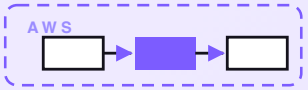
7-PART SERIES · FREE COMPANION



Return and RMA handler

A customer wants to send something back, and what follows is the same dull routine every time — check it's still in the return window, issue an RMA number, print a prepaid label, wait for the parcel, inspect it, then decide between a refund and a replacement. This is the design of a small serverless system that runs that whole returns desk end to end: it checks the request against your policy, issues the RMA and the label, tracks the parcel on its way back, and once it's received proposes refund or replacement with the reason attached. The eligibility and policy checks are plain code, and a human approves the money or the swap before anything moves. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Return and RMA handler

PART 1 OF 7

JUNE 28, 2026 PART 1 OF 7 · RETURN AND RMA HANDLER SERIES ~10 MIN READ

A return and RMA handler on AWS for a few dollars a month

A customer asking to return something kicks off a surprising amount of dull, error-prone work: check the policy, issue an RMA number, generate a prepaid label, wait for the parcel, inspect what comes back, then decide whether they get their money back or a replacement. Miss the return window and you eat a return you didn't owe; lose track of the parcel and you refund something that never arrived. This post walks through the design of a small system that runs the whole returns desk as one orchestrated flow — and never moves money on its own.

KEY TAKEAWAYS

- The whole returns desk runs as one Step Functions state machine: request → eligibility → RMA & label → await inbound → inspect → decide.
- Eligibility is plain code against your policy — return window, non-returnable flags, condition, and prior returns.
- It issues a real RMA number and a prepaid carrier label, then tracks the inbound parcel by its tracking number.
- Once the parcel is received and inspected, it proposes refund *or* replacement with the reason attached.
- Designed on AWS for about $2.90/month at roughly 120 returns a month. A human approves every refund or replacement.

The whole system on one page

Before any code, here's the shape of what we're designing. A return isn't a single action — it's a small process that plays out over days: someone asks, you check the policy, you issue a number and a label, the parcel travels back, somebody opens the box, and only then does anyone decide whether it's a refund or a replacement. The trouble is that each step usually lives in a different place — an email, a courier portal, a spreadsheet, somebody's memory — and the gaps between them are where returns get lost, refunded twice, or refunded for goods that never came back. The system below holds the whole process in one place and walks every return through the same stages.

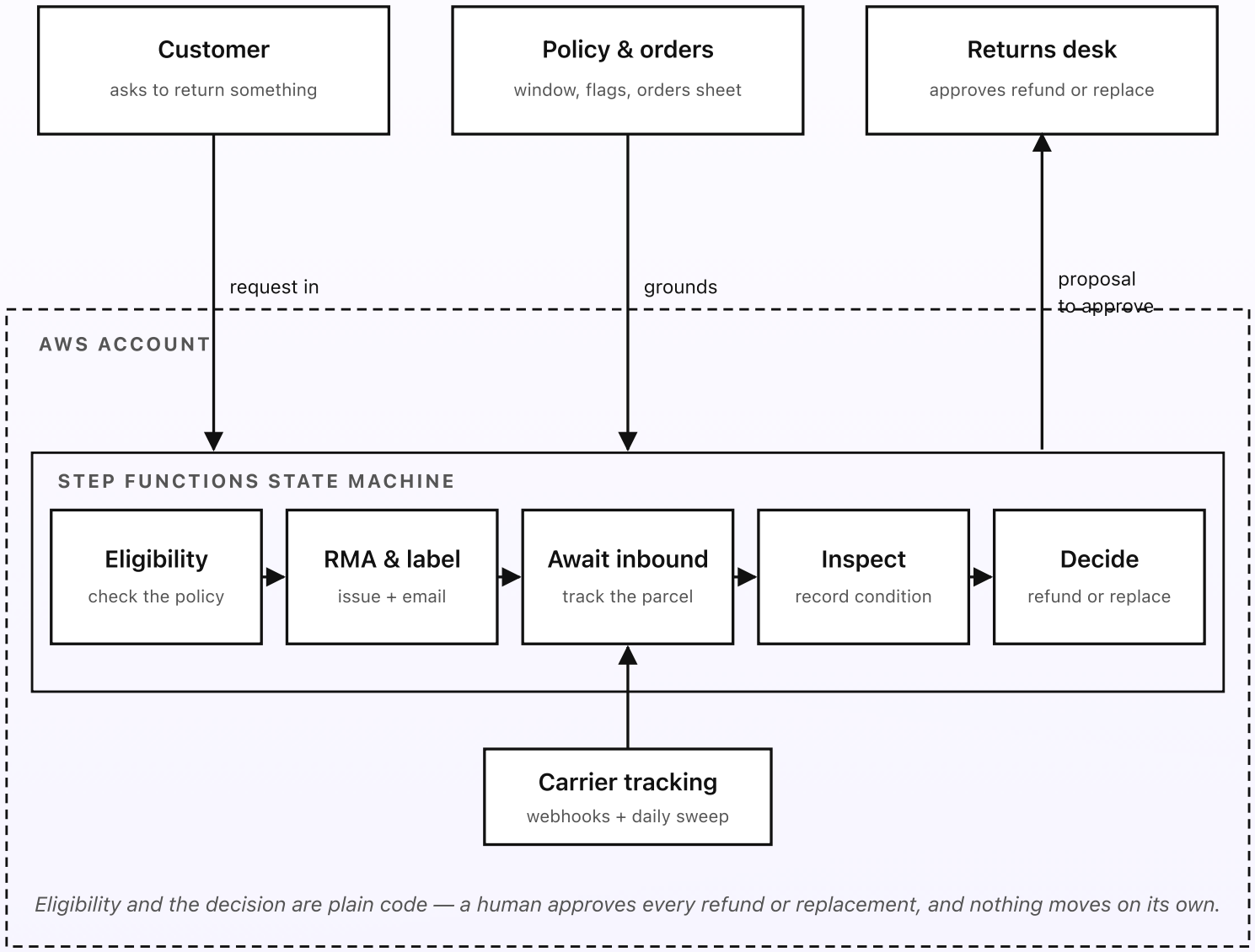

Customer
asks to return something
Policy & orders
window, flags, orders sheet
Returns desk
approves refund or replace
request in
grounds
proposal to approve
AWS ACCOUNT
STEP FUNCTIONS STATE MACHINE
Eligibility
check the policy
RMA & label
issue + email
Await inbound
track the parcel
Inspect
record condition
Decide
refund or replace
Carrier tracking
webhooks + daily sweep
Eligibility and the decision are plain code — a human approves every refund or replacement, and nothing moves on its own.

Fig 1. Three things outside, one state machine inside. The customer's request, the policy and orders it's checked against, and the returns desk that approves. Inside AWS, every return walks the same five stages, with the carrier's scans waking the wait.

What you set up once (the outside)

- **Policy and orders.** A Google Sheet in a Drive folder with one row per order: order number, customer name and email, the items and their prices, the purchase date, and the sales channel. You already keep this in whatever you sell through; this just puts it where the handler can read it. Alongside it, a short returns-policy doc holds the rules the system enforces — the return window (say 30 days from delivery), the non-returnable flags (final-sale, hygiene, perishable), the conditions you accept back, and the rule for when a return should be a refund versus a replacement. Changing the window or a flag is a doc edit, not a deploy.
- **How returns come in.** A customer starts a return one of three ways, covered in Part 2 — a returns form on your site that posts to a Lambda Function URL, a reply to an order email that lands through SES, or a manual start by your own team for a phone request. Whatever the lane, it becomes one normalised request: which order, which item, and the customer's stated reason.
- **The returns desk.** The person who signs off the outcome — usually whoever runs customer service or the warehouse. They never have to chase a parcel or look up a policy; they get a single email when a return is ready to resolve, with the inspection result and the proposed outcome, and three buttons: *Approve*, *Switch* (refund instead of replace, or the reverse), and *Reject*. A tap records the decision — it never reaches into your payment system on its own.

What runs on every return (the inside)

Every return is one execution of a Step Functions state machine, and it moves through the same five stages in order.

- **Eligibility.** The first stage finds the original order and runs a short list of plain-code checks against the policy: is it inside the return window, is the item flagged non-returnable, has it already been returned, is the stated condition allowed. Every check must pass. A fail is declined with the exact reason, or escalated where policy says a person should decide. This is Part 2.
- **RMA & label.** A cleared return gets a unique RMA number and a prepaid shipping label from the carrier's label API, with the key pulled from Secrets Manager. The label PDF is stored in S3, and the customer gets one email with the RMA number, the label, and exactly what to do. This is Part 3.
- **Await inbound.** The execution then pauses — cheaply, for as many days as it takes — holding a task token and the tracking number. A real carrier scan, arriving as a webhook, wakes it when the parcel is delivered back. A daily sweep catches the parcels that never ship. This is Part 4.
- **Inspect.** When the box arrives, a staffer opens it and records what they find — matches the order, sealed, used, damaged in transit — with a photo or two stored in S3. That recorded condition, not a guess, is what the decision runs on. This is Part 5.
- **Decide.** The last stage applies the policy in plain code and proposes one outcome — full refund, partial refund, replacement, or reject — with the reason attached, and sends it to the returns desk to approve. This is also Part 5.

In plain words

A customer fills in the returns form on Tuesday: order 4821, one pair of boots, “too small.” The handler finds order 4821 — delivered nine days ago, well inside the 30-day window, boots not flagged non-returnable — and clears it. It issues RMA `RMA-4821-7`, calls the carrier for a prepaid label, drops the PDF in S3, and emails the customer: “Here’s your return label and RMA number; pop the boots back in the box and drop it at any pickup point.” The execution pauses. Six days later the carrier scans the parcel as delivered to your warehouse; the webhook wakes the execution and emails the warehouse to inspect it. A staffer checks the boots back in — unworn, tags on, matches the order — and records it. The decide stage applies the policy: returned in time, correct item, resaleable, customer wanted a different size that’s in stock, so it proposes a *replacement*, not a refund, and sends it to the returns desk. The manager taps *Approve*; the replacement is queued and the customer is told it’s on the way. Nothing was lost, nothing was refunded by mistake, and the only human decision was the one that mattered.

The cost of running this is about $2.90 a month at roughly 120 returns. The cost of *not* running it is the refund issued for a parcel that never came back, the return waved through after the window because nobody checked the date, and the customer left waiting a week for a label nobody got round to.

DESIGN RULES THAT SHAPED EVERY DECISION

- One return, one execution. The whole process — request to outcome — is a single Step Functions run you can see the state of at any moment.
- Plain code decides what's allowed. Eligibility and the refund-or-replace decision are deterministic checks against a written policy, not a model.
- It owns the logistics, not just the money. RMA, label, and inbound tracking are part of the system — not a separate courier portal.
- Track the real parcel. The wait ends on an actual carrier scan, never on a guess that the box probably arrived.
- A human approves the outcome. Every refund and every replacement is signed off by a person, with the reason in front of them.
- Policy lives in a doc. The window, the flags, and the rules change without a deploy.

Why this shape

Most small teams run returns as a chain of disconnected steps. Someone emails a label from the courier site, someone else watches for the parcel, and a third person — days later, with no memory of the original request — decides what to do with whatever turned up. The gaps are expensive: a refund goes out before the item is back, a return slips through after the window because nobody re-checked the date, or a parcel is "received" on trust and the refund is keyed against a box that's still in a depot somewhere. None of these are hard problems individually;

they're just easy to drop when the steps live in different places and different heads.

Modelling the whole thing as one state machine fixes the gaps by construction. The return can't reach the decision before the parcel is genuinely received, because the decide stage is downstream of a wait that only ends on a real scan. It can't be approved without an inspection, because inspect comes first. And it can't be refunded by accident, because the only thing the machine ever does on its own is *propose* — a human approves the money or the swap. The policy that drives it all sits in a doc you can edit, so the rules stay yours.

The next four posts walk through each stage in turn: how a request gets checked, how an RMA and label get issued, how an inbound parcel gets tracked, and how a refund or replacement gets decided. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 28, 2026 PART 2 OF 7 · RETURN AND RMA HANDLER SERIES ~6 MIN READ

How a return request gets checked

Before a single label is printed, the system has to answer one question honestly: is this return even allowed? This post is about that step alone — how a request is matched to its original order and run through a short list of plain-code policy checks, and what happens, deliberately, when one of them fails.

KEY TAKEAWAYS

- A request is matched to its original order first — by order number, or by the customer's email and the item.
- Eligibility is a short list of plain-code checks: return window, non-returnable flags, prior returns, and condition.
- Every check must pass. There is no "probably fine" — a single fail stops the return.
- A fail is declined with the exact reason, or escalated to a person where policy says a human should decide.
- No model runs here. The rules come from your written policy doc and are applied as code.

The question before the label

It's tempting to issue a label the moment someone asks for one — it feels like good service. But a label issued for a return you don't actually owe is a cost you've volunteered for: the postage, the handling, and often the refund at the end of it. The job of this stage is to answer one question honestly before any of that happens: *is this return allowed under your own policy?* Everything downstream — the RMA, the label, the tracking, the refund-or-replace decision — assumes the answer is yes, so this is the gate that protects the rest.

The stage is a single Lambda, `rmar-eligibility`, and it's the first task in the state machine. It does two things in order: find the order the request is about, then run the policy checks against it.

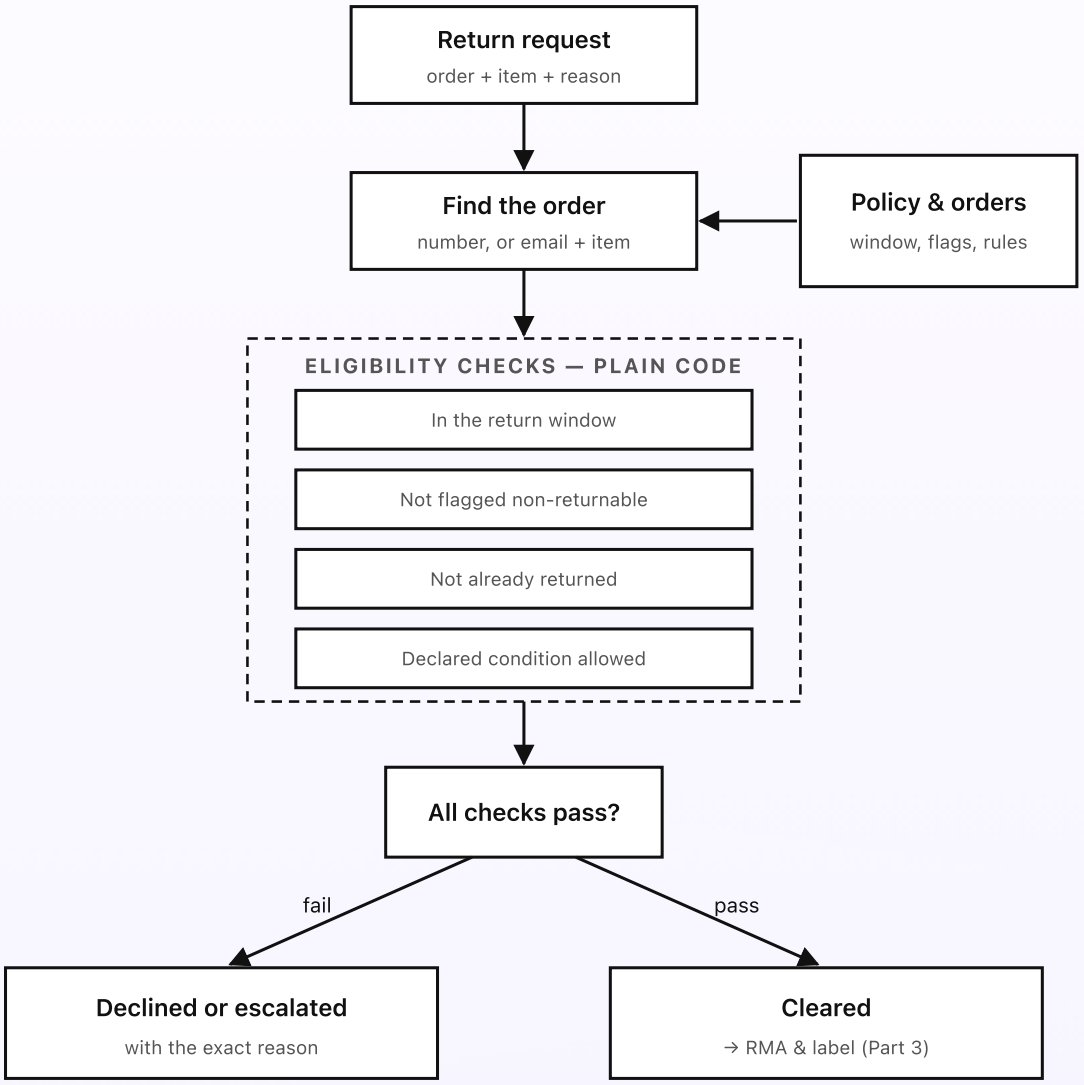
Finding the order

A return request carries whatever the customer typed: ideally an order number, but sometimes just "the boots I bought last month." The stage looks first for an order number in the request — the cleanest signal — and matches it against the orders mirrored from your Drive sheet. If there isn't one, it falls back to the customer's email plus the item described. A confident single hit moves on; an ambiguous one (several orders that could fit) or no match at all is escalated to a person rather than guessed, because every check that follows depends on having the right order underneath it.

The checks, in plain code

With the order in hand, the stage runs the policy as a short list of deterministic checks. Each is a clear pass or fail, and the request only proceeds if every one

passes.



No model runs here. The rules come from your written policy and are applied as code.

Fig 2. The eligibility gate. Find the order, run four plain-code checks against the policy, and either clear the return or decline it with the exact reason. One fail is enough to stop it.

What each check actually does

- **In the return window.** The window runs from delivery, not from purchase, so the stage takes the delivery date carried on the order and compares it to today against the window in the policy — commonly 30 days. A boot delivered nine days ago passes; one delivered 40 days ago fails, and the decline says exactly that: “returns close 30 days after delivery; this was delivered on 12 May.”
- **Not flagged non-returnable.** Some items can’t come back at all — final-sale, hygiene items once opened, perishables, custom-made goods. The catalogue carries a non-returnable flag per item; if it’s set, the return stops here with the reason named, rather than a label going out for something you’d never accept.
- **Not already returned.** The returns table is checked for an existing return on the same order line. This is what stops a customer — or an honest mistake — opening a second return on an item that’s already been refunded or replaced once.
- **Declared condition allowed.** The request asks the customer to state the condition (unopened, opened, faulty). The policy says which conditions you accept for which reasons — a change-of-mind return may need to be unopened, while a faulty item can come back used. A condition the policy doesn’t allow for the stated reason is flagged for a person rather than auto-declined, because “faulty” deserves a human read.

Why this stays plain code

There's an obvious temptation to let a model read the request and judge eligibility "intelligently." It's the wrong tool here. Eligibility is a question with a right answer that your policy already defines: the date is either inside the window or it isn't, the flag is either set or it isn't. A model would add cost, latency, and a small but real chance of waving through a return it shouldn't — or declining one it should — for reasons you can't audit. Plain code against a written policy gives you the opposite: every decision is explainable, every decline cites the rule it failed, and changing a rule is a doc edit. The only place judgement genuinely helps — the phrasing of the decline email, or a borderline "faulty" claim — is handed to a model for words or to a person for the call, never to a model for the verdict.

DESIGN RULES FOR THE ELIGIBILITY STAGE

- Order first, checks second. Everything depends on matching the right order; an unsure match escalates rather than guesses.
- Every check must pass. There is no partial credit — a single fail stops the return.
- Decline with the reason. A rejected return names the rule it failed, in words the customer can act on.
- The window runs from delivery. The clock starts when the customer got the item, not when they paid.
- Faulty gets a human. A condition the policy can't auto-accept is escalated, not auto-declined.

PART 3 OF 7

JUNE 28, 2026 PART 3 OF 7 · RETURN AND RMA HANDLER SERIES ~6 MIN READ

How an RMA and label get issued

Once a return clears the policy checks, the customer needs two things to actually send the item back: a reference number that ties the parcel to their return, and a label they can stick on the box. This post is about issuing both — cleanly, once, and with everything the customer needs in a single email.

KEY TAKEAWAYS

- A cleared return gets a unique RMA number that ties every later step — parcel, inspection, decision — back to it.
- The prepaid label comes from the carrier's label API, called with a key held in Secrets Manager, never in code.
- The label PDF and the tracking number are stored in S3 and on the return, so nothing depends on the carrier's portal later.
- The customer gets exactly one email: the RMA number, the label, and plain instructions for sending the item back.
- The stage is idempotent — a retry never issues a second label or a second RMA for the same return.

Two things the customer needs

Once a return is cleared, the customer needs two concrete things to actually send the item back: a reference that ties their parcel to their return, and a label they can print or show at a drop-off point. This stage — the Lambda `rmar-label`, the second task in the state machine — produces both, stores everything the rest of the system will need, and sends one clear email. It runs in a fixed order so that a retry can never leave a half-issued return behind.

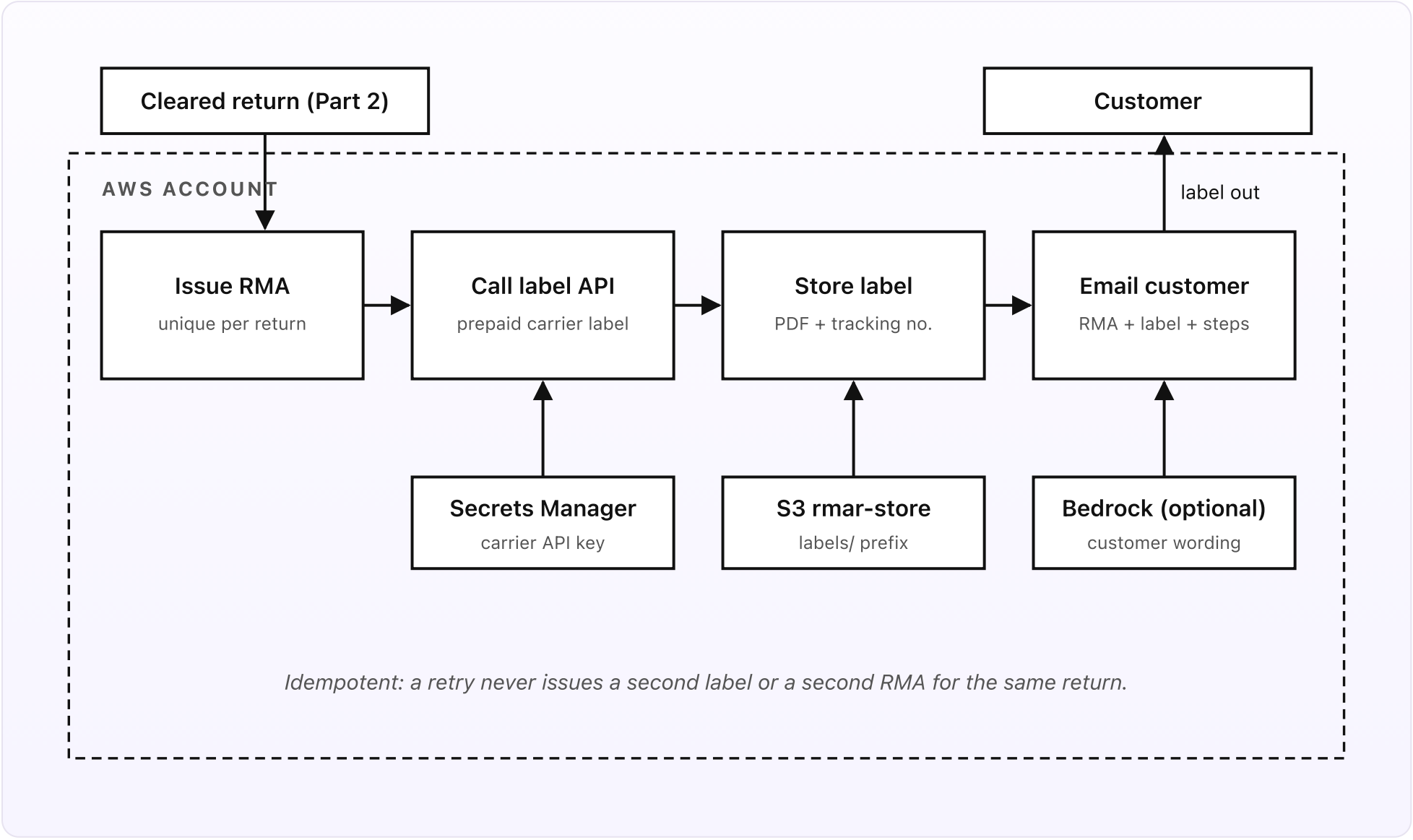


Fig 3. Issuing the RMA and label. Mint the number, call the carrier for a prepaid label with a key from Secrets Manager, store the PDF and tracking number in S3, and send the customer one email. Bedrock only ever touches the wording.

The RMA number

The RMA (return merchandise authorisation) number is the thread that runs through everything that follows. It's minted once, deterministically, from the return: something like `RMA-4821-7` — the order number, then a per-order sequence so a customer returning two items from one order gets two distinct RMAs. It becomes the partition key on the returns record, it's printed on the label and in the email, and it's what the warehouse writes on the box when it's checked back in. Because it's derived rather than random, the same return always resolves to the same RMA — which is half of what makes this stage safe to retry.

The label

The prepaid label comes from the carrier's label API — the same kind of call your shipping tool makes, just for an inbound parcel. The stage requests a return label for the item's weight and the customer's address, using an API key fetched from Secrets Manager at call time, never baked into the function or the policy doc. The carrier responds with two things worth keeping: the label PDF, which goes straight into the `labels/` prefix of the S3 bucket, and the tracking number, which is written onto the return record. That tracking number is the hook the next stage waits on, so storing it now — rather than relying on the carrier's portal later — matters. Label fees are a carrier cost on your shipping contract, not an AWS one; the system just asks for the label.

The one email

The customer gets a single message through SES: the RMA number, the label (attached and linked from S3 via a short-lived signed URL), and three plain instructions — what to pack, where to drop it, and by when. If Bedrock is enabled

it phrases that email in your voice; if it isn't, a template does the job just as well, because there's nothing here a model needs to decide — the facts are all known. One email, not three: no separate "your return is approved," "here's your number," and "here's your label," which is how customers end up confused about which message to act on.

Why issuing exactly once matters

This is the one stage that spends real money on an external service — every label API call may cost you a postage pre-pay — so it has to be safe against the thing that always eventually happens in a distributed system: a retry. If `rmar-label` times out after the carrier issued the label but before the state machine recorded success, a naive retry would buy a second label and confuse the customer with a second email. The stage avoids that by checking the return record first: if it already carries an RMA, a label key, and a tracking number, the work is done and the step simply returns the existing values. The derived RMA number and the recorded tracking number make the whole stage idempotent, so the state machine can retry it freely and the customer still gets exactly one label.

DESIGN RULES FOR THE RMA AND LABEL STAGE

- One RMA, derived not random. The number comes from the return, so the same return always maps to the same RMA.
- Keys live in Secrets Manager. The carrier API key is fetched at call time, never in code or the policy doc.
- Store the tracking number now. The inbound wait hooks onto it; don't depend on the carrier's portal later.
- One email, all the facts. RMA, label, and instructions together — not three messages to piece together.
- Idempotent by design. Already has an RMA and a label? Return them. Never buy a second label on a retry.

PART 4 OF 7

JUNE 28, 2026 PART 4 OF 7 · RETURN AND RMA HANDLER SERIES ~6 MIN READ

How an inbound parcel gets tracked

Between issuing a label and getting the box back, the system has nothing to do but wait — and waiting well is harder than it sounds. This post is about the await-inbound step: how a Step Functions execution pauses cheaply for days, how a real carrier scan wakes it back up, and how a daily sweep makes sure a parcel that never ships doesn't sit open forever.

KEY TAKEAWAYS

- After the label goes out, the state machine pauses on an "await inbound" step that can wait for days at no cost.
- The paused execution holds a task token, stored on the return and findable by the parcel's tracking number.
- A real carrier scan arrives as a webhook, is routed through EventBridge, and resumes the exact waiting execution.
- A daily EventBridge Scheduler sweep catches parcels that were never sent, so a return never sits open forever.
- The wait ends on a genuine delivered scan — never on a guess that the box probably arrived.

Waiting well

Between handing the customer a label and getting the box back, there's nothing to do but wait — and that waiting is where naive systems either burn money or lose track. You don't want a Lambda sitting spinning for a week, and you don't want to poll the carrier every hour for a parcel that hasn't moved. Step Functions handles this properly: the execution reaches an *await inbound* step, hands out a task token, and goes to sleep. It costs nothing while it sleeps — Step Functions bills state transitions, not the days a parcel spends in a van — and it wakes only when something real happens.

Two things can wake it: a carrier scan that says the parcel was delivered back to you, or a scheduled sweep that decides the parcel is never coming.

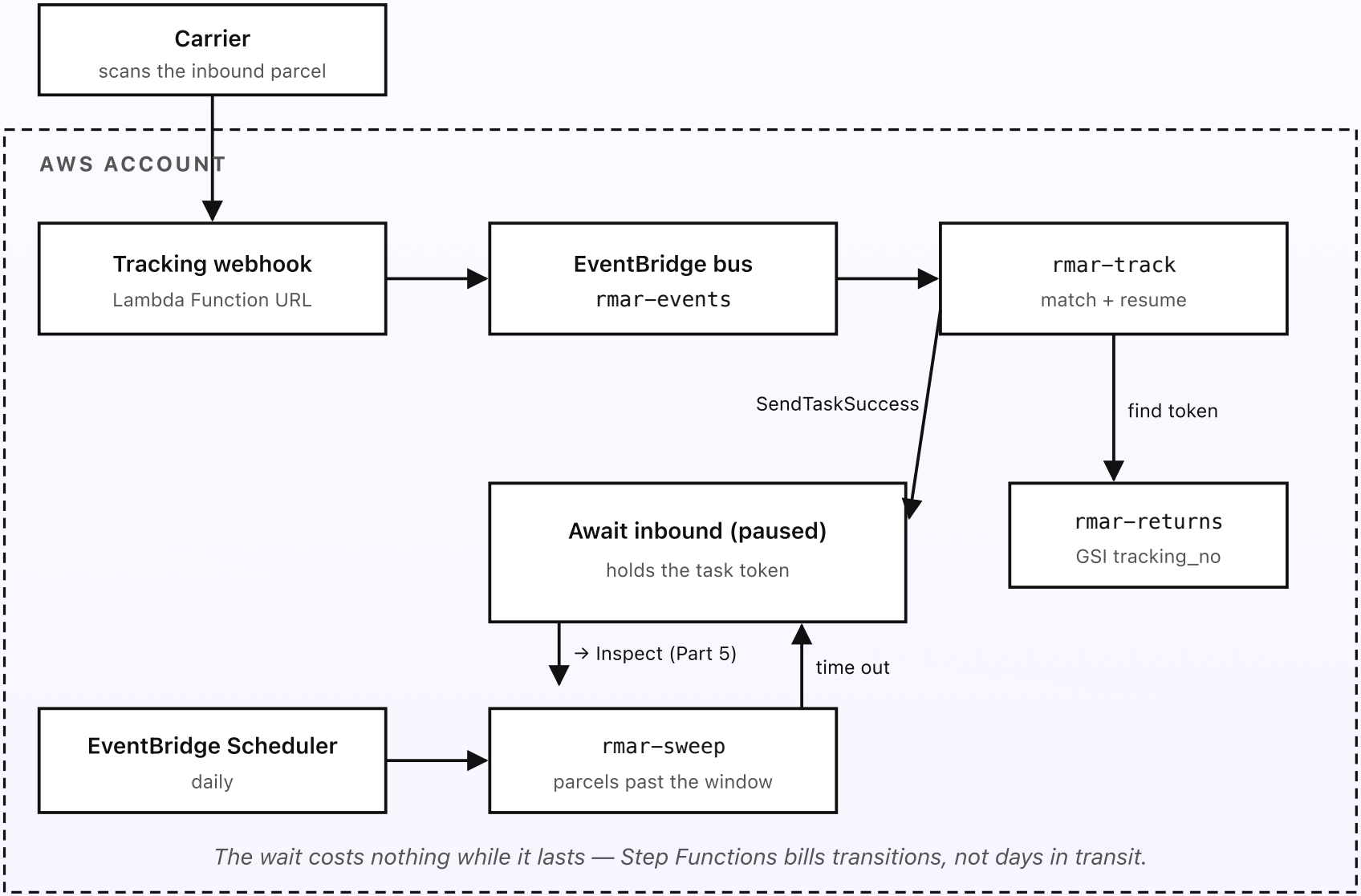

Carrier
scans the inbound parcel
AWS ACCOUNT
Tracking webhook
Lambda Function URL
EventBridge bus
rmar-events
rmar-track
match + resume
SendTaskSuccess
find token
Await inbound (paused)
holds the task token
rmar-returns
GSI tracking_no
→ Inspect (Part 5)
time out
EventBridge Scheduler
daily
rmar-sweep
parcels past the window
The wait costs nothing while it lasts — Step Functions bills transitions, not days in transit.

Fig 4. The inbound wait. A paused execution holds a task token; a real carrier scan routes through EventBridge to rmar-track, which finds the token by tracking number and resumes the exact return. A daily sweep times out parcels that never shipped.

The paused execution

When `rmar-label` finishes, the state machine moves to a task step configured to *wait for a task token*. Step Functions generates the token and the step hands it off — the function writes it onto the return record, alongside the tracking number from Part 3 — then the execution simply parks. There's no compute running, no queue being drained, nothing being polled. The whole return is now represented by one sleeping execution and one row in `rmar-returns` that says, in effect, "when tracking number `XYZ` is delivered, resume with this token."

The carrier scan that wakes it

Carriers will post tracking events to a webhook if you register one — collected, in transit, out for delivery, delivered. The handler behind the `rmar-track` Function URL takes each event, verifies it against a signing secret from Secrets Manager, and publishes it onto an EventBridge bus, `rmar-events`. A rule on that bus — matching only the events that matter, the "delivered to the return address" ones — triggers `rmar-track`. It looks up the return by tracking number using a GSI, reads the stored task token, and calls `SendTaskSuccess` with the scan details. That, and only that, resumes the execution, which moves straight on to the inspect stage. Putting EventBridge in the middle means the noisy in-transit scans are filtered out cheaply and the only thing that ever wakes a return is the scan that genuinely matters.

The sweep for parcels that never come

Plenty of returns are requested and never sent — the customer changes their mind, or the label sits in their inbox until they forget. Those executions would otherwise wait forever. `rmar-sweep`, fired daily by EventBridge Scheduler, scans for returns still parked past a sensible deadline — say 21 days after the label was issued — and resolves them: a gentle reminder a few days in, and a clean timeout that closes the return at the deadline so it isn't left dangling. A timed-out return is never refunded; it's simply closed, because nothing came back. The sweep is also where a stuck-in-transit parcel gets noticed and handed to a person, so a return that's genuinely lost in the post gets a human rather than silence.

DESIGN RULES FOR THE INBOUND WAIT

- Pause, don't poll. The execution sleeps on a task token; nothing runs and nothing is billed while a parcel travels.
- Wake on a real scan. Only a genuine delivered event resumes a return — never an assumption that it arrived.
- Filter at EventBridge. In-transit noise is dropped by the rule; only the delivered scan reaches the resume function.
- Find it by tracking number. The GSI on the returns table ties a scan back to the one execution waiting on it.
- Sweep the strays. A daily timeout closes returns that never shipped — closed, not refunded — and flags the genuinely lost.

PART 5 OF 7

JUNE 28, 2026 PART 5 OF 7 · RETURN AND RMA HANDLER SERIES ~6 MIN READ

How a refund or replacement gets decided

The parcel is back and someone has opened the box. Now comes the only decision that actually costs money: does the customer get a refund, a replacement, or neither? This post is about how the system turns an inspection result into a clear proposal — with the reason spelled out — and hands it to a person to approve.

KEY TAKEAWAYS

- The decision runs on a recorded inspection — what the box actually contained — not on what the customer said they'd send.
- A staffer logs the condition and a photo or two; that result, stored in S3, is what the decision is built on.
- The decide step is plain code: it proposes full refund, partial refund, replacement, or reject — with the reason attached.
- The proposal goes to the returns desk with three buttons: Approve, Switch, or Reject. A human signs off before anything moves.
- Once approved, the outcome is recorded and the customer is told. No refund or replacement leaves on its own.

The only decision that costs money

Everything so far has been preparation. The parcel is back, someone has opened the box, and now the system reaches the only step that actually moves money or stock: refund, replace, or neither. This is where a lot of returns go wrong — refunded on the customer's say-so before anyone looked in the box, or refunded *and* replaced because two people each handled half the return. The handler avoids both by making the decision depend on a recorded inspection, proposing exactly one outcome with its reason, and handing that to a person to approve.

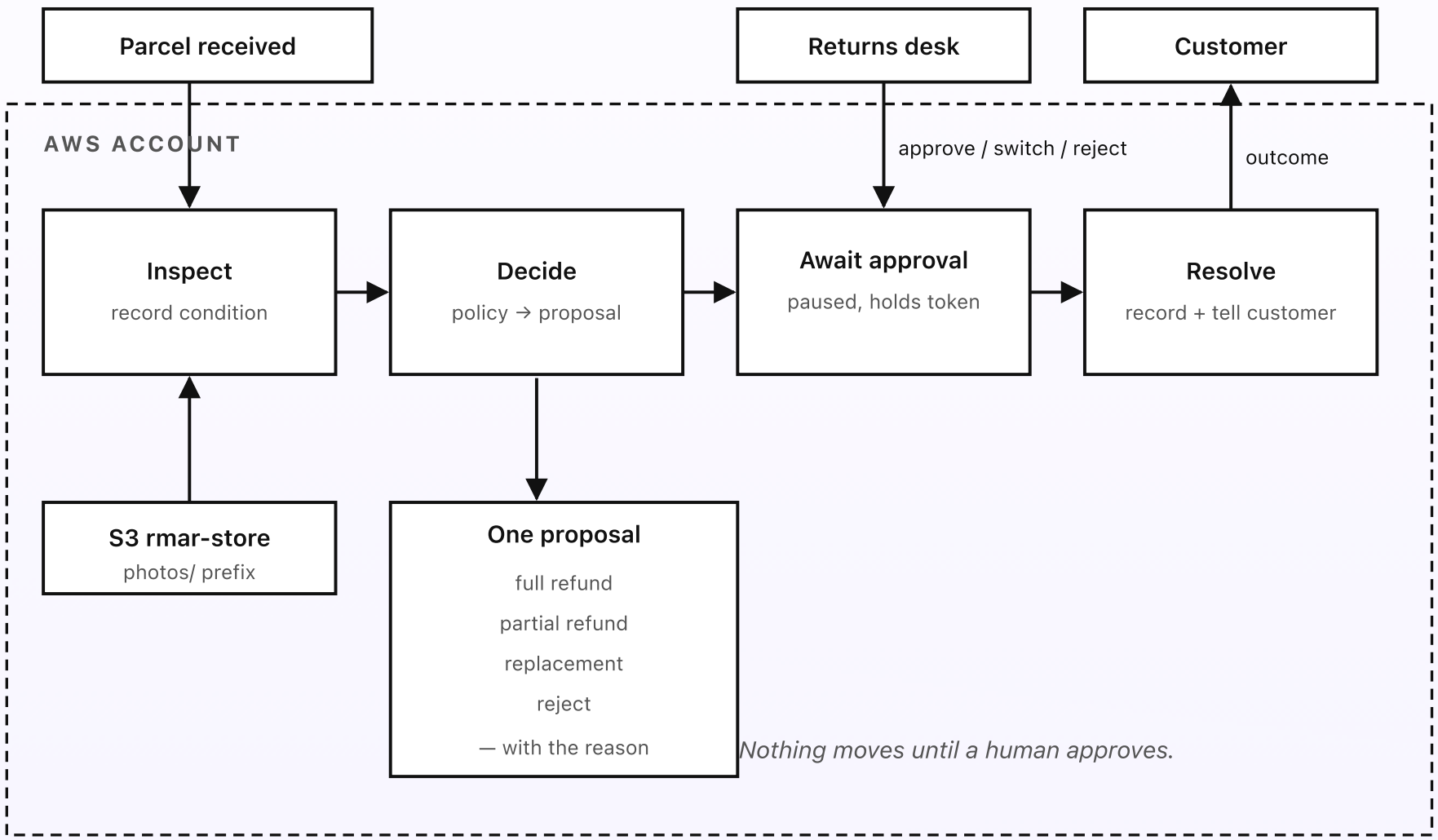

Parcel received
Returns desk
Customer
AWS ACCOUNT
approve / switch / reject
outcome
Inspect
record condition
Decide
policy → proposal
Await approval
paused, holds token
Resolve
record + tell customer
S3 rmar-store
photos/ prefix
One proposal
full refund
partial refund
replacement
reject
— with the reason
Nothing moves until a human approves.

Fig 5. From box to outcome. A staffer records the real condition; the decide step proposes one outcome with its reason; the returns desk approves, switches, or rejects; and only then is the result recorded and the customer told.

The inspection

When the carrier scan from Part 4 resumes the execution, the system doesn't decide anything yet — it emails the warehouse that a parcel for `RMA-4821-7` has arrived and needs checking. The inspect step is itself a short task-token wait: a staffer opens the box, then submits what they found through a simple form behind the `rmar-inspect` Function URL — which item, what condition (matches the order, sealed, used, damaged, wrong item), and one or two photos that land in the `photos/` prefix in S3. Submitting resumes the execution. The point is that the decision runs on what was actually in the box, recorded by a person, with photographic evidence kept — not on the customer's description from two weeks earlier.

The decision, in plain code

The `rmar-decide` step takes the inspection result and the original reason and applies your policy as code to propose exactly one outcome:

- **Full refund** — the right item came back in a resaleable condition within the window, and the reason is a straightforward change of mind or a fault you accept. The proposal carries the amount and the reason: "received sealed, matches order line, within window — propose full refund of £79."
- **Partial refund** — the item is back but not as it left: opened packaging, missing accessory, light wear. The policy sets the deduction, and the proposal shows

the maths: “returned used, −20% restocking per policy — propose refund of £63.20.”

- **Replacement** — the customer wanted a different size or a like-for-like swap and the item is resaleable and in stock. The proposal names the replacement: “faulty on arrival, same SKU in stock — propose replacement, no charge.”
- **Reject** — what came back doesn’t support a refund: the wrong item, damage beyond what the reason explains, or a box that’s effectively empty. The proposal says why and proposes returning it to the customer.

Like eligibility, this is deterministic. The policy already defines what a used return is worth and when a swap beats a refund; the step just applies it and shows its working. A model is never asked “should we refund this?” — only, at most, to phrase the customer’s outcome email kindly once a human has approved it.

The approval

The proposal doesn’t execute itself. The execution parks on an *await approval* step — the same task-token pattern as the inbound wait — and the returns desk gets one email: the RMA, the inspection photos, the proposed outcome, and the reason, with three buttons behind the `rmar-approve` Function URL. *Approve* takes the proposal as-is. *Switch* overrides it — refund instead of replace, or the reverse, with a note — for the judgement calls a policy can’t fully capture. *Reject* declines it. A tap resumes the execution into `rmar-resolve`, which records the final outcome and reason to the audit log and emails the customer what’s happening. The refund itself is queued into your normal payment run, and a replacement into your normal fulfilment — the handler proposes and records; it never reaches into the money or the stock on its own.

DESIGN RULES FOR THE DECISION

- Decide on the inspection, not the request. The recorded condition of what arrived is the only input that counts.
- One proposal, with its reason. Full refund, partial, replacement, or reject — never a menu, always a recommendation that shows its working.
- The policy does the maths. Restocking deductions and refund-versus-replace come from the written policy, applied as code.
- A human approves, switches, or rejects. The verdict is always a person's; the system only ever proposes.
- Record before you act. The outcome and reason are logged, then the refund or replacement joins your normal runs — not before.

PART 6 OF 7

JUNE 28, 2026 PART 6 OF 7 · RETURN AND RMA HANDLER SERIES ~6 MIN READ

What the return and RMA handler costs

A returns desk that costs more to run than the returns it processes is a bad trade. This post is the cost breakdown: every AWS service this system touches, what each adds up to at around 120 returns a month, why the total lands near $2.90, and which two lines — Step Functions transitions and the external label API — are the only ones that move when volume does.

KEY TAKEAWAYS

- About $2.90/month at roughly 120 returns, and the fixed cost is essentially zero — nothing runs between returns.
- Only two lines move with volume: Step Functions state transitions, and the carrier label API — and the label API isn't an AWS cost at all.
- The one real AWS fixed line is Secrets Manager, at $0.40 per secret per month for the label and webhook keys.
- Bedrock is optional. Turn the wording off and the bill drops by about $0.75 with no loss of function.
- At ten times the volume the bill lands near $9 — it scales gently because the fixed lines don't move.

Where the money goes

The handler is serverless end to end, and a return spends most of its life as a paused state machine that costs nothing while it waits. There's no instance ticking over, no queue being polled, no idle bill. You pay for a return only when one is actually moving through a stage. At a typical small-shop volume — call it 120 returns a month — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two secrets — carrier/label API key, webhook signing key ($0.40 each)	$0.80
Bedrock (Claude Haiku 4.5)	Optional wording for the RMA and outcome emails (~240 calls)	$0.75
S3	Label PDFs, inspection photos, and the Drive mirror	$0.35
DynamoDB (on-demand)	Returns table, orders mirror, audit — small reads and writes	$0.25
CloudWatch Logs	Function logs, 7-day retention	$0.20
EventBridge (bus + Scheduler)	Carrier tracking events, the daily sweep, the Drive sync	$0.17
SES (inbound + outbound)	Return requests in; RMA, approval, and outcome emails out	$0.15

AWS service	What it does here	Monthly
Lambda (Python 3.14, arm64)	Intake, eligibility, label, track, inspect, decide, approve, sweep, sync	$0.15
Step Functions (Standard)	~1,400 state transitions across ~120 executions	$0.05
SQS + DLQ	Buffering between the edges and the workers	$0.03
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~120 returns/month**	**$2.90**

The shape of that bill is the surprise. The two services people assume will dominate — Step Functions, which orchestrates the whole thing, and Lambda, which does the actual work — are among the cheapest lines on the list, because Step Functions Standard bills per state transition (a fraction of a cent each) and the Lambdas run for milliseconds. The biggest line is Secrets Manager, which charges a flat $0.40 per secret whether you process one return or a thousand. The second is Bedrock, and that's optional. Everything that does the genuine logistics — the orchestration, the matching, the decision — rounds to almost nothing.

Only Step Functions transitions and the external label API scale with volume; the one fixed cost is Secrets Manager.

Fig 6. The monthly bill at about 120 returns. Secrets Manager and an optional Bedrock are over half of it; the orchestration and the logistics round to cents.

The two things that scale

Only two costs in this system actually move when you do more returns. The first is Step Functions: every return is one execution of around a dozen state transitions, and Standard workflows bill $0.025 per 1,000 transitions, so 120 returns is a few cents and 1,200 is a few more. It scales perfectly linearly and stays tiny — the long

waits in the middle, which could last weeks, cost nothing, because a paused execution isn't transitioning. The second is the carrier's label API, and that one is genuinely per-return — but it's a postage cost on your courier contract, not an AWS line, so it doesn't appear on the table at all. Whatever a return label costs you today, the handler just asks for one per cleared return; it doesn't add a markup.

The fixed cost, and the optional one

Secrets Manager is the only thing here that bills while the system sleeps: two secrets at $0.40 each is $0.80 a month no matter what, more than a quarter of the total at this volume. Use a single combined key and you'd shave $0.40 off. Bedrock is the other notable line, and it's entirely optional: it only phrases the customer emails, so turning it off drops the bill by about $0.75 and replaces the wording with templates that read perfectly well. Everything else — S3, DynamoDB, logs, EventBridge, SES, Lambda, SQS — is usage-priced and rounds toward zero at this scale.

Push this to a busy shop — 1,200 returns a month, ten times the volume — and the bill lands near $9, not $29. It's strongly sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules stay at a cent, Budgets stays free. What grows is the genuinely usage-priced work — more Bedrock calls, more S3 for photos, a bit more DynamoDB and logs, and Step Functions creeping from a few cents to a few more. The honest read: the AWS bill is rounding error against the cost of running a returns desk by hand, where the real money is the refund issued in error and the parcel that quietly went missing — both of which this system is built to stop.

DESIGN RULES THAT SHAPED THE COST

- Pay per return, not per hour. No always-on compute, and a paused execution costs nothing while a parcel travels.
- Orchestration is cheap. Step Functions bills transitions, not waiting; the whole flow is a few cents a month.
- Know your one fixed cost. Secrets Manager is the only line that bills while nothing is happening.
- AI is optional. Bedrock only writes words; switch it off and the system still runs end to end.
- The label fee lives with the carrier. The biggest per-return cost is postage on your courier contract, not AWS.

PART 7 OF 7

JUNE 28, 2026 PART 7 OF 7 · RETURN AND RMA HANDLER SERIES ~8 MIN READ

Engineering reference: the return and RMA handler architecture

This is the return and RMA handler with the friendly labels removed: the state machine and every state in it, the real Lambda names, the returns table and its keys, the carrier-webhook and sweep wiring, the bucket layout, and the IAM scope. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- One Standard Step Functions state machine, `rmar-flow`, drives every return; ten small Lambdas back its tasks.
- Three of the states wait on a task token: AwaitInbound, Inspect, and AwaitApproval — each resumed by its own function.
- Three DynamoDB tables on-demand: `rmar-returns` (with tracking and order GSIs), `rmar-orders`, and an append-only `rmar-audit`.
- Inbound is SES plus Lambda Function URLs; carrier scans route through an EventBridge bus, and two Scheduler rules drive the sweep and the Drive sync.
- One region, `eu-west-2`, one account, no API Gateway, no NAT Gateway, nothing always-on.

The architecture, for engineers

This is the same system as Part 1 with the friendly labels removed and the real resources named. Everything runs in one region, `eu-west-2` (London), in one account. There is no API Gateway and no NAT Gateway; inbound HTTP arrives on Lambda Function URLs, email through SES, and the whole lifecycle of a return is one execution of a Standard Step Functions state machine that parks for free between stages.

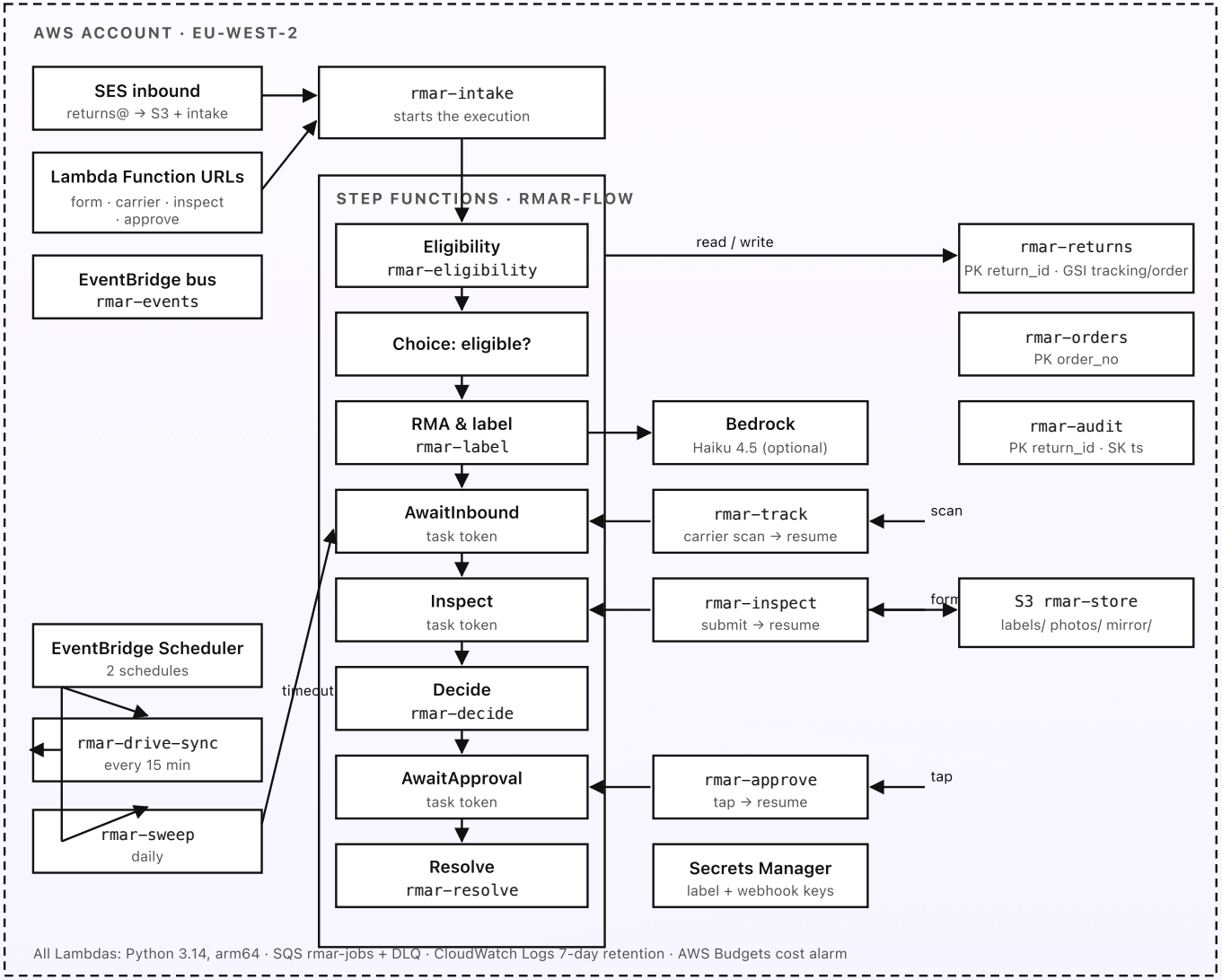

AWS ACCOUNT · EU-WEST-2
SES inbound
returns@ → S3 + intake
Lambda Function URLs
form · carrier · inspect · approve
EventBridge bus
rmar-events
rmar-intake
starts the execution
STEP FUNCTIONS · RMAR-FLOW
Eligibility
rmar-eligibility
Choice: eligible?
RMA & label
rmar-label
AwaitInbound
task token
Inspect
task token
Decide
rmar-decide
AwaitApproval
task token
Resolve
rmar-resolve
read / write
rmar-returns
PK return_id · GSI tracking/order
rmar-orders
PK order_no
Bedrock
Haiku 4.5 (optional)
rmar-audit
PK return_id · SK ts
rmar-track
carrier scan → resume
scan
rmar-inspect
submit → resume
form
S3 rmar-store
labels/ photos/ mirror/
rmar-approve
tap → resume
tap
Secrets Manager
label + webhook keys
EventBridge Scheduler
2 schedules
rmar-drive-sync
every 15 min
rmar-sweep
daily
timeout
All Lambdas: Python 3.14, arm64 · SQS rmar-jobs + DLQ · CloudWatch Logs 7-day retention · AWS Budgets cost alarm

Fig 7. The return and RMA handler drawn for engineers: SES and Function URL edges, the `rmar-flow` state machine and its eight states, three task-token resume functions, three DynamoDB tables, one S3 bucket, and two scheduled jobs. One region, one account, no API Gateway.

The state machine

One Standard state machine, `rmar-flow`, started once per return by `rmar-intake`. Standard (not Express) because executions live for days and the long waits are free. Its states, in order:

- **Eligibility** — task, `rmar-eligibility`. Matches the order and runs the policy checks.
- **EligibleChoice** — choice. On a fail, routes to **DeclineReturn**, a terminal task that emails the customer the reason; otherwise continues.
- **IssueRmaAndLabel** — task, `rmar-label`, with a retry policy; idempotent so a retry never buys a second label.
- **AwaitInbound** — task with `waitForTaskToken`. Parks until `rmar-track` resumes it on a delivered scan, or **TimeoutClose** fires from the sweep.
- **Inspect** — task with `waitForTaskToken`. Parks until a staffer submits the condition through `rmar-inspect`.
- **Decide** — task, `rmar-decide`. Produces one proposed outcome with its reason.
- **AwaitApproval** — task with `waitForTaskToken`. Parks until the returns desk taps a button via `rmar-approve`.

- **Resolve** — task, `rmar-resolve`. Records the outcome to the audit log and notifies the customer. Terminal.

Lambda functions

Ten functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. An SQS queue (`rmar-jobs`, with `rmar-jobs-dlq` after five attempts) buffers the edge functions from the slower work behind them.

- `rmar-intake` — backs the returns-form Function URL and the SES inbound rule; normalises the request and calls `StartExecution` on `rmar-flow`.
- `rmar-eligibility`, `rmar-label`, `rmar-decide`, `rmar-resolve` — the four synchronous task states described above.
- `rmar-track` — triggered by the EventBridge rule on delivered scans; finds the return by `tracking_no` and calls `SendTaskSuccess` on the AwaitInbound token.
- `rmar-inspect` — backs the inspect Function URL; stores photos to S3 and resumes the Inspect token.
- `rmar-approve` — backs the approve/switch/reject Function URL; resumes the AwaitApproval token with the human's decision.
- `rmar-sweep` — scheduled daily; nudges or times out returns parked past the deadline and flags genuinely lost parcels to a person.
- `rmar-drive-sync` — scheduled every 15 minutes; mirrors the orders sheet and policy doc into `rmar-orders` and the S3 `mirror/` prefix.

Data stores, edges, and schedules

- **DynamoDB (all on-demand).** `rmar-returns` — PK `return_id` (the RMA), GSI1 on `tracking_no` for the carrier-scan resume, GSI2 on `order_no`; holds the lifecycle status, the current task token, and the eligibility, inspection, and decision results. `rmar-orders` — PK `order_no`, the mirrored orders read by eligibility. `rmar-audit` — PK `return_id`, SK `ts`, append-only, one row per state change and decision.
- **S3.** One bucket, `rmar-store`, versioned: `labels/` for label PDFs, `photos/` for inspection photos, `mirror/` for the Drive snapshot, and a raw-mail prefix for SES inbound with a 30-day lifecycle expiry.
- **SES.** One inbound receipt rule on the returns address that writes to S3 and invokes `rmar-intake`; verified domain and DKIM for the RMA, approval, and outcome emails.
- **EventBridge.** A custom bus, `rmar-events`, with one rule matching delivered scans to `rmar-track`. EventBridge Scheduler runs two jobs — `rmar-drive-sync` at `rate(15 minutes)` and `rmar-sweep` at a daily `cron` before business hours.
- **Secrets Manager.** One secret for the carrier/label API key, one for the webhook signing key; fetched at call time, never in env vars or the policy doc.
- **Bedrock.** Model id `anthropic.claude-haiku-4-5` via the Global cross-Region inference profile, invoked only by `rmar-label` and `rmar-resolve`, and only for wording. Optional.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `rmar-intake` can read `rmar-orders` and call `StartExecution` on the one state machine; it cannot read the audit table or call Bedrock. The three resume functions hold only `states:SendTaskSuccess` and the narrow reads they need — `rmar-track` can query the `tracking_no` GSI and read the webhook secret, nothing more. `rmar-label` is the only role that can read the carrier secret and write to `labels/`; `rmar-resolve` is the only role that writes the final outcome. Only `rmar-label` and `rmar-resolve` carry `bedrock:InvokeModel`, scoped to the one Haiku profile. No role can delete from any table. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call and is not a data store. An AWS Budgets alarm watches the monthly spend and notifies if it drifts above a few dollars — the cheapest early signal that the label API or a loop is misbehaving.

DESIGN RULES THAT SHAPED THE BUILD

- One state machine, one return. The whole lifecycle is one Standard execution you can inspect at any point.
- Wait on tokens, not compute. The three pauses cost nothing; functions resume them only on a real event.
- One job per function. Ten small Lambdas, an SQS buffer, and no function that does everything.
- Least privilege, per role. Only the resume functions can SendTaskSuccess; only the label role reads the carrier secret.
- State in DynamoDB, blobs in S3. Returns, orders, and audit in tables; labels, photos, and the mirror in one bucket.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, optional and only for words.