applies. A small periodic sweep still exists, but only as a safety net for schedules that misfire (Part 7), not as the main clock.



The delay and hour are computed by code and re-checked the instant before sending — the model is never asked when.

Fig 3. Timing the ask. A due time is computed from the delay and opening hours, a one-off EventBridge schedule is created for that instant and named from the request id, and when it fires the sender re-checks opt-out, once-only, and quiet hours — dropping, rescheduling, or proceeding to grade and send.

Snapped inside opening hours

Opening hours are what turn a raw “completion time plus 24 hours” into a moment a human would actually pick. The settings doc holds the business’s hours and a preferred send window — a valeter might like late morning, a physio clinic early evening after people leave work. The code adds the delay, then snaps the result forward to the next point inside that window: a boiler service finished at 8pm on Friday doesn’t become an 8pm-Saturday ask, it becomes a Saturday- or Monday-morning one depending on whether the business works weekends. Nobody gets a review request at an hour that says “this came from a machine that doesn’t sleep”.

Because a request can sit on the clock for a day or more, the world can change while it waits — which is why fire time is a checkpoint, not a blind send. The instant the schedule fires, the sender re-reads the request and re-runs the guards from Part 2: has the customer opted out since we scheduled this? Have they somehow already been asked through another route? Is this request still eligible? And crucially, has the due time drifted into quiet hours because of a bank holiday or a settings change — in which case it’s pushed to the next open slot rather than sent. Only a request that clears all of that proceeds to be graded and written. Everything else is dropped and closed, or quietly moved. The clock is precise, but it always defers to the guardrails.

DESIGN RULES THAT SHAPED THE TIMING

- Never the instant it ends. A configured delay lets the work prove itself before the ask — grabby is worse than late.
- A schedule per ask. One-off EventBridge schedules fire each request at its own precise moment, with no always-on polling.
- Snap into opening hours. The due time is moved to a sensible send window, so no ask ever lands at 2am.
- Fire time is a checkpoint. Opt-out, once-only, and eligibility are re-checked the instant before sending, not just at scheduling.
- Quiet hours win. If the moment has drifted into a closed window, the request reschedules rather than sends.
- Code owns the clock. Every timing decision is deterministic; the model is only ever asked to grade and to phrase, later.

PART 4 OF 7

JULY 3, 2026 PART 4 OF 7 · REVIEW REQUEST SENDER SERIES ~8 MIN READ

How the review ask gets written

By the time this step runs, the system has decided to ask, when to ask, and that this customer looks happy. All that's left is to say it like the business would. This post is about the place a model is allowed to write: the single Bedrock call that turns a few known facts into one warm, on-brand ask with a direct review link — and the fences that keep it to one honest message.

KEY TAKEAWAYS

- Once the gate has read a customer as happy, one Bedrock Haiku 4.5 call writes a single warm ask in the business's voice.
- The model is handed only the facts it may use: the customer's first name, the business name, the job, and a review-link token.
- The review link is injected by code after the model writes, never typed by the model, so it can never be mangled or invented.
- The draft is checked for length, a single link, and an opt-out hint before it's sent — a bad draft falls back to a fixed template.
- The model decides wording only. Whether to send, when, and that this customer is happy were all settled upstream.

All that's left is to say it

By the time this step runs, every real decision has already been made. Part 2 proved the job was real and eligible, Part 3 waited for the right moment and re-checked opt-out and quiet hours, and the sentiment gate in Part 5 has read this customer as happy. The request already carries the customer's first name, the job they had done, the channel to reach them on, and the review link that suits them. Nothing about *whether*, *when*, or *who* is open any more. The only thing left is the wording — and wording is what decides whether a review ask gets a tap or a shrug, because a message that reads like a mail-merge is one a happy customer still ignores.

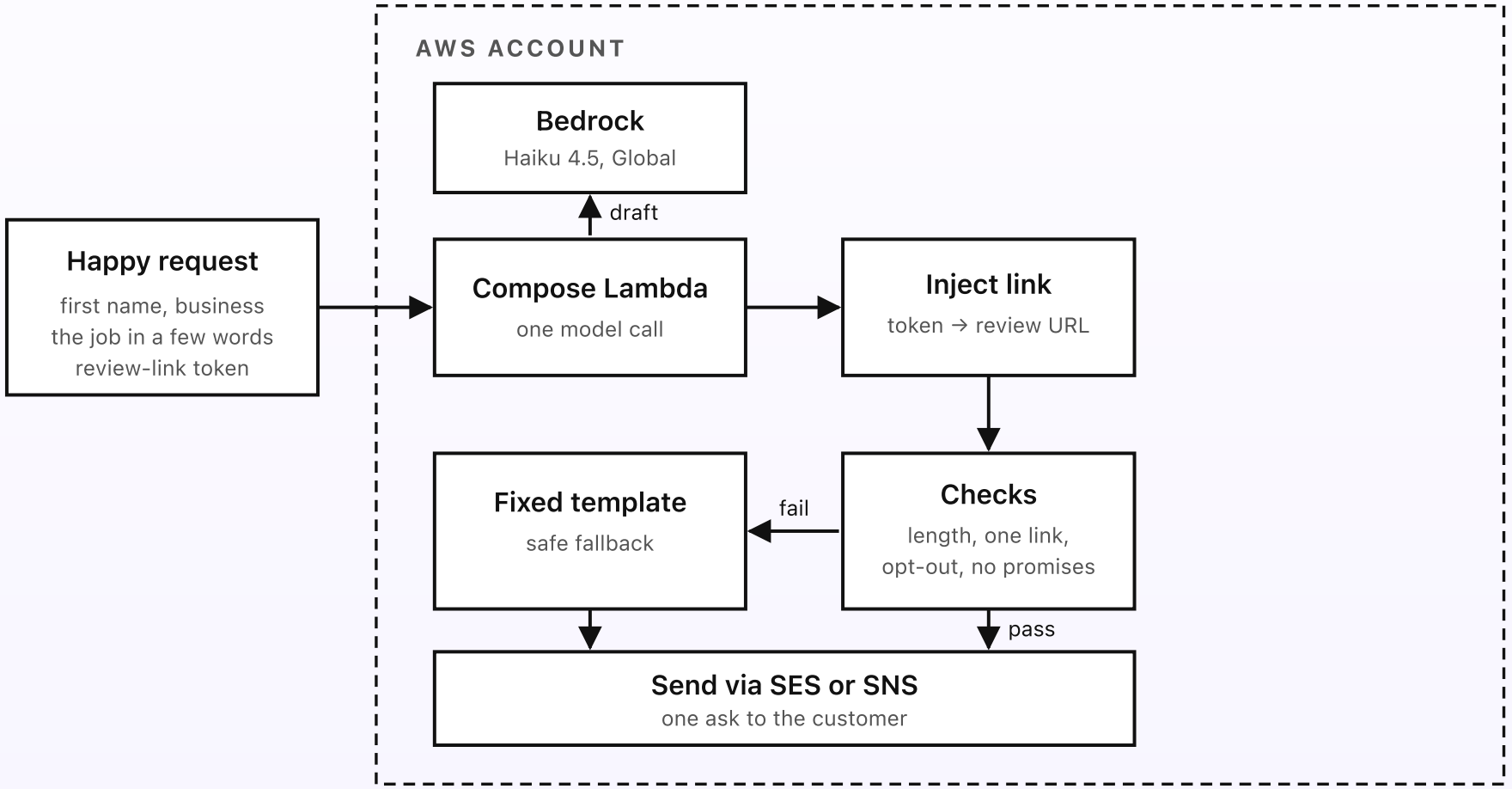
So this is one of only two places a model runs in the whole system, and the only one that writes customer-facing words. One Bedrock call, using Claude Haiku 4.5, takes a few facts and writes one short, warm ask that sounds like the business wrote it. Haiku is the right tool precisely because the task is small: a sentence or two, in a set tone, with no reasoning to do. It's fast and it costs a fraction of a penny, which matters when it runs on every happy customer.

Handed only the facts it may use

The prompt is deliberately narrow. The model is given the business name and voice notes from the settings doc ("warm, first-person, grateful without grovelling"), the customer's first name, the job in a few plain words ("full valet", "course of physio", "annual boiler service"), and a placeholder where the review link will go. It is told, in plain terms, to write one message under a sensible length that thanks the customer, references the job lightly, and invites a review — and to

use *only* those facts. It has no tools, no web access, and no knowledge of the customer beyond what it was handed, so there is nothing for it to over-share or invent about someone's appointment.

The review link itself is never written by the model. The model emits a token like `{{review}}`, and the code substitutes the real URL afterwards. This is a small thing that matters a lot: language models are perfectly capable of subtly mistyping a URL or "tidying" it into something that 404s, and a broken review link is the one error that quietly wastes the whole ask — the customer's goodwill was there, and it hit a dead page. By keeping the link out of the model's hands and injecting it deterministically, the link is always exactly right, and the same mechanism means the unhappy branch in Part 5 can inject the *private feedback* link into the same slot without the model ever knowing which road it's on.



The model writes words only; the link is injected by code, every draft is validated, and a bad draft falls back to a fixed template.

Fig 4. Composing one review ask. A single Bedrock call drafts the wording with a link placeholder; code injects the real review URL, the draft is validated for length, one link, an opt-out hint and no promises, and anything that fails falls back to a fixed template before the one ask is sent.

Checked before it goes

A model writing a one-off ask is low-risk, but “low” is not “none”, so the draft is validated by code before it leaves the building. If the ask goes by SMS it must come in under a sensible length or it’s trimmed or rejected; by email it still needs to stay short and skimmable. It must contain exactly one link, the one we injected, and no others. It must carry the small opt-out hint that keeps the business on the right side of messaging rules (“reply STOP to opt out”, or an unsubscribe footer on email). And it must not contain anything that reads as a discount, a bribe for a good review, or a promise — because paying or pressuring for reviews breaks every platform’s rules and the whole point is an honest ask.

If the draft fails any of those checks — or if Bedrock is slow or errors — the composer doesn’t retry forever or send something half-formed. It falls back to a fixed, hand-written template: “Hi {name}, thanks for choosing {business} for your {job}. If you’ve a moment, a quick review really helps us: {link}. Reply STOP to opt out.” The fallback is plain, but it’s always correct, always on time, and always within the rules. The model makes the common case sound human; the template guarantees the system never goes silent, or worse says something odd, just because a model had a bad moment.

Why let a model near it at all

It’s fair to ask why this needs a model when a template already exists as the fallback. The honest answer is tone and fit. A loyal regular who’s had ten valets reads differently from a first-timer; “annual boiler service” wants different warmth from “emergency call-out at 11pm”; a physio clinic and a car valeter want very different voices. A single template can’t flex to all of that without becoming a pile of `if` statements, and a stiff, obviously-templated ask is exactly what makes a happy customer decide it’s not worth two minutes. One cheap Haiku call gives every ask a little of the business’s actual character, while the deterministic link injection, the validation, and the template fallback make sure that character never comes at the cost of correctness. The model gets the warmth; the code keeps the guarantees.

DESIGN RULES THAT SHAPED THE COMPOSER

- One call, one ask. A single Haiku call writes one message — no chains, no retries that could fan out into more sends.
- Only the facts it's handed. First name, business, voice, job, and a link token — no tools, no lookups, nothing to over-share.
- The link is the code's job. The model emits a token; the real URL is injected afterwards, so review and feedback links can never be mangled or swapped.
- Validate every draft. Length, a single link, an opt-out hint, no bribes or promises — checked before it's sent.
- A template is always ready. If the model fails or drifts, a fixed, correct ask goes out instead — the system never goes quiet.
- The model decides nothing. Whether, when, to whom, and which link were all settled upstream; the model only chooses words.

PART 5 OF 7

JULY 3, 2026 PART 5 OF 7 · REVIEW REQUEST SENDER SERIES ~7 MIN READ

How unhappy customers get filtered

The single most valuable thing this system does is knowing who *not* to ask for a public review. This post is about the fork in the road: the sentiment gate that reads a rating or a reply, grades it, and quietly routes anyone who looks unhappy to a private feedback form — and a person — instead of handing them the public star rating.

KEY TAKEAWAYS

- The most valuable thing this system does is know who *not* to ask for a public review — that's the sentiment gate.
- One Bedrock call grades the captured signal into a simple band: happy, unhappy, or unclear. The band decides the road, not the model.
- Happy goes to the public review link; unhappy goes to a private feedback form that comes straight to the owner — never to Google.
- A genuine complaint is escalated to a person with the job and the words attached, so a bad experience gets fixed, not just filed.
- The gate errs toward private. An *unclear* signal gets a gentle open question, not a push for public stars.

Knowing who not to ask

Every other part of this system is in service of one idea: never hand an unhappy customer a public megaphone. Blast the same “leave us a review” link at everyone and you will, sooner or later, send it to the customer whose boiler was fixed late, whose physio never explained the exercises, whose car came back with a smear on the dash — and that customer, invited to rate you in public, will. A private grumble that a phone call would have settled becomes a permanent one-star that every future customer reads first. The gate exists to catch that person *before* the ask goes out and route them somewhere the problem can actually be put right.

So at send time, just before the composer runs, the request passes through the sentiment gate. Its whole job is to look at the signal captured back in Part 2 — a star rating, a reply, a run of past sentiment — and decide which of two very different messages this customer should get: the warm public ask, or the quiet private one. Everything downstream flows from that single fork.

Grade the signal, then fork

Grading is the second and last place a model runs, and it’s used exactly as narrowly as the composer. One Bedrock Haiku 4.5 call is handed the signal — “3 of 5 stars”, “took three visits to sort”, “lovely job, thanks” — and asked to grade it into one of three bands: **happy**, **unhappy**, or **unclear**. That’s all it returns: a band and a one-line reason, no prose, no decision. A model earns its place here because sentiment is genuinely fuzzy — “fine, I suppose” and “could’ve been better but ok” are the kind of lukewarm human phrasing a keyword list reads

wrong — but it never gets to *act* on its read. It labels; deterministic code branches on the label.

And the branching is deliberately cautious. A clear *happy* goes to the public review link and the warm ask from Part 4. Anything *unhappy* is diverted: no public link ever, a gentle note pointing to the private feedback form, and if the signal is bad enough, a straight escalation to a person. The interesting band is *unclear* — a middling three stars, a terse reply, or no signal at all — and here the gate leans private: it sends an open question ("how did we do? anything we could improve?") that goes to the feedback form rather than the review page. The reasoning is simple asymmetry: mistakenly sending a happy customer to a feedback form costs you one review you might have got; mistakenly sending an unhappy one to the public page costs you a one-star that never washes off. When unsure, protect the rating.

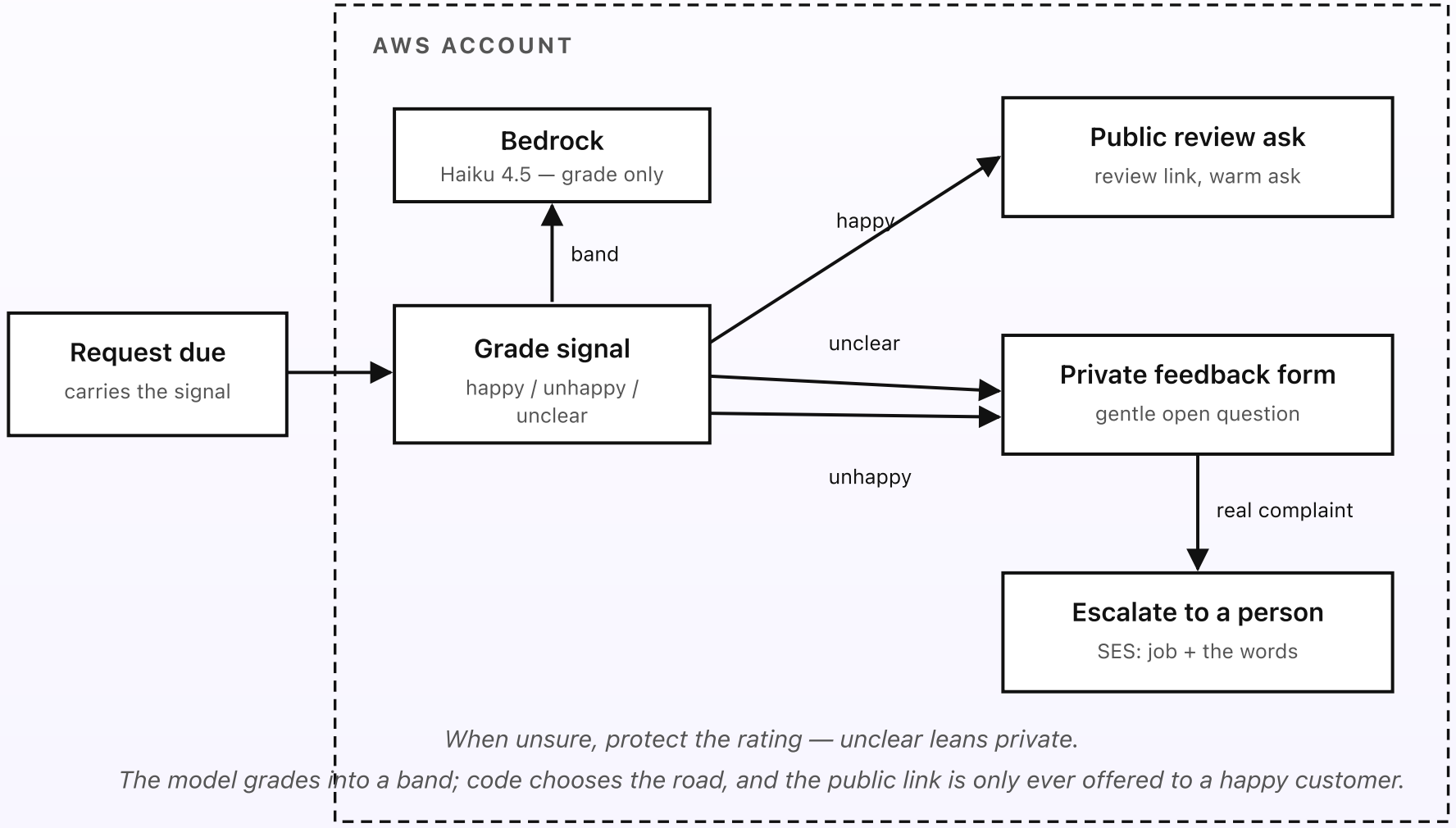

AWS ACCOUNT
Bedrock
Haiku 4.5 — grade only
band
Request due
carries the signal
Grade signal
happy / unhappy / unclear
happy
unclear
unhappy
Public review ask
review link, warm ask
Private feedback form
gentle open question
real complaint
Escalate to a person
SES: job + the words
When unsure, protect the rating — unclear leans private.
The model grades into a band; code chooses the road, and the public link is only ever offered to a happy customer.

Fig 5. The sentiment gate. One Bedrock call grades the captured signal into happy, unhappy, or unclear; deterministic code then forks — happy customers get the public review link, unclear and unhappy ones get a private feedback form, and a genuine complaint is escalated to a person with the job and the words attached.

What private feedback actually does

Diverting an unhappy customer only helps if the private path goes somewhere useful, so it does two things. First, it gives the customer a real outlet: a short feedback form, hosted plainly, that opens with a genuine question rather than a star-rating widget — "we'd love to know how it went, and anything we could have done better." A customer who feels heard in private is far less likely to go and vent in public, and often just wants the acknowledgement. Second, when they submit it — or when the signal was bad enough to skip the ask entirely — the system escalates to a person: the owner or manager gets an email through SES with the customer, the job, the original signal, and whatever they wrote, so they can pick up the phone. The physio clinic manager rings Tom about his exercises; the boiler engineer's boss calls the customer whose service ran late. The complaint gets *fixed*, not filed.

It's worth being clear about what this is not. This isn't about hiding bad reviews or gaming a rating — an unhappy customer is completely free to go and leave one star anywhere they like, and nothing here stops them. It's about which customers the business *proactively points* at its public page. Asking a delighted customer for a review and giving an unhappy one a private, human way to be heard first is exactly what a good business does by instinct when it has the time to notice. The gate just makes sure the busy ones notice every time, automatically, without ever having to read the mood themselves.

DESIGN RULES THAT SHAPED THE GATE

- Never megaphone the unhappy. The public review link is only ever offered to a customer graded happy — full stop.
- The model grades, code forks. Bedrock returns a band and a reason; deterministic code chooses the road and the link.
- When unsure, go private. An *unclear* signal gets a gentle open question to the feedback form, not a push for public stars.
- Fix it, don’t file it. A real complaint escalates to a person with the job and the words, so someone can put it right.
- Not censorship, curation. Anyone can still review in public; the system only chooses who it actively invites to.
- One grade per request. The signal is graded once at send time and recorded, so the fork is auditable and never re-litigated.

PART 6 OF 7

JULY 3, 2026 PART 6 OF 7 · REVIEW REQUEST SENDER SERIES ~6 MIN READ

What the review request sender costs

A system that costs more than the reviews it earns is a gadget. This post is the cost breakdown: every AWS service this design touches, what each adds up to at around 150 completed jobs a month, and why the total lands near $1.90 — plus what happens to the bill when the volume goes up tenfold.

KEY TAKEAWAYS

- About $1.90/month at roughly 150 completed jobs, and the fixed cost is almost nothing — nothing runs when no jobs are finishing.
- The biggest line is Bedrock: two small calls per job (one to grade, one to write), and even that is well under a pound.
- The only real fixed cost is Secrets Manager: two platform secrets at $0.40 each, billed whether or not a job comes in.
- At ten times the volume (around 1,500 completed jobs) the bill lands near $12 — it scales with use, not with idle time.
- SMS carrier fees vary by country and provider; the numbers here are a UK-leaning estimate, not a fixed AWS price list.

Where the money goes

The system is serverless end to end, so there's no instance ticking over between jobs and no idle bill. You pay for a review request only when a job finishes. At a typical small-business volume — call it 150 completed jobs a month, each grading a signal and most of them sending one ask, with a few private-feedback diversions and escalations on top — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two platform secrets — webhook signing key, review/messaging API key ($0.40 each)	$0.80
Bedrock (Claude Haiku 4.5)	Two small calls per job — grade the signal, compose the ask (~150 each)	$0.50
SES	Most review asks and feedback links by email, plus escalation to your team	$0.20
SNS (SMS)	Asks sent by text where that's the customer's channel (a country-varying estimate)	$0.20
DynamoDB (on-demand)	Jobs, requests, customers mirror, opt-out, audit — small reads and writes	$0.10
CloudWatch Logs	Function logs, 7-day retention	$0.05
Lambda (Python 3.14, arm64)	Webhook, sender, escalator, sweep, sync	$0.02

AWS service	What it does here	Monthly
SQS + DLQ	Buffering between the webhook and the slower model and send calls	$0.02
EventBridge Scheduler	One-off per-request send schedules, plus the safety-net sweep and sync	$0.01
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~150 completed jobs/month**	**$1.90**

The shape of that bill is the point. The only line that costs money while the system sleeps is Secrets Manager — two secrets at $0.40 each, $0.80 a month no matter what, which is over a third of the total at this volume. Everything else is genuinely usage-priced and rounds to zero at idle. The line that does the most work is Bedrock, because this system thinks twice per job — once to grade the sentiment signal and once to write the ask — yet even two Haiku calls a job come to half a dollar a month. The messages themselves split across email (cheap) and the odd SMS (a few pence each), and all the machinery doing the real work of catching, scheduling, and routing together costs less than the Bedrock line alone.

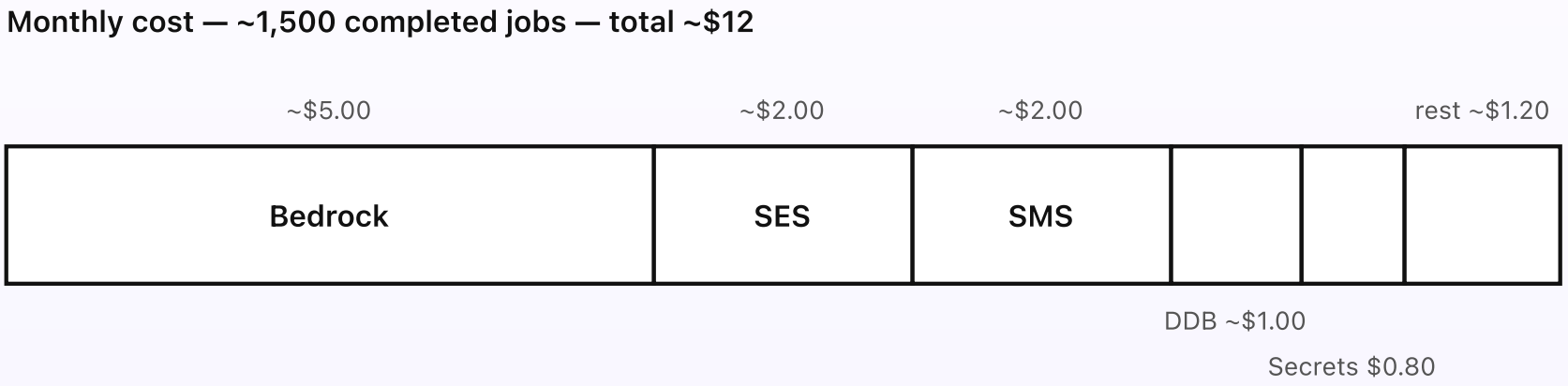
The line that isn't purely AWS

The SMS line deserves a caveat. AWS prices outbound SMS per message, and the exact rate depends on the destination country and the mobile carrier — a UK mobile is a few pence, other countries differ, and some routes add carrier

surcharges. The $0.20 here assumes most asks go by email through SES (fractions of a cent) and only a minority go by text; if you send every ask by SMS this line rises, and if your review platform sends the message on your behalf it moves onto their invoice and off this table entirely. Either way it's the same handful of pence per text, and it's the one line worth watching as volume grows — which is exactly why the AWS Budgets alarm sits on top of the whole thing.

What ten times the volume costs

Push this to a busy business — 1,500 completed jobs a month, ten times the volume — and the bill lands near $12, not $19. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules stay at a cent, and AWS Budgets stays free. What scales is the genuinely usage-priced work — about $5 of Bedrock for ten times the grade-and-compose calls, roughly $2 of email, another $2 or so of SMS, and a few dollars more spread across DynamoDB, logs, and Lambda. Even then, the two model calls per job dominate, and all the machinery in between stays close to free.



Fixed lines don't move with volume; only the two model calls and the messages scale, so the bill grows sub-linearly.

Fig 6. The monthly bill at ten times the base volume, about 1,500 completed jobs. Bedrock and the messages are the bulk of it; the fixed lines stay put, so the total grows sub-linearly — near $12, not ten times $1.90.

The honest way to read this: the AWS bill is rounding error against what a review is worth. A single extra five-star review for a valeter, a physio clinic, or a boiler engineer pulls in customers worth far more than $1.90 a month — and a single one-star quietly avoided is worth more still. Even at $12 a month for a busy shop, the system pays for itself the first time it turns a happy customer into a public review or catches an unhappy one before they reach for the stars — and every

unhappy customer it diverts still gets a human ringing them back, with the whole story already gathered.

DESIGN RULES THAT SHAPED THE COST

- Pay per completed job, not per hour. No always-on compute means no idle bill.
- Spend the model sparingly. Two Haiku calls per job — one to grade, one to phrase — and never to decide or to route.
- Cheap work stays cheap. Catching, scheduling, and routing are plain Lambda and DynamoDB, cents at this scale.
- Know your one fixed cost. Secrets Manager is the only line that bills while the system sleeps.
- Watch the messaging lines. SMS is the part whose price varies most, so the Budgets alarm sits right on top of it.

PART 7 OF 7

JULY 3, 2026 PART 7 OF 7 · REVIEW REQUEST SENDER SERIES ~10 MIN READ

Engineering reference: the review request sender architecture

This is the review request sender with the friendly labels removed: the real resource names, the runtime, the table key schemas, the single public Function URL, the per-request schedules, the two model calls, and the IAM scope. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Five Lambda functions, all Python 3.14 on arm64, with a single SQS queue and a dead-letter queue in front of the sender.
- One public surface: a single Lambda Function URL on `rrs-webhook` that takes completions, opt-outs, and private-feedback posts — no API Gateway.
- Five DynamoDB tables, all on-demand: jobs, requests (with a state GSI and a TTL), customers mirror, opt-out list, and an append-only audit log.
- EventBridge Scheduler runs one one-off schedule per request for the timed send, plus two recurring rules for the sweep and the sync.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called twice by the sender — grade, then compose. Single region, `eu-west-2`.

The architecture, for engineers

This is the same system as Part 1 with the friendly labels removed and the real resources named. Everything is in one region, `eu-west-2` (London), in one account. There is no API Gateway, no NAT Gateway, and nothing always-on; the only inbound surface is one Lambda Function URL, the timed send is driven by per-request EventBridge schedules, outbound messages go through SES and SNS, and work is buffered on a single SQS queue with a dead-letter queue.

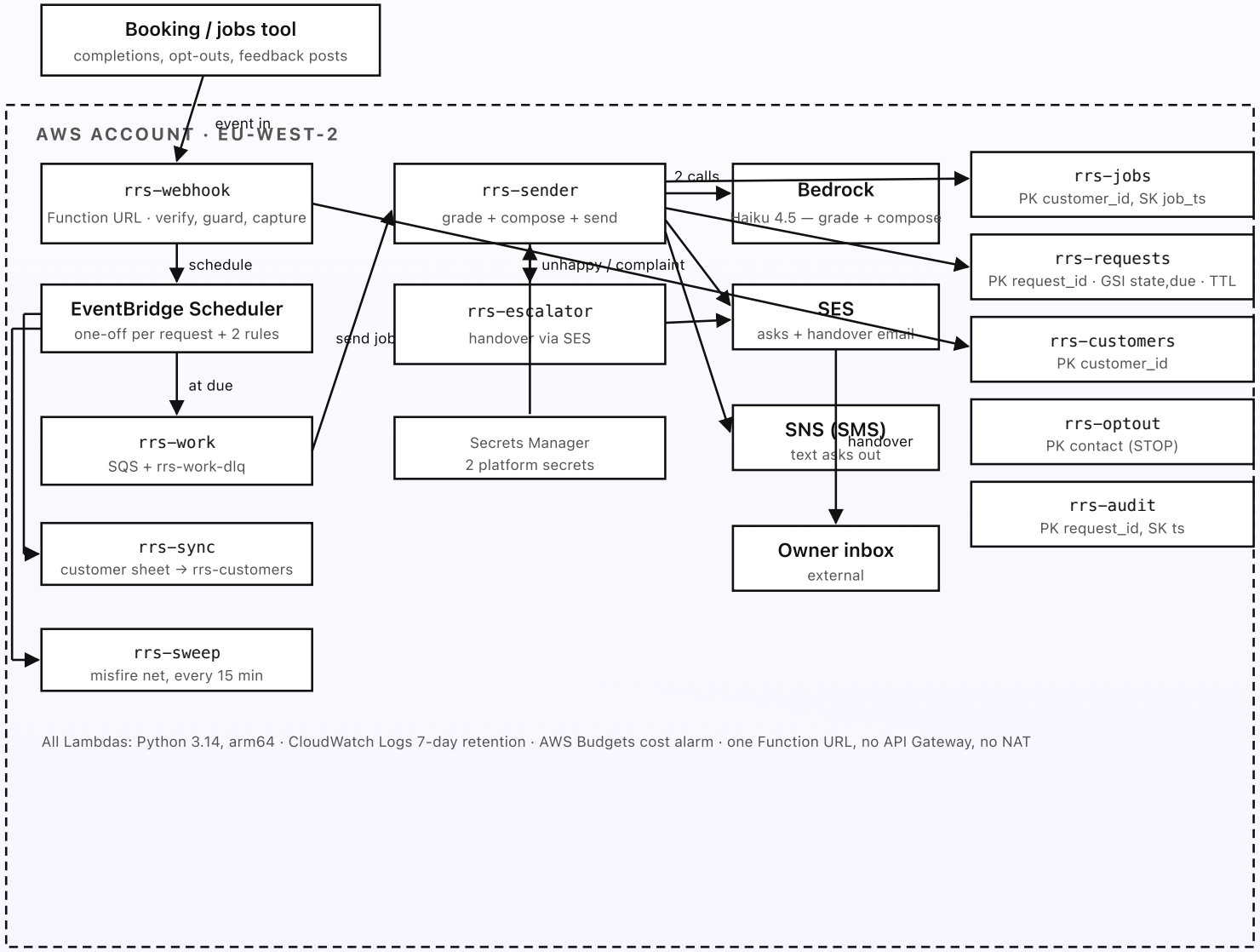

Booking / jobs tool
completions, opt-outs, feedback posts
event in
AWS ACCOUNT · EU-WEST-2
rrs-webhook
Function URL · verify, guard, capture
schedule
EventBridge Scheduler
one-off per request + 2 rules
at due
rrs-work
SQS + rrs-work-dlq
rrs-sync
customer sheet → rrs-customers
rrs-sweep
misfire net, every 15 min
send job
rrs-sender
grade + compose + send
unhappy / complaint
rrs-escalator
handover via SES
Secrets Manager
2 platform secrets
2 calls
Bedrock
Haiku 4.5 — grade + compose
SES
asks + handover email
SNS (SMS)
text asks out
handover
Owner inbox
external
rrs-jobs
PK customer_id, SK job_ts
rrs-requests
PK request_id · GSI state,due · TTL
rrs-customers
PK customer_id
rrs-optout
PK contact (STOP)
rrs-audit
PK request_id, SK ts
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm · one Function URL, no API Gateway, no NAT

Fig 7. The review request sender drawn for engineers: one Function URL on `rrs-webhook`, a one-off EventBridge schedule per request feeding an SQS-buffered `rrs-sender`, five Lambdas, five DynamoDB tables, Bedrock called twice by the sender, SES and SNS for the asks, and two recurring jobs. One region, one account, no API Gateway.

Lambda functions

Five functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. Each does one job and hands off; the SQS queue (`rrs-work`, with `rrs-work-dlq` as its dead-letter queue after five attempts) decouples the timed send from the slower model and messaging calls, and gives the send path free retries.

- `rrs-webhook` — the only public surface. Backs the single Lambda Function URL and handles all three inbound event types from the tool. For a completion: verifies the signature, guards one request per job against `rrs-requests` with a conditional write on the completion id, records the job in `rrs-jobs`, checks `rrs-optout` and eligibility, captures the signal from the event and `rrs-customers`, and creates a one-off EventBridge schedule for the computed due time. For a private-feedback submission: writes it to `rrs-audit` and enqueues an escalation. For an opt-out: writes `rrs-optout`. Nothing slow happens here.
- `rrs-sender` — SQS-triggered when a due schedule fires. Re-checks `rrs-optout`, once-only state, and quiet hours (rescheduling if the moment has drifted). Makes the first Bedrock call to grade the captured signal, branches on the band, and for a sendable branch makes the second Bedrock call to compose, injects the review or feedback link, validates the draft, sends one ask

via SES or SNS, and writes state to `rrs-requests` (`asked` / `diverted`) and `rrs-audit`.

- `rrs-escalator` — SQS-triggered. Builds the handover (customer + job + original signal + grade + any feedback text), de-duplicates against open escalations, and emails the owner inbox via SES so a person can put a real complaint right.
- `rrs-sync` — scheduled. Pulls the customer sheet from the source and upserts rows into `rrs-customers`, including the prior-sentiment history the gate reads when no fresh signal exists.
- `rrs-sweep` — scheduled safety net. Queries `rrs-requests` on the state GSI for requests whose due time has passed but which never reached a terminal state — a misfired or dropped schedule — and reschedules or escalates them, so no request is silently lost. It never re-sends a request already marked `asked` or `diverted`.

Data model and exactly-once

Five DynamoDB tables, all on-demand, hold every bit of state; the design leans on conditional writes and a single terminal state per request to make the whole thing safe to retry.

- **`rrs-jobs`** — PK `customer_id`, SK `job_ts`; one item per completed job with the raw completion payload and the signal captured at the door. It's the record of what finished and when.
- **`rrs-requests`** — PK `request_id` (derived deterministically from the completion id, which is what makes the one-per-job guard work). Each item

carries the due time, the channel, the grade once assigned, and a `state` that moves `scheduled` → `asked` / `diverted` / `suppressed` and never back. A GSI on `state` plus due time lets the sweep find stragglers cheaply, and a TTL on a retention timestamp expires old requests automatically.

- **`rrs-customers`** — PK `customer_id`; the directory mirrored from the sheet, including contact details and prior sentiment.
- **`rrs-optout`** — PK `contact` (email or E.164 mobile); the suppression list, checked at the door and again at send time before every message.
- **`rrs-audit`** — PK `request_id`, SK `ts`, append-only; every grade, every message, every escalation, and the facts each was built from, for when someone asks "why did this customer get that?".

Exactly-once rests on two moves. The *one request per job* guard is a conditional write to `rrs-requests` keyed on the completion id: the first completion wins, retries and re-marks no-op. The *one send per request* guard is a conditional state transition in `rrs-sender`: it only sends if the request is still `scheduled`, and flips it to a terminal state in the same conditional write, so a duplicate SQS delivery or a sweep racing the live path can never send twice. The EventBridge schedules are named from the request id, so re-creating one is idempotent too. Anything that still fails five times lands in `rrs-work-dlq` for inspection rather than being retried forever or lost.

Messaging, schedules, and the model

- **Function URL.** One, on `rrs-webhook`, with provider signature verification in-function; `AuthType NONE` at the edge because authenticity is enforced by the

shared secret, not by IAM. No API Gateway.

- **SES and SNS.** SES sends the review ask and the private-feedback ask by email from a verified domain with DKIM, and sends the escalation handover to the owner inbox; SNS sends the ask by SMS where that's the customer's channel. Every send re-checks `rrs-optout` first.
- **EventBridge Scheduler.** One one-off schedule per request — an `at()` expression at the computed due time, named from the request id, targeting `rrs-work` — plus two recurring rules: `rrs-sweep` at `rate(15 minutes)` gated to opening hours, and `rrs-sync` at `rate(15 minutes)`.
- **Secrets Manager.** Two secrets — the webhook signing secret and the review-platform / messaging API key — fetched at call time, never in env vars or the sheet.
- **Bedrock.** Model id `anthropic.claude-haiku-4-5` via the Global cross-Region inference profile, invoked only by `rrs-sender`: once to grade the signal into a band, once to compose the ask. Both calls are handed only the facts they need, and neither ever sees or writes a link.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `rrs-webhook` can read `rrs-customers` and `rrs-optout`, conditionally write `rrs-requests` and `rrs-jobs`, read the signing secret, create schedules, and send to `rrs-work` — it cannot call Bedrock, SES, or SNS. `rrs-sender` is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile; it can send via SES and SNS and write `rrs-requests` and `rrs-audit`, but cannot delete from any table. `rrs-escalator` can send via SES and read its

inputs only. The scheduled functions hold the narrow sheet and table permissions they need and no inbound surface at all. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call for capacity and is not a data store. An AWS Budgets alarm watches the monthly spend — with SMS and Bedrock the lines most likely to move, it's the cheapest early warning that volume, or a loop, is running hot.

That's the whole system: a completion webhook, a per-request schedule, two small model calls to grade and to phrase, and a firm rule that only happy customers are ever pointed at a public review — drawn for engineers, but built on the same idea the friendly diagram opened with. Ask once, ask well, ask only the people who are glad, and catch everyone else in private before a bad day becomes a bad rating.