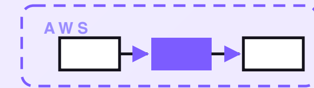
7-PART SERIES · FREE COMPANION



Warranty registration handler

A new product goes home in a box and the warranty card goes straight in the bin — so when something fails a year later, nobody can prove what they bought or when. This is the design of a small serverless system that turns registration into a thirty-second job: the buyer scans the QR on the product or fills a short form with the serial and proof of purchase, and the system verifies that purchase against the business's own order records, computes the warranty term and expiry, stores the record, and sends back a confirmation that explains the coverage in plain language. It schedules the maintenance and pre-expiry reminders so the customer hears from the business again at exactly the right moment, and routes any later claim to a person with the whole record attached. It registers a serial exactly once, rejects duplicates and out-of-window purchases, and keeps every record queryable. 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/warranty-registration-handler

CONTENTS

Warranty registration handler

01 A warranty registration handler on AWS for a few dollars a month

02 How a registration gets submitted

03 How a purchase gets verified

04 How a warranty record gets stored

05 How expiry reminders get scheduled

06 What the warranty registration handler costs

07 Engineering reference: the warranty registration handler architecture

PART 1 OF 7

JULY 6, 2026 PART 1 OF 7 · WARRANTY REGISTRATION HANDLER SERIES ~11 MIN READ

A warranty registration handler on AWS for a few dollars a month

A warranty is only worth what you can prove, and the proof usually goes in the bin with the packaging. This post walks through the design of a small serverless system that turns registering a new product into a thirty-second scan — verified against the business's own orders, with the coverage explained in plain language and the reminders already on the calendar — and quietly hands any later claim to a person.

KEY TAKEAWAYS

- A buyer scans the QR on the product (or fills a short form) with the serial and proof of purchase; seconds later the warranty is registered.
- The system verifies the purchase against the business's own order records — a serial that was never sold, or a duplicate, is turned away.
- It computes the warranty term and the exact expiry, stores a queryable record, and one Bedrock call explains the coverage in plain language.
- Maintenance and pre-expiry reminders are scheduled at registration and released on their due dates, so the customer hears back at the right moment.
- Designed on AWS for about $1.80/month at roughly 200 registrations. A serial registers exactly once; any later claim goes to a human with the record attached.

The whole system on one page

Before any code, here's the shape of what we're designing. Almost every product ships with a warranty and a little card that asks the buyer to register it — and almost nobody does. The card goes in the bin, the receipt fades, and eighteen months later a customer with a genuine fault can't prove what they bought or when, while the business has no idea who owns its products or how to reach them. The system below closes that gap at the one moment the buyer is actually holding the thing: they scan a QR code, the purchase is checked against real order records, and the warranty is registered, explained, and diarised within seconds.

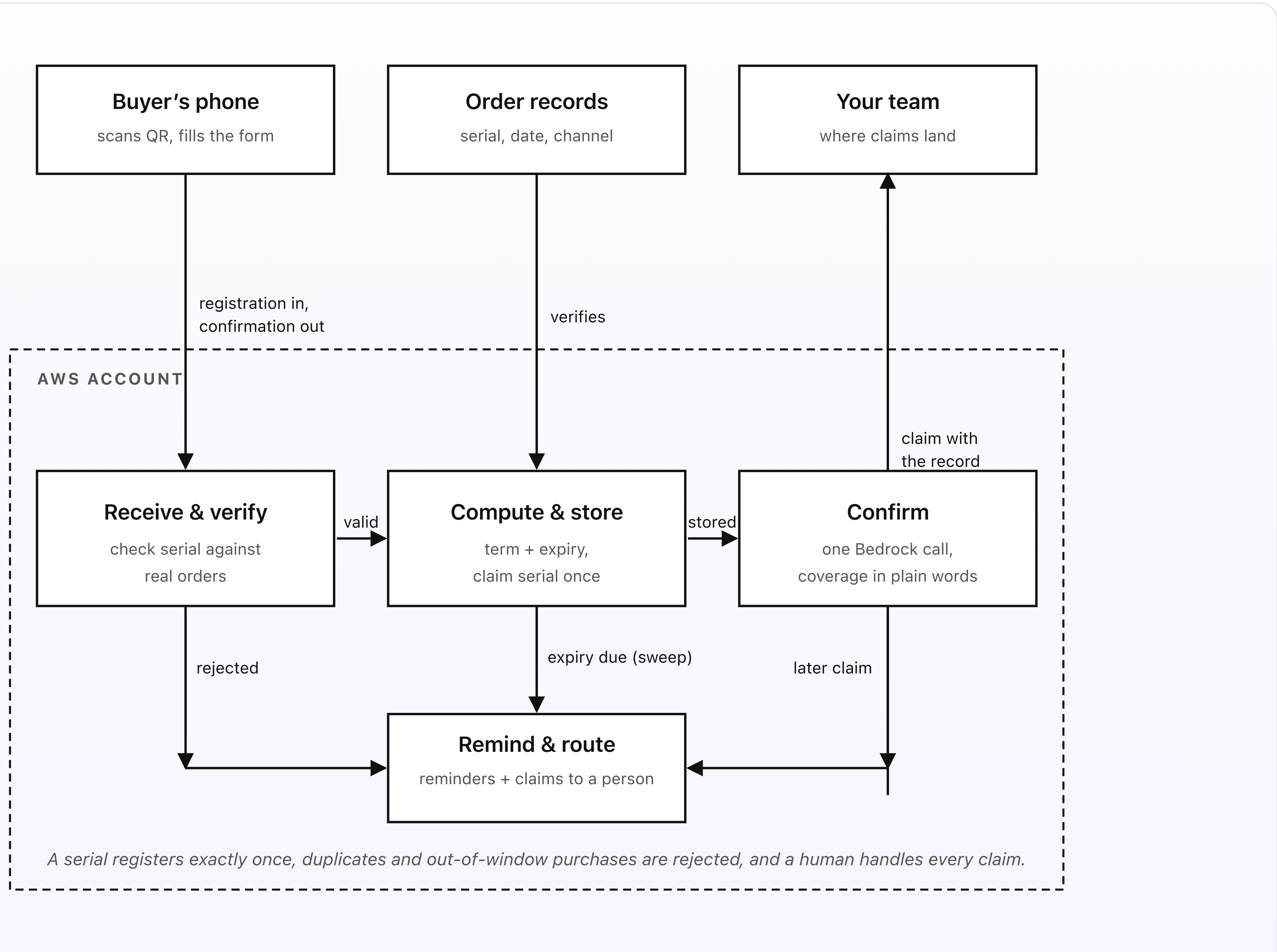

Buyer's phone
scans QR, fills the form
Order records
serial, date, channel
Your team
where claims land
registration in,
confirmation out
verifies
AWS ACCOUNT
claim with
the record
Receive & verify
check serial against
real orders
valid
Compute & store
term + expiry,
claim serial once
stored
Confirm
one Bedrock call,
coverage in plain words
rejected
expiry due (sweep)
later claim
Remind & route
reminders + claims to a person
A serial registers exactly once, duplicates and out-of-window purchases are rejected, and a human handles every claim.

Fig 1. Three things outside, four pieces inside AWS. A buyer registers a product; Receive & verify checks it against real orders, Compute & store computes the expiry and claims the serial once, and Confirm explains the coverage. A scheduled sweep sends reminders, and any later claim branches to a person.

What you set up once (the outside)

- **The QR sticker and the form.** Whatever your product already carries a serial on — a battery, a frame, a control board — gets a small QR label that opens the registration form pre-filled with that serial. Buyers without the scan can reach the same short web form and type the serial by hand. The form asks for two things only: the serial and a proof of purchase (an order number, or a photo of the receipt). It posts to one AWS URL, and its signing key lives in Secrets Manager. This is Part 2.
- **Order records.** The sales data you already have — from your shop platform, till, or dealer network — with one row per order: the serial or batch it shipped with, the purchase date, and where it was bought. This is mirrored into AWS so a registration can be checked against a real sale rather than taken on trust, and it's what tells the system a serial was genuinely sold, when, and therefore whether it's still inside the registration window. A small settings doc alongside it holds the warranty terms per product and the reminder timings. This grounds Part 3.
- **Your team.** The person who handles anything the system deliberately won't decide on its own — chiefly a warranty claim, months or years later. They get the claim with the full registration attached: the serial, the verified purchase, the computed expiry, and the confirmation that went out. The system registers,

explains, and reminds; a human handles claims, exceptions, and anything that smells wrong. This is Part 5.

What runs on every registration (the inside)

- **Receive and verify.** The form posts a registration to one Lambda Function URL. The function checks the request is genuine and not part of a flood, then verifies the serial and proof of purchase against the mirrored order records — rejecting a serial that was never sold, one that's already registered, and a purchase that falls outside the eligibility window. This is Parts 2 and 3.
- **Compute and store.** For a verified registration, the system reads the warranty term for that product, computes the exact expiry date from the purchase date, and writes the record — claiming the serial with a conditional write so the same unit can never be registered twice. The expiry and reminder dates are indexed so the sweep can find them later. This is Part 4.
- **Confirm.** One Bedrock Haiku 4.5 call takes the stored coverage terms — what's covered, for how long, until when — and writes a single warm, plain-language confirmation for the buyer. The model phrases the coverage; it never decides what the coverage is. This runs at the end of Part 4.
- **Remind and route.** A scheduled sweep reads the indexed dates and sends the reminders each warranty earns — a maintenance nudge partway through the term, a heads-up before the cover lapses — exactly once. And the claim lane hands any later claim to a person with the record attached. This is Part 5.

In plain words

It’s a Saturday afternoon and someone has just wheeled a new e-bike out of Cadence Cycles. On the down-tube is a small sticker: “Scan to register your 5-year warranty.” They scan it, the form opens with the frame serial `CDN-8842193` already filled, they type in the order number from the receipt, and tap send. Within about eight seconds their phone shows: “You’re registered, thanks! Your frame is covered for 5 years and the motor and battery for 2, until 14 July 2028. We’ll remind you when your first free service is due.” Behind that, the system matched the serial to the actual sale, saw the purchase was two days ago and well inside the window, computed the expiry from the purchase date, and stored the record — and the shop didn’t touch a thing.

Two years and ten months later, a scheduled sweep notices that frame’s motor-and-battery cover expires in a month. The buyer gets one message: “A heads-up — the motor and battery cover on your Cadence bike ends on 14 July. Frame cover continues to 2033.” A fortnight after that they email to say the motor’s cutting out. That’s a claim, not a registration — the system does *not* try to adjudicate it. It pulls the stored record, attaches the verified purchase and the exact coverage, and drops the whole thing in front of a person at the shop, who can see at a glance the claim is inside the motor warranty and act on it. One scan at the start, well-timed nudges in between, and a human on the one decision that matters.

DESIGN RULES THAT SHAPED EVERY DECISION

- One serial, one warranty. A conditional write on the serial means the same unit can never be registered twice, however many times someone tries.
- No purchase, no cover. Every registration is checked against real order records; a serial that was never sold, or is out of window, is rejected.
- The dates are computed, not typed. The term and expiry are derived from the product and the purchase date, then stored — never entered by hand.
- The model only writes words. Verification, the maths, and the scheduling are deterministic; Bedrock just explains the coverage in plain language.
- Remind at the right moment, once. Maintenance and pre-expiry nudges are diarised at registration and each fires exactly once, honouring opt-out.
- Claims go to a person. The system registers, explains, and reminds; it never adjudicates a claim — that lands with a human, record attached.

Why this shape

Most small brands handle registration one of three ways: a paper card almost nobody posts back, a generic web form that trusts whatever the buyer types, or nothing at all. The paper card is lost the moment the box is opened. The trusting form is worse than useless — it fills a database with serials that were never sold, warranties on grey-market units, and dates people guessed — so when a claim

arrives no one believes the record anyway. And doing nothing means the business never learns who owns its products, can't warn them about a recall, and meets every customer for the first time at the worst possible moment: when something has broken.

The shape above fixes exactly that. It leans on the order records the business already keeps as the single source of truth, so a warranty only exists behind a real sale. It makes registering effortless at the one instant the buyer is motivated — product in hand, box just opened — by putting a QR on the thing itself. And it turns the record into something that keeps working: reminders that bring the customer back for a service or a renewal, and a claim path that arrives with the evidence already gathered. The common case — a genuine buyer registering a genuine product — is fully automatic; the few that are duplicates, fakes, or out of window are turned away, and the one case that needs judgement, a claim, is put in front of a person.

The next four posts walk through each piece in turn: how a registration gets submitted, how a purchase gets verified, how a warranty record gets stored, and how the reminders get scheduled. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JULY 6, 2026 PART 2 OF 7 · WARRANTY REGISTRATION HANDLER SERIES ~9 MIN READ

How a registration gets submitted

Before anything is verified or stored, the system has to receive the registration safely: a scan or a short form, one public endpoint, and enough checking that a stranger can't flood it or register a serial they never bought. This post is about that front door — how a QR scan or a form submission becomes a single, claimed job ready to verify.

KEY TAKEAWAYS

- Registration starts with a QR sticker on the product or a short web form — both post to one Lambda Function URL, with no API Gateway.
- The form asks for two things only: the serial number and a proof of purchase; the QR pre-fills the serial so most buyers just add the receipt.
- The post is signed and rate-limited before anything runs, so a stranger can't script thousands of fake registrations against the endpoint.
- A repeat submission of the same serial in a short window collapses to one job, so a double-tap or a flaky connection doesn't create two.
- Nothing is verified or stored here — this step just turns a scan or a form into one clean, claimed job for the verifier to check.

From a scan to a job

Everything starts with the buyer doing the one thing warranty cards never manage to make them do: actually registering. The trick is to ask at the right moment and to ask for almost nothing. The moment is the unboxing, when the product is in their hands and the receipt is still in the bag; the "almost nothing" is a QR sticker on the product itself that opens a form with the serial already filled in. All the buyer adds is a proof of purchase — an order number, or a snap of the receipt — and taps send. Someone who's lost the sticker, or bought second-hand, can reach the same short form and type the serial by hand.

That form posts to 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 form submission needs and the cheapest way to receive one. The function's job here is *not* to decide whether the warranty is valid — that's Part 3's work, against the order records. Its job is narrower and it matters just as much: accept the submission safely, make sure it's genuine and not a flood, and turn it into exactly one clean job. Most of this post is about that front door, because a public endpoint that anyone can post to is where a registration system either stays trustworthy or fills up with rubbish.

Two ways in, one door

The QR path and the typed path produce the same thing. Take Halewood Tools, a small brand selling cordless drills: every drill leaves the factory with a serial laser-etched on the battery foot and a QR sticker beside it. A tradesperson scans it on a wet Tuesday, the form opens showing `HAL-DR18-004417`, they paste the

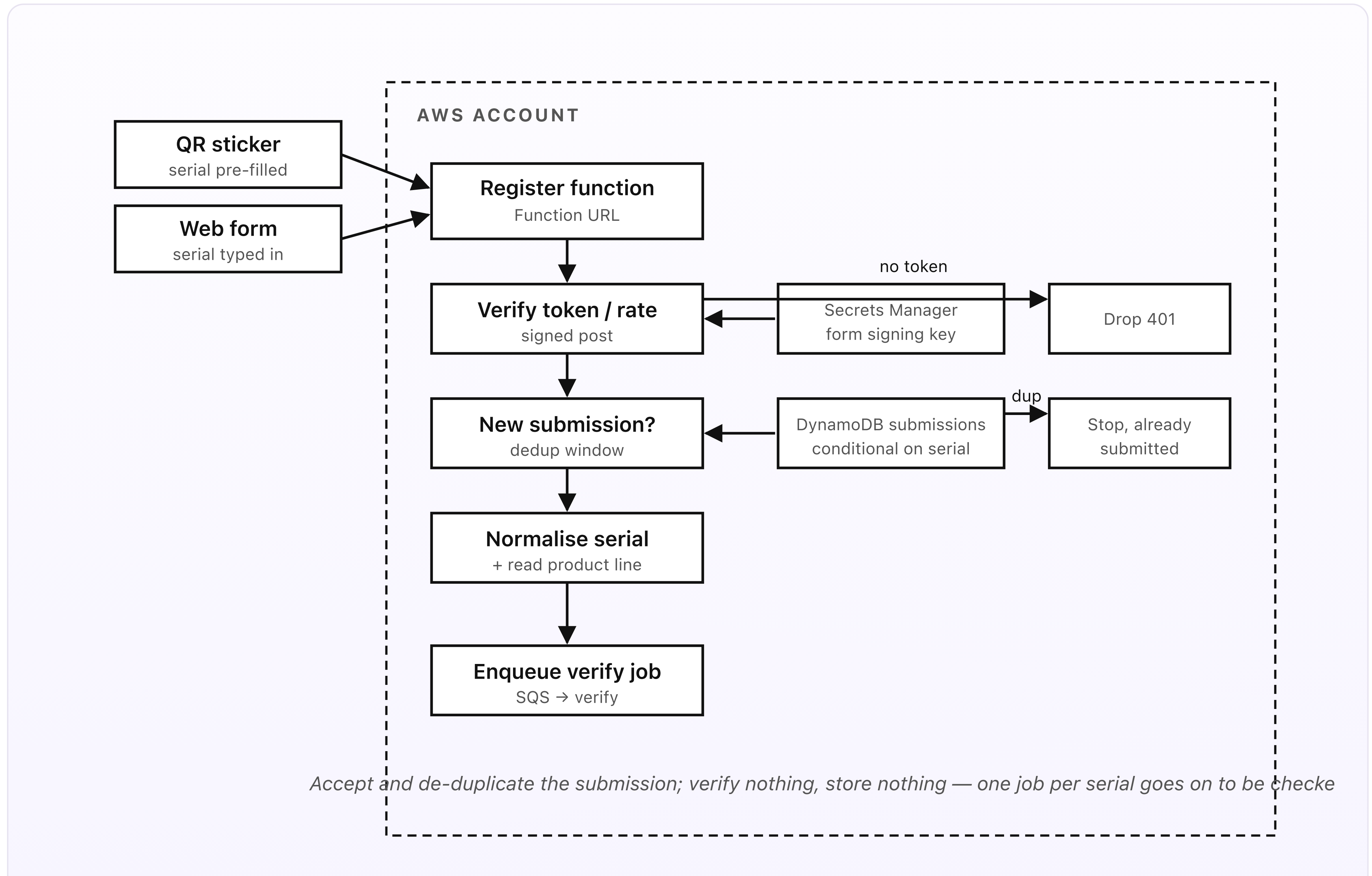
merchant's order number, and send. A different customer bought the same drill as a gift, has no sticker to scan, so they open `register.halewoodtools.co.uk`, read the serial off the battery, and type it in. Both submissions land on the same Function URL carrying the same two facts — a serial and a proof of purchase — plus a little context the form adds automatically: which product line, and a timestamp. From here on, the system doesn't care which way the buyer came in.

Keeping the form to two fields is a deliberate choice. Every extra box — name, address, email, model, date — is a reason to abandon the registration, and most of it the system can derive anyway: the order records already know who bought it, when, and what. Ask for the serial and the proof; recover the rest from the sale. The one contact detail the system does need for confirmations and reminders comes from the order record too, so the buyer never re-types what the business already holds.

Prove it's genuine, then prove it's new

A public URL invites abuse, so two checks run before the submission becomes a job. The first is authenticity and rate. The form is served with a short-lived signed token, and the post carries a signature the function verifies against a secret held in Secrets Manager; a request with no valid token is dropped with a `401`. On top of that the endpoint is rate-limited per source, so nobody can script tens of thousands of guesses at serial numbers to see which ones register. Without this, a public registration form is an open invitation to pollute the warranty database; with it, only real submissions from the real form get through.

The second check is duplication. Buyers double-tap, connections drop and retry, and an over-eager scanner can fire the same form twice. The system must create exactly one registration job per serial, so the function writes a short-lived submission marker keyed on the serial using a conditional write to DynamoDB. The first submission in that window wins and creates the job; any repeat inside the window sees the existing marker and stops. This is only the *submission* guard — the permanent “one warranty per serial” rule is enforced later, at the store step in Part 4, with its own conditional write — but catching duplicates this early keeps the verifier from doing the same work twice.



Accept and de-duplicate the submission; verify nothing, store nothing — one job per serial goes on to be checke

Fig 2. The submission gate. A QR scan or a form post hits one Function URL; the register function verifies the signed token and rate, collapses repeat submissions with a conditional write on the serial, normalises the serial and reads the product line, and enqueues a single verify job. It decides nothing about validity.

Cleaning up before handing off

Before the job goes on the queue, the function does the small, unglamorous work that makes Part 3's matching reliable. Serials arrive in a dozen shapes — with spaces, dashes, lowercase, a scanned barcode's leading zeros — so the function normalises to one canonical form, the same form the order records use, so that `hal dr18 004417` and `HAL-DR18-004417` are the same unit. It reads the product line from the form (the QR encodes it; the typed form has a small dropdown) so the verifier knows which warranty terms apply. And it keeps the proof of purchase as given — an order number to match, or a receipt image to hand to a person if the automatic match can't be made.

What it deliberately does *not* do is look anything up, compute anything, or send anything. No order records are read here, no expiry is worked out, no confirmation goes out. Keeping the public-facing function thin is what keeps it fast and cheap and hard to abuse: it does just enough to be sure the submission is real, singular, and tidy, then drops a clean job on the SQS queue — serial, proof, product line, timestamp — and returns. Part 3 picks it up and does the one thing that actually decides whether a warranty exists: checking it against a real sale.

DESIGN RULES THAT SHAPED THE SUBMISSION STEP

- One public surface. A single Lambda Function URL receives both the QR post and the typed form; there is no API Gateway and no other way in.
- Ask for two things. Serial and proof of purchase — everything else about the buyer is recovered from the order record, not re-typed.
- Verify before you trust. A signed token and a rate limit keep a public URL from being scripted into a pile of fake registrations.
- One submission, one job. A conditional write over a short window collapses double-taps and retries so the verifier never runs twice.
- Normalise at the door. The serial is cleaned to the canonical form the order records use, so matching later is a plain lookup.
- Decide nothing here. This step verifies no purchase and stores no warranty; it just hands a clean, singular job to Part 3.

PART 3 OF 7

JULY 6, 2026 PART 3 OF 7 · WARRANTY REGISTRATION HANDLER SERIES ~9 MIN READ

How a purchase gets verified

A registration is only trustworthy if the purchase behind it is real. This post is about the check that makes the whole system honest: matching the submitted serial and receipt against the business's own order records, and turning away the three things that shouldn't become a warranty — a serial that was never sold, one that's already registered, and a purchase that's outside the window.

KEY TAKEAWAYS

- A registration is only trusted if the purchase behind it is real, so the serial and proof are matched against the business's own order records.
- Three things are turned away here: a serial that was never sold, a serial already registered, and a purchase outside the eligibility window.
- The order records are mirrored into DynamoDB and keyed by serial, so verifying a registration is a direct lookup, not a slow scan.
- The eligibility window is computed from the purchase date on the order record — the date the buyer types is never trusted on its own.
- A near-miss — a receipt image but no clean order match — isn't rejected outright; it's handed to a person to confirm rather than guessed.

A warranty needs a real sale

By the time this step runs, Part 2 has given it a clean, singular job: a normalised serial, a proof of purchase, and the product line. What it hasn't done is decide whether any of that is true. That's this step's whole purpose, and it's the check that makes the difference between a warranty database worth having and one that's quietly full of fiction. A registration form that believes whatever it's told will happily record warranties on units that were stolen, counterfeited, bought grey-market, or simply invented by someone chancing a free repair — and every one of those looks identical to a genuine record until a claim arrives and nobody can trust it.

So the verifier does one deterministic thing: it matches the submission against the business's own order records — the sales data mirrored into AWS — and only a submission that corresponds to a real sale is allowed to become a warranty. There's no model here and no judgement call in the common case; a serial either was sold or it wasn't. Everything in this post is about the three ways a submission can fail that test, and the one careful exception where a person is asked to look.

The three rejections

Consider Bramble & Moss, a mattress company offering a ten-year guarantee. Every mattress carries a serial sewn into the side seam, and every sale — through their own site or a handful of stockists — lands in the order records as a row: serial, purchase date, channel. When a registration comes in, the verifier looks the serial up in that mirror and asks three questions in turn.

Was it ever sold? If the serial matches no order at all, the registration is rejected. This is the counterfeit-and-typo gate: a made-up serial, a mistyped one, or a genuine-looking unit that never passed through the books gets no warranty. A plain typo is handled gently — the buyer is told the serial wasn't found and asked to check it — but the system will not invent a sale to cover it. **Is it already registered?** If the serial matches an order but a warranty record already exists for it, the registration is rejected as a duplicate. Someone re-submitting their own registration is simply told it's already done; someone trying to register a mattress that isn't theirs is stopped cold. (The permanent enforcement of this lives in Part 4's conditional write; the verifier checks it early so it can give a clear answer instead of failing at the last step.) **Is it inside the window?** If the sale is real and unregistered but the purchase date on the order record is outside the registration

window — say, more than 90 days ago for a product that must be registered promptly — it's rejected as out of window, with the reason made plain.

The crucial detail is that every one of these answers comes from the order record, not from the buyer. The buyer's typed order number is used to *find* the sale; the purchase date, the channel, and the product all come from the sale itself. A buyer can't backdate a purchase to slip inside the window, or claim a product they didn't buy, because the facts that decide eligibility are the business's own, not theirs.

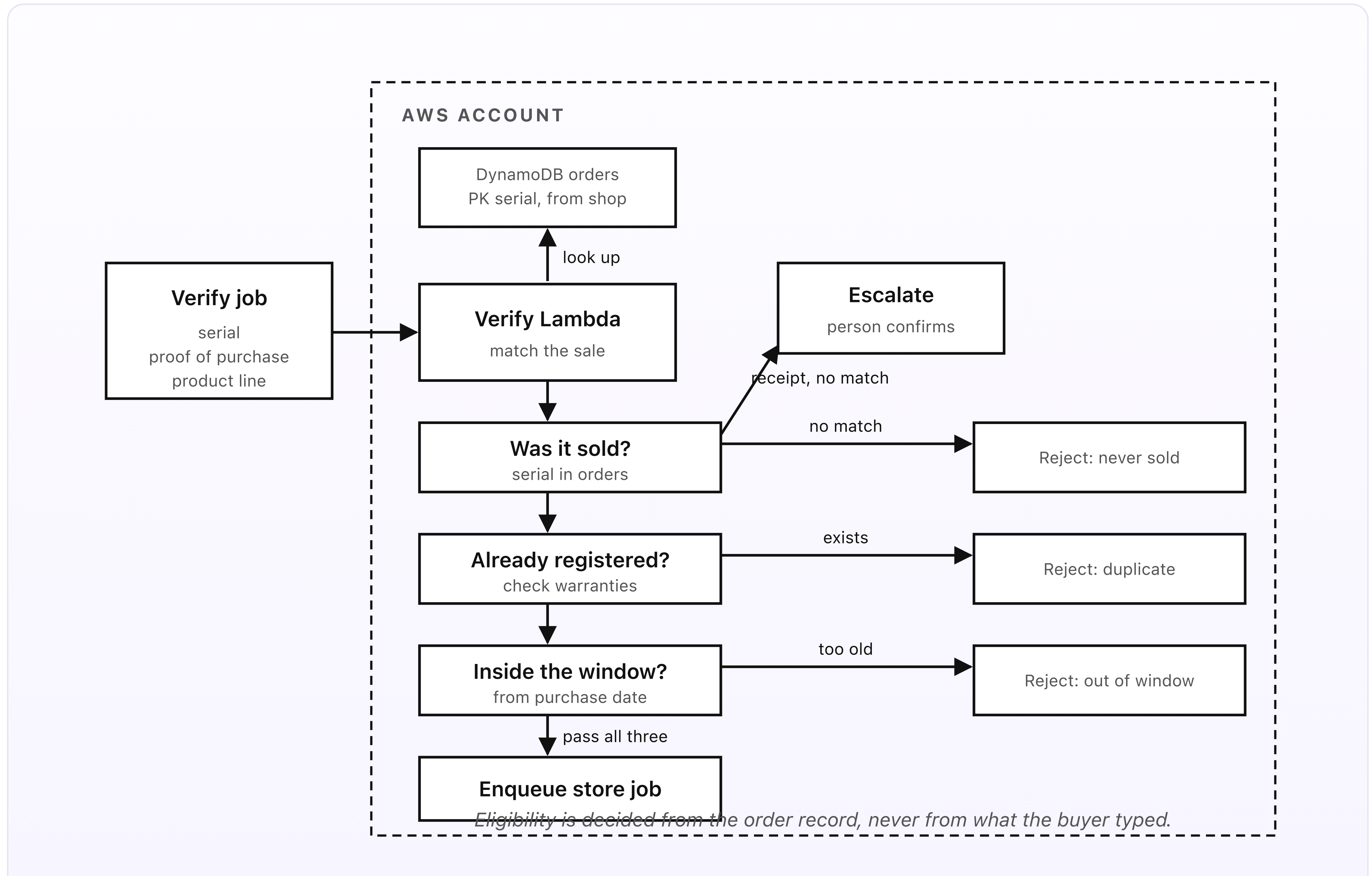

AWS ACCOUNT
DynamoDB orders
PK serial, from shop
look up
Verify job
serial
proof of purchase
product line
Verify Lambda
match the sale
Escalate
person confirms
receipt, no match
no match
Was it sold?
serial in orders
Reject: never sold
exists
Already registered?
check warranties
Reject: duplicate
too old
Inside the window?
from purchase date
Reject: out of window
pass all three
Enqueue store job
Eligibility is decided from the order record, never from what the buyer typed.

Fig 3. Verifying a purchase. The verifier looks the serial up in the mirrored order records and asks three questions in order — was it sold, is it already registered, is it inside the window — rejecting each failure with a clear reason. A receipt with no clean match goes to a person; a full pass becomes a store job.

A direct lookup, not a scan

Verification has to be quick and cheap because it runs on every registration, so the order records aren't queried live from the shop platform each time — they're mirrored into a DynamoDB table keyed by serial. A scheduled sync (covered in Part 7) keeps that mirror current: new orders flow in, so a mattress sold this morning can be registered this afternoon. Because the table is keyed by the serial, verifying a registration is a single-item lookup rather than a scan across the whole sales history, which is what keeps this step to a fraction of a penny even for a brand with years of orders behind it. The order number the buyer typed is used as a secondary confirmation — it should match the order the serial belongs to — which catches an honest mix-up where the right serial is paired with the wrong receipt.

When to ask a person

Not every imperfect submission deserves a flat "no". The common awkward case is a buyer who has a genuine receipt — a photo of a paper till slip from a stockist — but whose order isn't in the mirror cleanly: a dealer who batches their sales weekly, a marketplace order that lists the item differently, a serial that's a digit off from a real one. Rejecting these outright would punish real customers for the

business's own messy data. So a submission that looks genuine but can't be matched automatically isn't discarded — it's handed to a person, with the serial, the receipt image, and the closest order records attached, for a quick human yes-or-no. If they confirm it, the registration proceeds exactly as if it had matched; if they don't, it's declined with a note. The automatic path stays strict, and the escape hatch keeps it fair.

Everything that clears all three checks — the overwhelming majority — becomes a store job on the SQS queue: the verified serial, the sale it matched (with the real purchase date and channel), and the product line whose warranty terms apply. Part 4 takes that and turns it into a durable record.

DESIGN RULES THAT SHAPED VERIFICATION

- Match against real sales. A warranty only exists behind an order in the business's own records — nothing is taken on trust.
- Three clear rejections. Never sold, already registered, and out of window each get a plain reason, not a silent failure.
- The record decides, not the buyer. Purchase date, channel, and product come from the order, so eligibility can't be gamed by what's typed.
- Look up, don't scan. The orders mirror is keyed by serial, so verifying a registration is a single cheap lookup.
- A genuine near-miss goes to a human. A real receipt with no clean match is confirmed by a person, not rejected on a technicality.
- No model here. Verification is deterministic; a sale either exists or it doesn't, and that's not a question for a language model.

PART 4 OF 7

JULY 6, 2026 PART 4 OF 7 · WARRANTY REGISTRATION HANDLER SERIES ~9 MIN READ

How a warranty record gets stored

Once a purchase is verified, the system turns it into a durable fact: how long the cover runs, the day it ends, and a record that can never be quietly created twice. This post is about that write — the term-and-expiry maths, the conditional claim on the serial that guarantees one warranty per unit, and the expiry date indexed so the reminders can find it later.

KEY TAKEAWAYS

- The store step turns a verified sale into a durable record: the warranty term, the exact expiry date, and the coverage, all computed and written once.
- The term and expiry are derived from the product's warranty rule and the purchase date on the order — never typed by anyone.
- The serial is claimed with a conditional write, so the same unit can never be registered twice even if two submissions race at the same instant.
- The expiry and reminder dates are indexed on a GSI so the scheduled sweep can find due warranties without scanning the whole table.
- Only after the record is safely stored does one Bedrock call write the confirmation, explaining the coverage in plain language.

From a verified sale to a durable fact

By the time this step runs, Part 3 has proven the purchase is real, unregistered, and in window, and handed over a store job carrying the verified serial, the sale it matched, and the product line. What's left is to turn that into a fact the business can rely on for years: a warranty record that says exactly what's covered, for how long, and until when — written in a way that can never quietly happen twice, and indexed so the reminders can find it long after everyone's forgotten this particular Tuesday. This is the step where a registration stops being an event and becomes a record.

Two things make that record trustworthy: the dates are computed rather than entered, and the write is guarded so it can only ever create one warranty per serial. Get those right and everything downstream — the confirmation, the reminders, the claim months later — is standing on solid ground.

The term and expiry are computed

A warranty term is a rule about a product, not a number a customer supplies. The settings doc holds those rules per product line, and they're often layered: Cadence e-bikes, from Part 1, cover the frame for five years and the motor and battery for two; Halewood's drills are three years on the tool and one on the battery; a Bramble & Moss mattress is a flat ten years. The store step reads the rule for this product line and applies it to the *purchase date from the order record* — the verified one, not anything the buyer typed — to compute the exact expiry date, or dates, for each covered component. A two-year motor warranty on a bike sold on 12 July 2026 expires on 12 July 2028; the maths is plain calendar arithmetic, done once, in Python.

Computing the dates rather than storing a duration matters because a duration is a question and a date is an answer. "Two years" still needs a start date and a calendar to mean anything; "expires 12 July 2028" is unambiguous, sortable, and something the reminder sweep can query directly. So the record stores the resolved dates: when each component's cover ends, and the dates the reminders should fire. If the warranty rule ever changes, existing records keep the expiry they were sold with, because it's baked into the record at registration rather than recomputed from a rule that might have moved.

Claimed exactly once

The record is written to the warranties table keyed by the serial, and the write is a *conditional* one: create this item only if no item for this serial already exists. This is the permanent enforcement of the “one serial, one warranty” rule that the whole system rests on. Part 2’s dedup window and Part 3’s duplicate check are early, friendly guards that give a clear answer; this conditional write is the last, absolute one. If two submissions for the same serial somehow reach this step at the same instant — a double-tap that beat the dedup window, a retry racing the original — only one conditional write can succeed. The other fails cleanly, is recognised as a duplicate, and the buyer is simply told the product is already registered. There is no window, however small, in which a serial can end up with two warranties.

Because the write is idempotent in this way, the whole pipeline behind it can safely retry. If the store Lambda crashes after computing the dates but before confirming, the job comes back off the queue and runs again; the second attempt either completes the write or discovers the record it already made. That’s what lets the system lean on SQS and at-least-once delivery without ever double-registering — the conditional write is the single point where “exactly once” is actually decided.

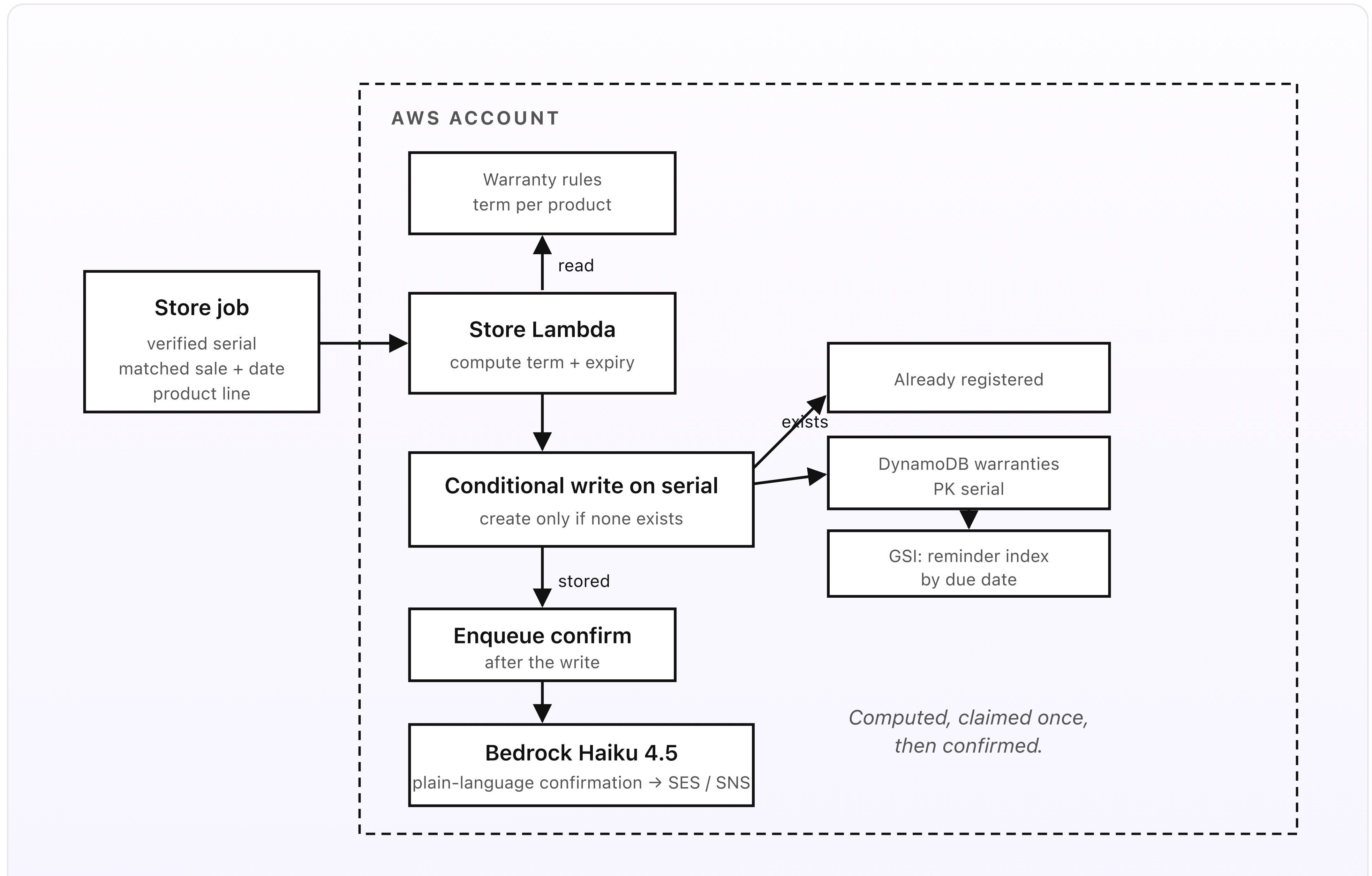

AWS ACCOUNT
Store job
verified serial
matched sale + date
product line
Warranty rules
term per product
read
Store Lambda
compute term + expiry
Already registered
exists
DynamoDB warranties
PK serial
Conditional write on serial
create only if none exists
GSI: reminder index
by due date
stored
Enqueue confirm
after the write
Bedrock Haiku 4.5
plain-language confirmation → SES / SNS
Computed, claimed once,
then confirmed.

Fig 4. Storing a warranty. The store step reads the warranty rule, computes the term and exact expiry from the verified purchase date, then claims the serial with a conditional write — only one can win. The record carries the coverage and the reminder dates, indexed on a GSI, and only a successful write triggers the confirmation.

Indexed so the reminders can find it

A warranty record isn't much use to the reminder system if the only way to find the ones due next week is to read every record the business has ever created. So alongside the coverage and expiry, the store step writes the reminder dates onto a global secondary index — a second way into the same table, keyed by *when* a warranty next needs attention rather than by which serial it is. The maintenance nudge for the Cadence bike (say, five months after purchase) and the pre-expiry heads-up (a month before the motor cover ends) each land in that index under their due date. Part 5's sweep queries the index for "everything due today" and gets a short, exact list, no matter how many warranties are stored in total. A time-to-live attribute is also set on the record's housekeeping markers, so transient state expires itself and the table stays lean; the warranty record proper is kept as long as the cover runs, and beyond, so a claim can always be looked up.

Only then, the confirmation

The buyer's confirmation is the last thing to happen, and deliberately so: nothing is sent until the record is safely stored, because a "you're registered" message that isn't backed by a saved warranty is a lie waiting to be found out. Once the write succeeds, a confirm job is enqueued and one Bedrock Haiku 4.5 call — the

only place a model runs in the whole system — turns the stored coverage into a warm, plain-language message. It's handed only the facts from the record: the product, what's covered, and the expiry dates it just computed. It writes something like “You're all set — your mattress is guaranteed for 10 years, until 4 August 2036,” not a wall of terms and conditions. The model phrases the coverage; it never decides what the coverage is, and the dates in the message are the ones the code computed, not numbers the model invented. As in any well-behaved use of a model here, the draft is checked for the right dates and a sane length, and a plain fixed template stands ready if the model is slow or drifts, so the confirmation is always correct and always on time.

DESIGN RULES THAT SHAPED THE STORE STEP

- Compute dates, don’t store durations. The exact expiry is derived from the verified purchase date and baked into the record.
- One conditional write decides everything. The serial is claimed only if unclaimed, so “one warranty per unit” holds even under a race.
- Idempotent by design. Because the write is conditional, the whole pipeline can retry safely without ever double-registering.
- Index by due date. Reminder dates go on a GSI so the sweep finds what’s due with a query, not a full-table scan.
- Store first, then say so. The confirmation is only sent after the record is written — no message without a warranty behind it.
- The model phrases, the code decides. Bedrock explains the coverage in plain words; the dates and terms are the ones computed, never invented.

PART 5 OF 7

JULY 6, 2026 PART 5 OF 7 · WARRANTY REGISTRATION HANDLER SERIES ~8 MIN READ

How expiry reminders get scheduled

The registration isn't the end of the relationship; it's the start of it. A warranty record carries dates that matter months or years away — a service is due, the cover is about to lapse — and something has to notice them at the right moment. This post is about the scheduled sweep that turns those stored dates into well-timed, once-only reminders.

KEY TAKEAWAYS

- A warranty record carries dates that matter months or years away — a service is due, the cover is about to lapse — and something has to notice them.
- An EventBridge Scheduler rule runs a sweep on a fixed daily cadence, querying the reminder index for warranties whose next nudge falls due.
- Two kinds of reminder go out: a maintenance nudge partway through the term, and a pre-expiry heads-up before the cover ends.
- Each reminder is sent exactly once — the sweep marks it done on the record — and honours the opt-out list, so no one is nagged or double-messaged.
- The sweep never touches eligibility or claims; it only turns already-stored, already-computed dates into well-timed messages.

The dates that matter later

Registration captures the customer at their most engaged — product in hand, box just opened — and then, if nothing else happens, the relationship goes quiet for years. That silence is a waste. A warranty record quietly holds two moments worth breaking it for: a point partway through the term when a service would keep the product healthy (and the customer happy), and a point near the end when the cover is about to lapse and the customer should know. Neither is triggered by anything the customer does; they're triggered by a date arriving. A system that

only ever reacts to incoming events would sail straight past both, because the thing that needs acting on is the calendar turning over, not a message coming in.

Catching a date needs something that runs on a clock. That’s the reminder sweep: a small scheduled job whose whole purpose is to read the dates every warranty stored back in Part 4 and send the nudge each one has earned, at the right moment and exactly once.

A job on a clock

The sweep is driven by EventBridge Scheduler — a managed cron that invokes a Lambda on a fixed cadence with no always-on compute and effectively no cost. Once a day is plenty; a warranty reminder is not a time-of-day matter, and a day’s granularity across a multi-year term is invisible to the customer. Each run does one simple thing: query the reminder index — the GSI keyed by due date from Part 4 — for every warranty whose next reminder falls on or before today and hasn’t already been sent. Because the dates were computed and indexed at registration, this is a tight query that returns a short list, not a scan across every warranty the business holds.

The two reminders are cut from the same cloth but say different things. The **maintenance nudge** fires partway through the term — five months into a Cadence e-bike’s life, say, when the first free service is due, or a year into a mattress warranty with a “rotate it and check the support” tip. The **pre-expiry heads-up** fires a set period before a component’s cover ends: “your Halewood drill’s battery cover ends on 3 October — the tool itself stays covered to 2029.” Both are drawn straight from the stored record, so the dates in the message are the exact ones

computed at registration, and both go out through the same messaging path the confirmation used: email by default, SMS where the customer preferred it.

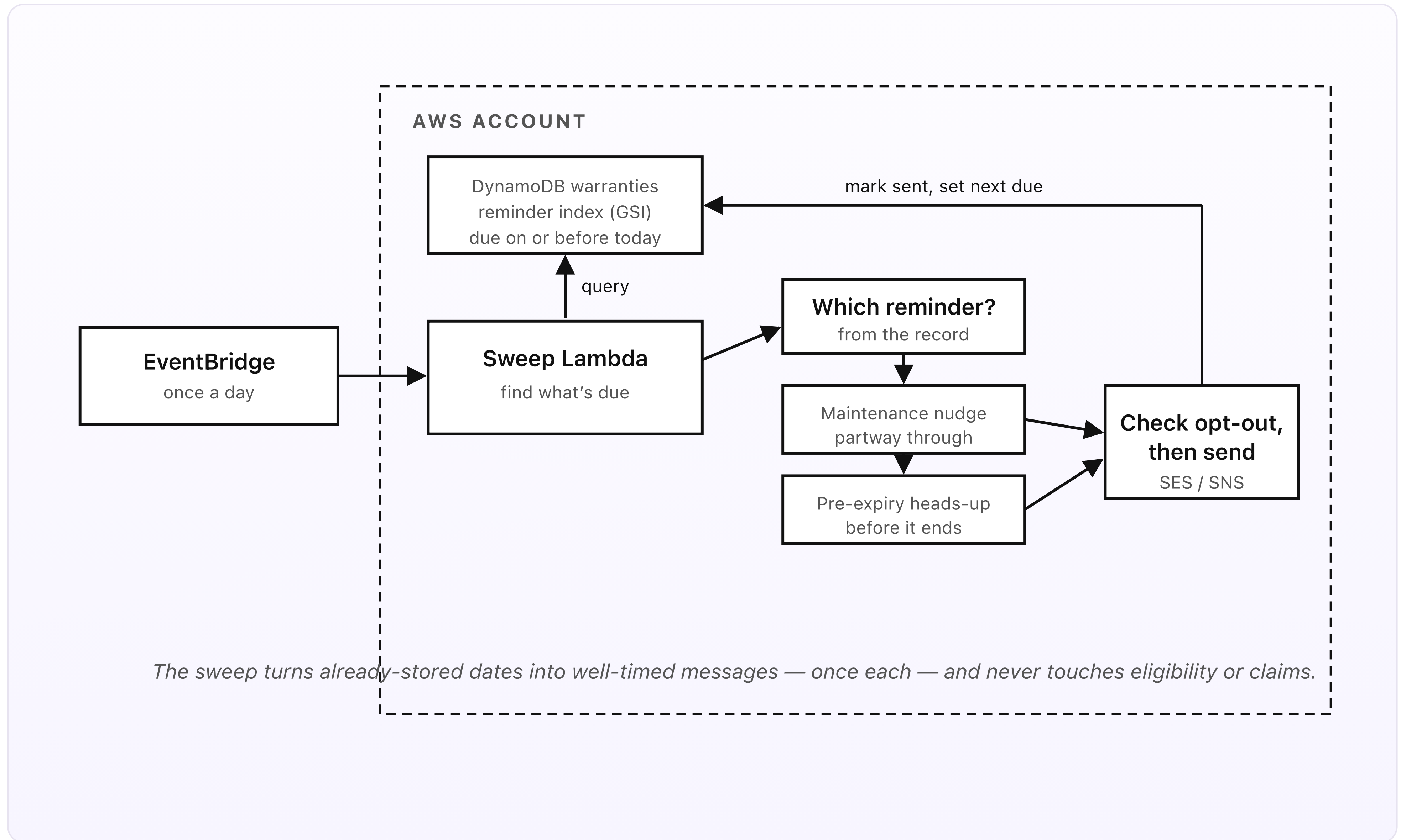


Fig 5. The reminder sweep. EventBridge Scheduler runs the sweep once a day; it queries the reminder index for warranties due, decides which nudge each earns, checks the opt-out list, sends via SES or SNS, and marks the reminder sent and the next one due

— so each fires exactly once.

Once each, and never a nag

The one thing a reminder system must never do is become a pest, so two guardrails sit on every send. The first is exactly-once: when the sweep sends a reminder, it writes back to the record marking that reminder done and setting the next due date — from maintenance to pre-expiry, and after pre-expiry to none. The next day's run therefore never re-sends the same nudge, even though it re-queries the same index; the "sent" marker is what makes the sweep idempotent, exactly as the escalated flag did in the reference system. If a run half-finishes and repeats, a reminder already marked sent is simply skipped.

The second is opt-out. Every send checks the same suppression list the confirmation respected: a customer who's asked to stop hearing from the business is never sent a maintenance or expiry reminder, full stop. And the reminders are spaced by design — a service nudge and a single pre-expiry heads-up across a multi-year term is a light touch, not a drip campaign. The system is trying to be useful at two well-chosen moments, not to fill the customer's inbox. Anything more — a renewal offer, an upsell — is a decision for a person, not something the sweep does on its own.

Why a sweep, and not a timer per warranty

You could imagine scheduling a one-off reminder for each warranty at registration — set a timer for the service date, another for the expiry date, fire each when it comes. It works, but a brand with tens of thousands of live warranties would be

juggling hundreds of thousands of individual scheduled entities stretching years into the future, each one a thing to create, amend if the record changes, and clean up. A single daily sweep over a due-date index is far simpler and cheaper: one rule, one function, one query that scales to whatever volume the business has reached, and idempotent by design because the “sent” marker means re-running it never double-messages. The dates live with the data they belong to, on the record, rather than scattered across a scheduler, which also means a correction to a warranty automatically corrects its reminders — there’s no separate timer to remember to update.

And like everything else in the system, the sweep stays firmly in its lane. It reads warranties and their reminder dates, checks the opt-out list, sends a message, and marks the record. It doesn’t decide eligibility, it doesn’t touch claims, and it never messages anyone the confirmation path wouldn’t have. That narrowness is what makes it safe to run unattended, every day, for as long as the warranties it’s watching over remain live.

DESIGN RULES THAT SHAPED THE REMINDERS

- Act on a date arriving. A reminder is triggered by the calendar, not by the customer, so only a scheduled sweep can see it.
- A clock, not a trigger. EventBridge Scheduler runs the sweep daily with no always-on compute and effectively no cost.
- Query the index, don’t scan. The due-date GSI returns just what’s due today, however many warranties are stored in total.
- Send each reminder once. A “sent” marker on the record makes the sweep idempotent — no nudge ever goes twice.
- Opt-out is sacred, and the touch is light. Two well-chosen nudges across the term, and none at all to anyone who opted out.
- One sweep beats many timers. A single daily query is simpler and cheaper than a scheduled entity per warranty, and corrects itself.

PART 6 OF 7

JULY 6, 2026 PART 6 OF 7 · WARRANTY REGISTRATION HANDLER SERIES ~6 MIN READ

What the warranty registration handler costs

A system that costs more than the goodwill it earns is a gadget. This post is the cost breakdown: every AWS service this design touches, what each adds up to at around 200 registrations a month, and why the total lands near $1.80 — plus what happens to the bill when the volume goes up tenfold.

KEY TAKEAWAYS

- About $1.80/month at roughly 200 registrations, and the fixed cost is almost nothing — nothing runs when nobody's registering.
- The two biggest moving lines are one small Bedrock call per confirmation and the outbound reminder messages. Everything else is cents.
- The only real fixed cost is Secrets Manager: two secrets at $0.40 each, billed whether or not a single product is registered.
- At ten times the volume (around 2,000 registrations) the bill lands near $9 — 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 registrations and no idle bill. You pay for a registration only when one happens, and for a reminder only when one goes out. At a typical small-brand volume — call it 200 registrations a month, each getting one confirmation, with a spread of maintenance and pre-expiry reminders firing from the back catalogue of live warranties — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two secrets — form signing key, order-lookup API key ($0.40 each)	$0.80
Bedrock (Claude Haiku 4.5)	One compose call per confirmation and per reminder (~200 + reminders)	$0.35
SNS (SMS)	Reminder texts for buyers who chose SMS, plus a few claim alerts	$0.25
DynamoDB (on-demand)	Serials, warranties + expiry index, orders mirror, audit — small reads and writes	$0.12
CloudWatch Logs	Function logs, 7-day retention	$0.10
SES	Confirmation and reminder email, and claim handovers to your team	$0.08
Lambda (Python 3.14, arm64)	Register, verify, store, notify, escalator, sweep, orders-sync	$0.05
SQS + DLQ	Buffering between the form and the slower verify, store, and model calls	$0.03
EventBridge Scheduler	The daily reminder sweep and the orders sync	$0.02
AWS Budgets	Cost alarm (first two budgets are free)	$0.00

AWS service	What it does here	Monthly
Total	**~200 registrations/month**	**$1.80**

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 nearly half the total at this volume. Everything else is genuinely usage-priced and rounds to zero at idle. The lines that move with volume are the two that actually reach the customer: the one small Bedrock call that phrases each confirmation and reminder, and the messages themselves. All the machinery doing the real work — receiving, verifying against orders, computing expiries, storing records, sweeping for due dates — is plain Lambda and DynamoDB, and together it costs less than the messaging.

The line that isn’t purely AWS

The SMS line deserves a caveat. Most warranty confirmations and reminders go by email, which is why SES sits so low and SNS is modest — SMS is only for the buyers who asked for it. But 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 $0.25 here is a UK-leaning estimate for the handful of texts a month at this volume; your real number will track your country and your customers’ preferences. If you send reminders by email only, this line all but disappears. Either way, SMS is 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 busier brand — 2,000 registrations a month, ten times the volume, and a correspondingly larger back catalogue of warranties throwing off reminders — and the bill lands near $9, not $18. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules stay at a couple of cents, and AWS Budgets stays free. What scales is the genuinely usage-priced work — roughly $3.50 of Bedrock for ten times the confirmations and reminders, about $2.50 of SMS, and a few dollars more spread across DynamoDB, logs, SES, and Lambda. Even then, the two things that reach the customer dominate, and all the machinery in between stays close to free.

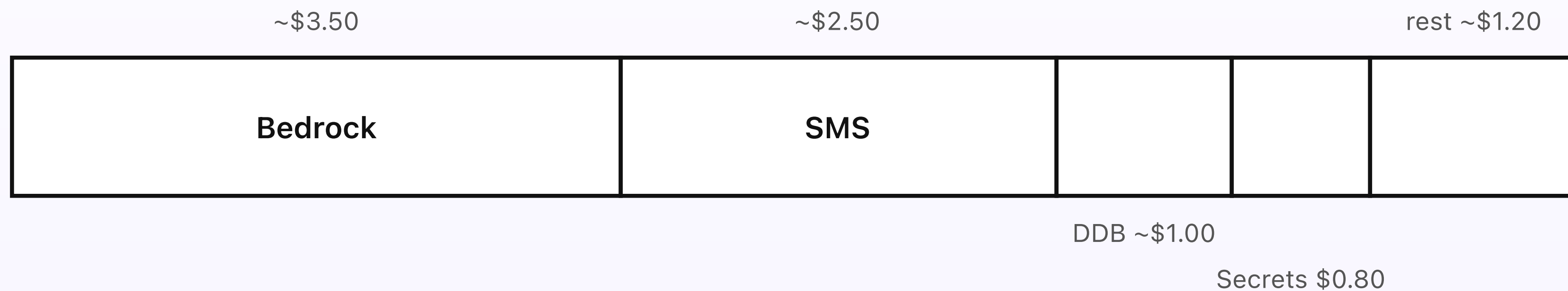


Fig 6. The monthly bill at ten times the base volume, about 2,000 registrations. Bedrock and SMS are the bulk of it; the fixed lines stay put, so the total grows sub-linearly — near $9, not ten times $1.80.

The honest way to read this: the AWS bill is rounding error against what a registered warranty is worth. A single customer you can reach for a recall, bring back for a paid service, or handle a claim for without an argument is worth far more than $1.80, and this design captures them by the hundred. Even at $9 a month for a busy brand, the system pays for itself the first time a clean, verified

record settles a claim that would otherwise have been a stand-off — and every customer it registers is one the business can actually find again.

DESIGN RULES THAT SHAPED THE COST

- Pay per registration, not per hour. No always-on compute means no idle bill.
- Spend the model sparingly. One Haiku call per confirmation or reminder, and only to phrase — never to verify or to decide.
- Cheap work stays cheap. Verifying, computing expiries, storing, and sweeping 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 SMS line. It's the part whose price varies by country, so the Budgets alarm sits right on top of it.

PART 7 OF 7

JULY 6, 2026 PART 7 OF 7 · WARRANTY REGISTRATION HANDLER SERIES ~10 MIN READ

Engineering reference: the warranty registration handler architecture

This is the warranty registration handler with the friendly labels removed: the real resource names, the runtime, the serial ledger and the expiry index, the single public Function URL, the reminder schedule, and the IAM scope. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Seven Lambda functions, all Python 3.14 on arm64, wired through one SQS queue with a dead-letter queue.
- One public surface: a single Lambda Function URL on `wrhr-register` that takes the QR post and the web form — no API Gateway.
- Five DynamoDB tables, all on-demand: the warranties record (keyed by serial), the submission ledger, the orders mirror, the opt-out list, and an append-only audit log.
- The warranties table carries a reminder GSI keyed by due date; one EventBridge Scheduler drives the daily reminder sweep and the orders sync.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by the notifier. 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, customer messages go out through SES and SNS, and work is buffered on a single SQS queue.

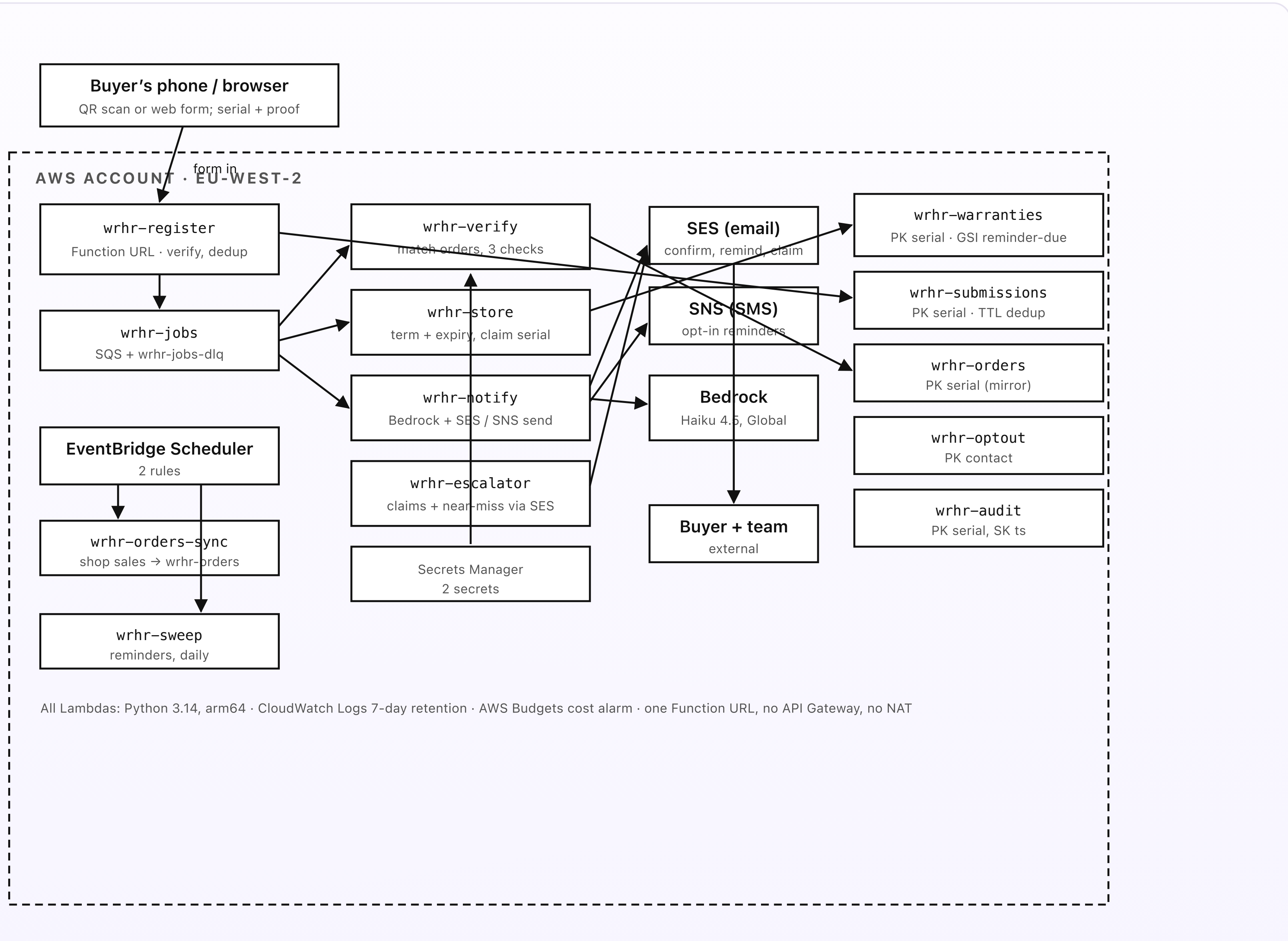

Buyer's phone / browser
QR scan or web form; serial + proof
form in
AWS ACCOUNT · EU-WEST-2
wrhr-register
Function URL · verify, dedup
wrhr-jobs
SQS + wrhr-jobs-dlq
EventBridge Scheduler
2 rules
wrhr-orders-sync
shop sales → wrhr-orders
wrhr-sweep
reminders, daily
wrhr-verify
match orders, 3 checks
wrhr-store
term + expiry, claim serial
wrhr-notify
Bedrock + SES / SNS send
wrhr-escalator
claims + near-miss via SES
Secrets Manager
2 secrets
SES (email)
confirm, remind, claim
SNS (SMS)
opt-in reminders
Bedrock
Haiku 4.5, Global
Buyer + team
external
wrhr-warranties
PK serial · GSI reminder-due
wrhr-submissions
PK serial · TTL dedup
wrhr-orders
PK serial (mirror)
wrhr-optout
PK contact
wrhr-audit
PK serial, 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 warranty registration handler drawn for engineers: one Function URL on `wrhr-register`, an SQS-buffered set of seven Lambdas, five DynamoDB tables with a reminder GSI on `wrhr-warranties`, Bedrock called only by the notifier, SES and SNS for messages, and two scheduled jobs. One region, one account, no API Gateway.

Lambda functions

Seven 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 (`wrhr-jobs`, with `wrhr-jobs-dlq` as its dead-letter queue after five attempts) decouples the public form from the slower verification, store, and model calls, and carries a typed job (`verify`, `store`, `confirm`, `remind`) from one stage to the next.

- `wrhr-register` — the only public surface. Backs the single Lambda Function URL and receives both the QR post and the typed form. Verifies the signed token against the secret, rate-limits per source, de-duplicates against `wrhr-submissions` with a conditional write on the serial (TTL a few minutes), normalises the serial, and enqueues a `verify` job. Nothing slow happens here — no order lookup, no compute, no send.
- `wrhr-verify` — SQS-triggered on `verify` jobs. Looks the serial up in `wrhr-orders`, confirms the sale exists, checks `wrhr-warranties` for an existing record, and tests the purchase date against the product's registration window. A pass enqueues a `store` job; a fail records the reason; a genuine near-miss (a receipt with no clean match) goes to `wrhr-escalator`.
- `wrhr-store` — SQS-triggered on `store` jobs. Reads the warranty rule for the product line, computes the term and exact expiry dates from the verified

purchase date, and claims the serial with a conditional `PutItem` into `wrhr-warranties` (`attribute_not_exists(serial)`), populating the reminder GSI and writing `wrhr-audit`. On success it enqueues a `confirm` job.

- `wrhr-notify` — SQS-triggered on `confirm` and `remind` jobs. Makes the single Bedrock call, validates the draft (right dates, sane length, template fallback), checks `wrhr-optout`, and sends via SES (email) or SNS (SMS). It writes the send to `wrhr-audit`. This is the only function that calls Bedrock.
- `wrhr-escalator` — builds the handover (serial + verified sale + computed coverage + reason), de-duplicates against open cases, and emails the support inbox via SES. This is the lane for later claims and for near-miss confirmations.
- `wrhr-orders-sync` — scheduled. Pulls the shop's sales (platform API, dealer feed, or export) and upserts rows into `wrhr-orders` keyed by serial.
- `wrhr-sweep` — scheduled. Queries the `reminder-due` GSI on `wrhr-warranties` for reminders due on or before today and not yet sent, enqueues a `remind` job for each, and marks the reminder sent and the next one due so it's only ever fired once.

Data stores, schedules, and messaging

- **DynamoDB (all on-demand).** `wrhr-warranties` — PK `serial`; one item per registered unit with its coverage, the computed expiry dates, the reminder schedule and "sent" markers, and a state field; a GSI `reminder-due` (PK `due_date`, SK `due_ts`) is the sweep's query path. The conditional write on `serial` is what guarantees one warranty per unit. `wrhr-submissions` — PK `serial`, short-lived with a TTL, the dedup ledger for the register step. `wrhr-orders` — PK `serial`, the sales mirror holding purchase date and channel.

`wrhr-optout` — PK `contact` (email or E.164), checked before every send.
`wrhr-audit` — PK `serial`, SK `ts`, append-only, holding each confirmation and reminder and the facts it was built from.

- **Function URL.** One, on `wrhr-register`, with signed-token and signature verification in-function; `AuthType NONE` at the edge because authenticity is enforced by the shared secret and rate limit, not by IAM. No API Gateway.
- **SES and SNS.** SES sends confirmations, reminders, and claim handovers from a verified domain with DKIM; SNS sends SMS reminders to the buyers who opted for them. Both go through `wrhr-notify` (customer) or `wrhr-escalator` (team).
- **EventBridge Scheduler.** Two rules — `wrhr-sweep` at `rate(1 day)` for the reminder sweep, and `wrhr-orders-sync` at `rate(1 hour)` to keep the orders mirror current so a sale can be registered soon after it's made.
- **Secrets Manager.** Two secrets — the form signing key and the order-lookup API key — fetched at call time, never in env vars or the settings doc.
- **Bedrock.** Model id `anthropic.claude-haiku-4-5` via the Global cross-Region inference profile, invoked only by `wrhr-notify`.

Idempotency and failure handling

Exactly-once registration rests on one thing: the conditional write in `wrhr-store`. Because `PutItem` with `attribute_not_exists(serial)` can succeed only for the first writer, the entire pipeline behind it can be at-least-once without risk. SQS may deliver a job twice, a Lambda may retry after a timeout, two submissions may race — and at most one of them creates the warranty; the rest see the condition

fail and treat it as "already registered". The earlier dedup in `wrhr-register` is a cheap first filter, not the guarantee. Jobs that keep failing — a malformed payload, an order feed that's down — land in `wrhr-jobs-dlq` after five attempts, where they can be inspected and replayed rather than lost or retried forever. The reminder sweep is idempotent for the same reason on the read side: a reminder marked `sent` is skipped, so a re-run never double-messages.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `wrhr-register` can read the signing secret, conditionally write `wrhr-submissions`, and send to `wrhr-jobs` — it cannot read orders, call Bedrock, or send a message. `wrhr-verify` can read `wrhr-orders` and `wrhr-warranties` and enqueue, nothing more. `wrhr-store` can conditionally write `wrhr-warranties` and append to `wrhr-audit`, but cannot delete from any table and cannot send. `wrhr-notify` is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile; it can publish to SNS and send via SES and read `wrhr-optout`, but touches no order data. `wrhr-escalator` can send via SES and read its inputs only. The scheduled functions hold the narrow permissions they need — `wrhr-orders-sync` writes only `wrhr-orders`, `wrhr-sweep` reads the reminder GSI and enqueues — and neither has an inbound surface. 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 the line most likely to move, it's the cheapest early warning that volume (or a loop) is running hot.

That's the whole system: a QR scan or a short form, verified against real orders, computed into a dated record that can only ever be written once, explained in plain language, and diarised so the customer hears back at exactly the right moment — with a person on the one decision, a claim, that should never be automated. Seven small functions, five tables, one model call per message, and no always-on anything, for a couple of dollars a month.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Seven small Lambdas beat one that does everything; the queue decouples the slow calls from the public form.
- One public surface. Only `wrhr-register` is reachable from outside, on a single Function URL, authenticated by a signed token.
- One conditional write is the guarantee. `attribute_not_exists(serial)` in `wrhr-store` makes registration exactly-once under any retry or race.
- Least privilege, per role. Only the notifier can call Bedrock; only verify reads orders; only store writes warranties.
- Dates live with the data. The expiry and reminder dates sit on the record and drive the GSI the sweep queries — no separate timers.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called once per message.