7-PART SERIES · FREE COMPANION

Order status responder

Customers ask “where’s my order?” by email, SMS, and web chat, and the answer is always the same boring lookup — find their order, check the carrier, tell them when it’s coming. This is the design of a small serverless responder that matches each message to the right order, pulls the carrier’s live tracking, and writes one friendly, accurate reply with a real ETA. It never invents a tracking status, and anything it can’t match — or any order that’s delayed, lost, or stuck — is handed to a person with the full context. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Order status responder

PART 1 OF 7

JUNE 20, 2026 PART 1 OF 7 · ORDER STATUS RESPONDER SERIES ~10 MIN READ

An order status responder on AWS for a few dollars a month

Every shop that ships anything hears the same question all day: *where's my order?* It arrives by email, by SMS, and through the chat box on the website, and answering it is the same dull lookup every time — find the order, check the carrier, say when it's coming. This post walks through the design of a small responder that does that lookup the moment the question is asked, replies with a real ETA, and quietly hands the hard ones to a person.

KEY TAKEAWAYS

- Three inbound channels: email through SES, SMS through a provider webhook, and the web-chat box on your site.
- Every message is matched to one order — by the order number in it, or by the sender's email or phone.
- A matched order gets a live carrier lookup; one Bedrock call turns those real facts into one friendly reply with an ETA.
- It never invents a status. Unmatched messages and delayed, lost, or stuck orders go to a person with full context.
- Designed on AWS for about $2.80/month at roughly 500 enquiries a month. It only ever replies — a human handles the rest.

The whole system on one page

Before any code, here's the shape of what we're designing. Every shop that ships things fields the same question all day — *where's my order?* — and it arrives on whatever channel the customer happens to be on. The work is always identical: find their order, check the carrier, tell them when it's coming. The system below does exactly that, and nothing more.

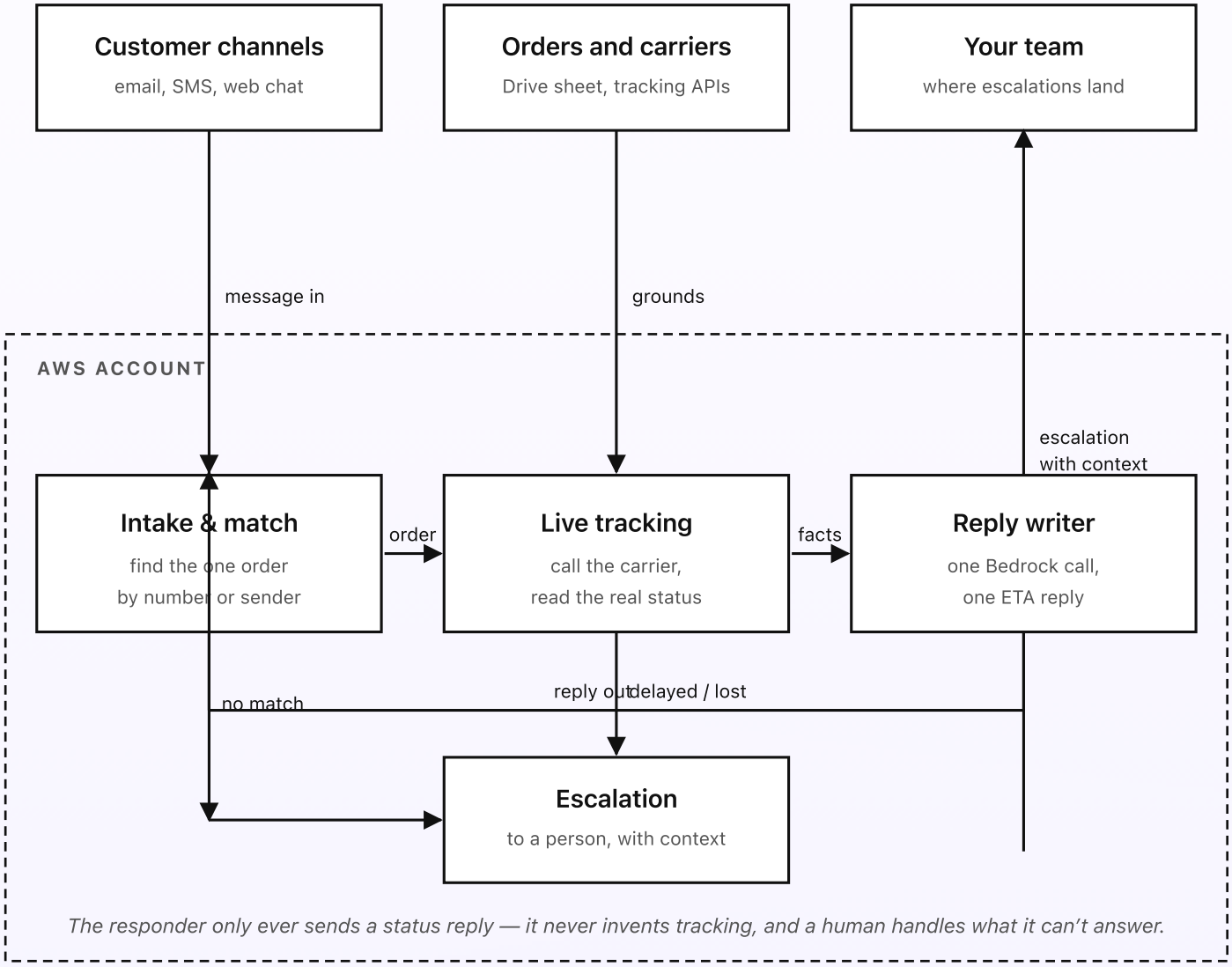

Customer channels
email, SMS, web chat
Orders and carriers
Drive sheet, tracking APIs
Your team
where escalations land
message in
grounds
AWS ACCOUNT
escalation
with context
Intake & match
find the one order
by number or sender
order
Live tracking
call the carrier,
read the real status
facts
Reply writer
one Bedrock call,
one ETA reply
no match
reply out
delayed / lost
Escalation
to a person, with context
The responder only ever sends a status reply — it never invents tracking, and a human handles what it can't answer.

Fig 1. Three channels outside, four pieces inside AWS. Messages flow in from email, SMS, and web chat. Intake matches each to one order, Live tracking calls the carrier, and the Reply writer sends one ETA reply. Anything unmatched or delayed branches to Escalation.

What you set up once (the outside)

- **Orders and carriers.** A Google Sheet in a Drive folder with one row per order: order number, customer name, customer email, customer phone, the items, the carrier, the tracking number, and a status. You already keep this, or something like it, in whatever you sell through; this just puts it where the responder can read it. New messages enter via three channels covered in Part 2 — an email lane (customers reply to your order emails or write to a support address), an SMS lane (your SMS provider posts inbound texts to a webhook), and the web-chat box on your site. The carriers' own tracking APIs are the second source: they, not the sheet, are the truth about where a parcel actually is.
- **A small settings doc.** One short Google Doc in the same folder. It holds the things you'll want to change without a deploy: your business name and the voice for replies, the carriers you use and which account each tracking number belongs to, the "quiet hours" for any outbound SMS, and the rules for when to escalate rather than answer — for example, always escalate anything that reads as a complaint, and always escalate an order with no carrier scan in five days.
- **Your team.** The person who picks up everything the responder deliberately won't touch — usually whoever does support. They get an email (or a row in a shared inbox) with the original message, the order the system matched it to, and the raw carrier response, so they can answer with everything in front of

them. The responder never refunds, cancels, or reships; those are always a human's call.

What runs on every message (the inside)

- **Intake and match.** Whatever channel a message lands on, it ends up as one normalised record: who sent it, on what channel, and the text. The matcher looks first for an order number in the text; failing that, it matches on the sender's email or phone against the orders mirrored from the sheet. A confident single hit moves straight on. No match, or several open orders for one customer, goes to escalation rather than a guess. This is Part 2.
- **Live tracking.** With an order in hand, a small Lambda calls the matching carrier's tracking API using a key from Secrets Manager, and turns the carrier's own status codes into one normalised picture: in transit, out for delivery, delivered, or a problem. The carrier's answer — not the sheet, not a guess — is the only source of a status. This is Part 3.
- **Reply writer.** One Bedrock Haiku 4.5 call takes the real tracking facts — carrier, last scan, location, ETA — and writes a single friendly reply in your voice, sent back out on the same channel the customer used. The model is handed the facts and told to use only those; if there are no facts, there is nothing to write and the order is escalated instead. This is Part 4.
- **Escalation.** The lane for everything that shouldn't get an automatic reply: an unmatched message, an upset customer, or an order the carrier shows as delayed, lost, or stuck. It also runs as a daily sweep that re-checks in-flight orders and flags the ones going quiet before the customer has to chase. This is Part 5.

In plain words

A text comes in to your shop's number at 9:12 on a Saturday: "Hi, any update on order 10482?" The responder finds order 10482 in the mirrored sheet — one item, shipped two days ago with Royal Mail, tracking `AB123456789GB`. It calls Royal Mail's tracking API: the parcel was scanned at the local delivery office at 7:40 that morning, out for delivery today. One Bedrock call turns that into: "Hi Sam — your order 10482 is out for delivery today with Royal Mail and should arrive by this evening. You'll get it at the address ending Flat 2B. Anything else I can help with?" The customer has an honest answer within seconds of asking, on the channel they used, and nobody on your team touched it.

The same afternoon a different text arrives: "order 10090 still hasn't turned up and it's been two weeks, this is ridiculous." The responder matches the order, sees the last carrier scan was eleven days ago, and reads the tone. It sends nothing automatic. Instead it lands in your support inbox with the message, the order, and the tracking history attached, flagged *delayed — customer upset*, so a person picks it up with the full picture and decides whether that's a reship, a refund, or a call to the carrier.

DESIGN RULES THAT SHAPED EVERY DECISION

- One question, one job. The responder answers “where’s my order?” and does nothing else — no refunds, no cancellations, no reships.
- The carrier is the only source of a status. The responder reports what the tracking API says, and never makes a status up.
- A confident match or nothing. Ambiguous and unmatched messages go to a person, not to a guess.
- The model only writes words. Matching and tracking are deterministic; Bedrock is handed real facts and told to use only those.
- Settings live in a doc. Voice, carriers, quiet hours, and escalation rules change without a deploy.
- It never goes quiet on a problem. Anything delayed, lost, stuck, or upset is escalated to a human with the full context.

Why this shape

Most small teams answer “where’s my order?” one of three ways: someone stops what they’re doing to look it up, the message sits unanswered until someone has a quiet moment, or there’s a canned “please allow 3–5 working days” auto-reply that tells the customer nothing. Looking it up by hand is fine until twenty come in at once. Leaving it is how a one-line question becomes a one-star review. And the canned reply is worse than silence, because it pretends to answer while skipping the only part the customer cares about — the actual parcel.

The shape above keeps the sheet you already maintain as the list of orders, leans on the carrier's own tracking as the source of truth, and adds a small system that does the lookup the instant the question is asked. The 80% of enquiries that are simply "is it coming?" get an honest, specific answer in seconds. The few that are genuinely going wrong are pulled out and handed to a person early, with everything they need to act — while there's still time to fix it.

The next four posts walk through each piece in turn: how a message gets matched to an order, how live tracking gets pulled, how a status reply gets written, and how a stuck order gets escalated. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 20, 2026 PART 2 OF 7 · ORDER STATUS RESPONDER SERIES ~6 MIN READ

How a “where’s my order?” message gets matched

Before anyone can be told where their order is, the system has to know *which* order they mean. This post is about that step alone: how a message that arrived by email, SMS, or chat is turned into one confident order — and what happens, deliberately, when it can’t be.

KEY TAKEAWAYS

- Three channels arrive in different shapes; the first job is to flatten them into one normalised message record.
- Matching tries the order number in the text first — the cleanest signal — then falls back to the sender’s email or phone.
- The orders are read from a DynamoDB mirror of your Drive sheet, refreshed every 15 minutes, so matching never waits on Drive.
- A single confident hit moves on; several open orders for one sender, or none, is escalated rather than guessed.
- Matching is plain Python. No model decides which order a customer means.

Three shapes, one record

A message can arrive three ways, and they look nothing alike on the wire. An email comes through SES as a raw MIME blob dropped in S3 — headers, subject, a body that might be HTML, maybe a signature and three quoted replies underneath. An SMS arrives as a small JSON payload posted by your SMS provider to a Lambda Function URL — a from-number, a to-number, and a short body. A web-chat message is another small JSON post from the widget on your site, carrying whatever identifier the visitor gave (often an email, sometimes nothing).

The intake Lambda's first job is to make these the same. Each becomes one normalised record: the channel it came in on, a sender identity (email address or phone number), the plain-text body with quoted history and signatures stripped, and a timestamp. From here on, nothing downstream cares whether the question came by text or email — it's just a message with a sender and some words.

Number first, then sender

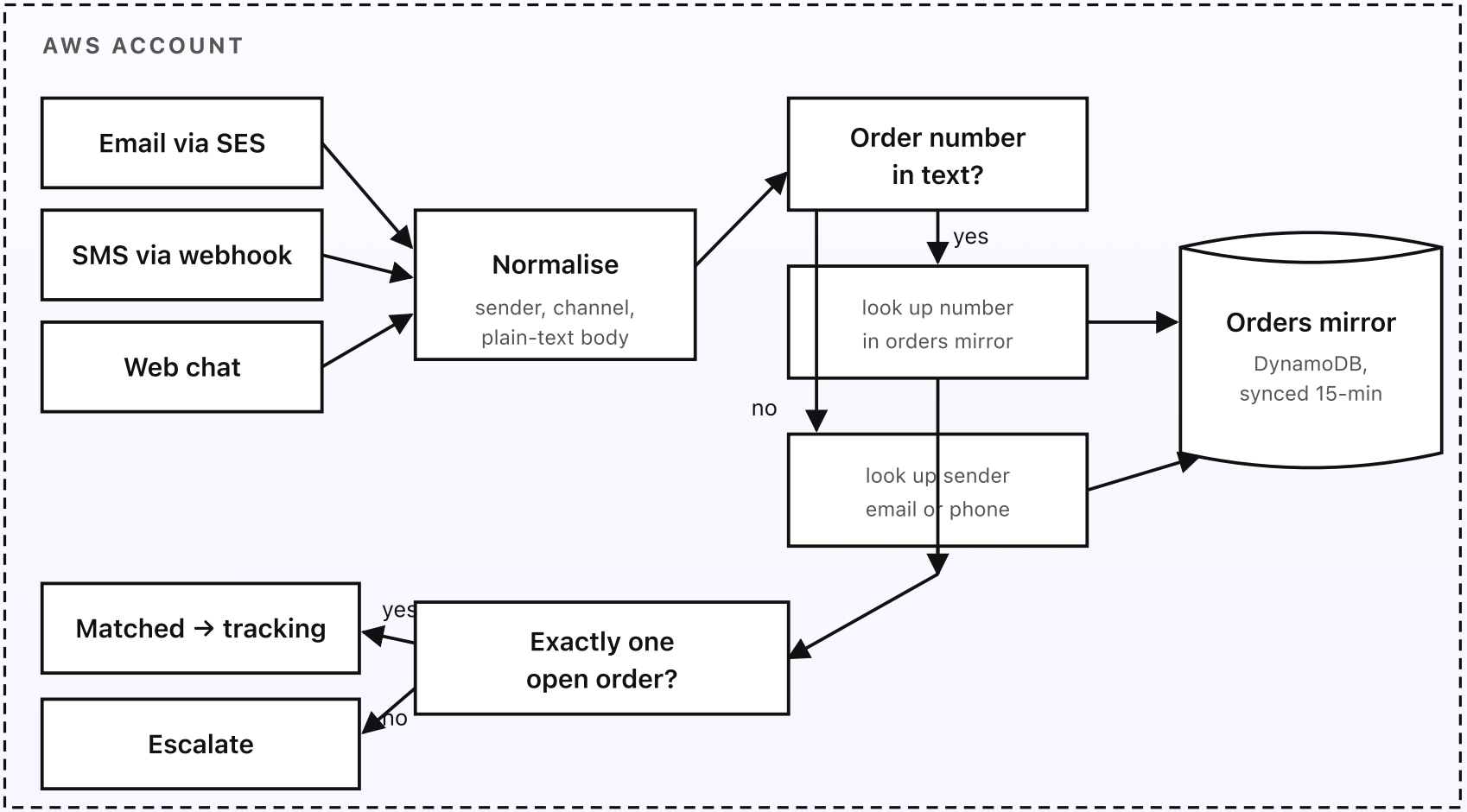
With a clean record, matching runs in a fixed order, cheapest and most certain signal first.

- **Order number in the text.** Most people quote it — "any update on 10482?", "order #10482", "ref 10482". A small set of patterns pulls candidate numbers out of the body and subject, and each is looked up directly in the orders mirror. A hit on a live order is the strongest match there is, because the customer told you exactly what they meant.
- **Sender email or phone.** When there's no number — "hi, where's my stuff?" — the matcher looks up the sender instead: the `from` email for email and chat,

the phone number for SMS, against the customer columns in the mirror. If that sender has exactly one open order, that's the match.

- **Neither, or too many.** If nothing matches, or the sender has several open orders and the text doesn't say which, the matcher does not pick one. It escalates with everything it tried, so a person resolves it in one reply rather than the system guessing wrong.

The phone and email are normalised before they're compared — numbers to E.164, emails lower-cased — so "+44 7700 900123" and "07700 900123" land on the same customer. The match is plain Python comparing strings; there is no model anywhere on this path, because "which order does this person mean" is a lookup, not a judgement call.



Matching is plain Python — a confident single hit moves on; anything ambiguous goes to a person.

Fig 2. Three channels normalise into one record, then a two-step match — order number first, sender second — against a DynamoDB mirror of the Drive sheet. One open order moves on; none or several escalates.

Why a mirror, not the sheet

The orders live in your Drive sheet, but the matcher never reads Drive directly. A small `osr-drive-sync` Lambda, on an EventBridge schedule every 15 minutes, pulls the sheet and writes each order as a row in a DynamoDB table keyed by order number, with secondary lookups by email and phone. Matching then reads from DynamoDB, which is fast, predictable, and doesn't burn through Drive API quota every time a customer texts. It also means a burst of fifty enquiries at once doesn't hammer Google — they all read the same warm mirror.

Fifteen minutes of staleness is fine here, because the thing that actually moves — the parcel — isn't in the sheet at all. The sheet says which carrier and tracking number an order has; whether it's moved is the carrier's job to answer, live, which is Part 3. The mirror only needs to be fresh enough to know an order *exists* and who it belongs to.

DESIGN RULES THAT SHAPED THE MATCHER

- Normalise first. Channel differences die at the door; everything downstream sees one message shape.
- Strongest signal first. An order number the customer typed beats an inferred sender match every time.
- Read a mirror, not the source. DynamoDB keeps matching fast and keeps Drive quota out of the hot path.
- One hit or escalate. The matcher never breaks a tie between two orders; a person does.
- No model on the match. Which order someone means is a lookup, and lookups should be deterministic.

PART 3 OF 7

JUNE 20, 2026 PART 3 OF 7 · ORDER STATUS RESPONDER SERIES ~7 MIN READ

How live tracking gets pulled

A matched order has a carrier and a tracking number, but those are just a promise until something checks them. This post is about the lookup: how the responder calls the carrier's live tracking, turns a dozen carrier-specific status codes into one honest picture, and refuses to make anything up when the carrier doesn't answer.

KEY TAKEAWAYS

- A matched order carries a carrier and a tracking number; the tracking Lambda turns those into a live status.
- Each carrier's API speaks its own dialect; the Lambda normalises every response into one small set of statuses.
- Carrier keys live in Secrets Manager, one per carrier, never in code or the sheet.
- Results are cached briefly in DynamoDB so ten "any update?" texts about one parcel make one carrier call, not ten.
- If the carrier is slow, errors, or has no scan, the responder reports the truth or escalates — it never fills the gap with a guess.

From an order to a status

By the time this step runs, the matcher has handed over one order, and that order has two fields that matter here: a carrier (“Royal Mail”, “DPD”, “Evri”) and a tracking number. Everything else — the items, the customer, the order date — is context for the reply, not for the lookup. The tracking Lambda’s whole job is to take that carrier-and-number pair and come back with one honest answer: where is this parcel, and when will it arrive?

It does that by calling the carrier’s own tracking API. That call is the only thing in the entire system that knows the real state of a parcel. The sheet might say “shipped”; that just means a label was printed. The carrier’s API knows whether it was actually scanned, whether it’s sitting in a depot, whether it went out for delivery this morning, or whether it was handed over three days ago and signed for. The responder treats that API response as the single source of truth and reports nothing the API didn’t say.

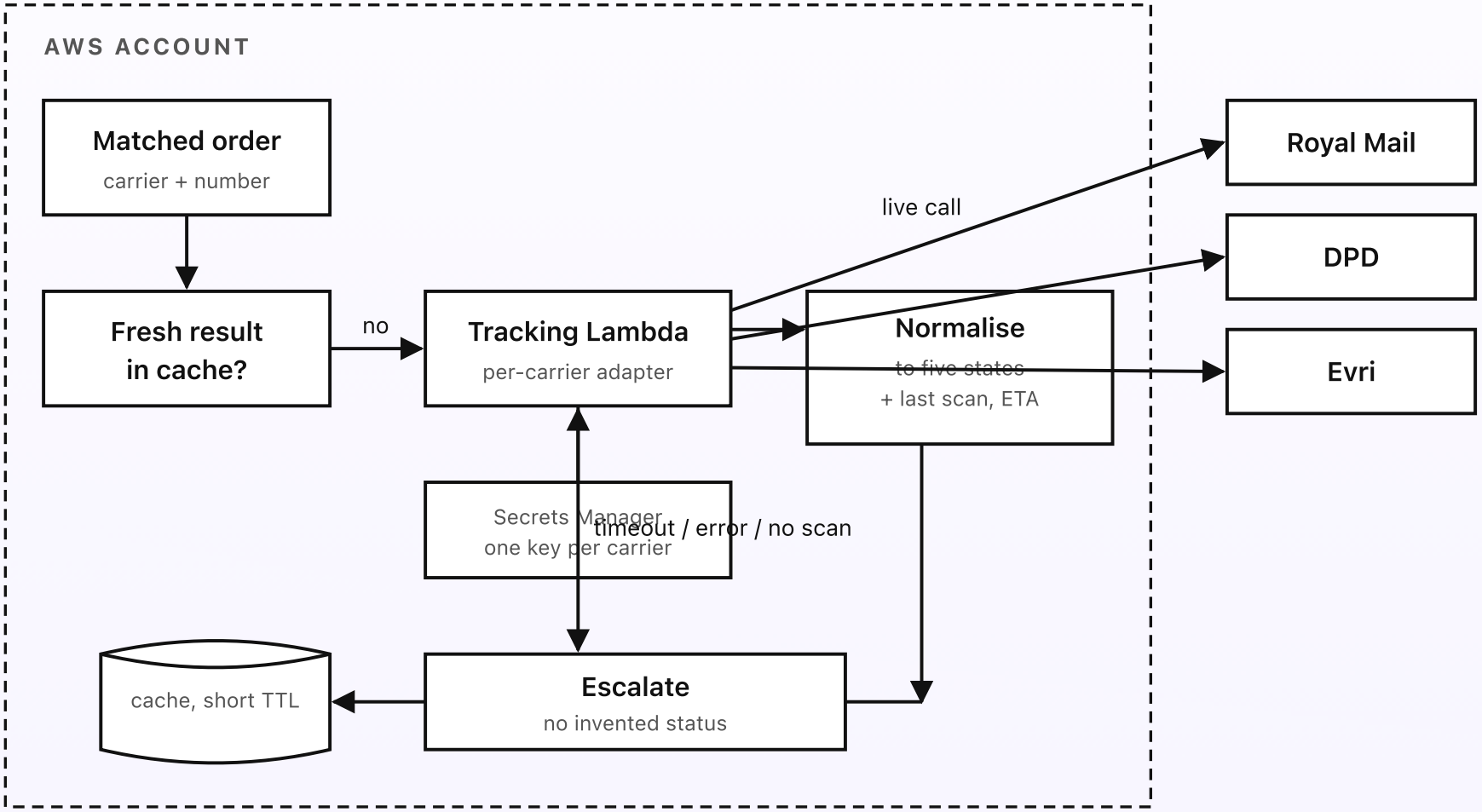
Many dialects, one vocabulary

Every carrier’s API is different. They authenticate differently, they shape their JSON differently, and — most awkwardly — they all have their own list of status codes. One carrier’s “OFD” is another’s “Out for delivery” is another’s numeric `17`. If that mess leaked into the reply, customers would get a different-sounding answer depending on which courier you happened to use.

So the Lambda has a thin adapter per carrier that does two things: makes the authenticated call, and maps the carrier’s status onto one small shared vocabulary. The reply writer in Part 4 only ever sees that shared vocabulary:

- **In transit** — accepted and moving, with the last scan location and time, and the carrier's estimated delivery date if it gives one.
- **Out for delivery** — on a vehicle for delivery today.
- **Delivered** — with the delivery time and, where the carrier provides it, who signed or where it was left.
- **Exception** — the carrier is flagging a problem: a failed delivery, a held parcel, an address issue, a return to sender.
- **No movement** — the label exists but the carrier has no scan, or hasn't scanned it in days.

Adding a carrier means writing one more adapter — one function that knows that carrier's auth and its status table — and nothing downstream changes. The reply writer and the escalation logic only ever reason about the five normalised states, never about a particular courier's codes.



The carrier is the only source of a status. If it doesn't answer, the responder reports that or escalates — never a guess.

Fig 3. The tracking lookup: a short-lived cache absorbs repeat asks, a per-carrier adapter makes the authenticated call with a key from Secrets Manager, and the response is normalised to five states. Silence or errors escalate rather than invent a status.

Caching, keys, and silence

Two practical things make this lookup cheap and safe. First, a short cache. The same parcel often gets asked about more than once — the customer texts, then emails an hour later, then a partner asks too. Each carrier-and-number result is written to a small DynamoDB table with a short time-to-live (a handful of minutes). A second enquiry about the same parcel inside that window reads the cache instead of calling the carrier again. Carriers update their tracking every few hours, not every few seconds, so a few minutes of cache costs the customer nothing in accuracy and saves a pile of API calls on a busy day.

Second, every carrier credential lives in Secrets Manager, one secret per carrier, fetched at call time and never written into code, environment variables, or the sheet. Rotating a key is a Secrets Manager update, not a deploy.

And then the part that matters most — what happens when the carrier *doesn't* answer. APIs time out, return errors, or come back with a tracking number they've simply never scanned. In every one of those cases the responder does the same thing: it does not invent a status. A timeout or error escalates with a note that the carrier couldn't be reached. A "no movement" result — a label with no scan, or no scan in days — is exactly the kind of stuck order that should reach a person, so it escalates too (Part 5). The one thing the responder will never do is tell a customer "it's on its way" when the carrier never said so.

DESIGN RULES THAT SHAPED THE TRACKING LOOKUP

- The carrier API is the only source of a status. Nothing else — not the sheet, not a model — gets to say where a parcel is.
- Normalise to a small shared vocabulary. Five states, so the reply reads the same whichever courier carried it.
- One adapter per carrier. Adding a courier is one function; nothing downstream changes.
- Cache briefly. Repeat asks about one parcel make one carrier call, not ten.
- Keys in Secrets Manager. One secret per carrier, rotated without a deploy.
- Silence is not a status. Timeouts, errors, and no-scan parcels escalate — the responder never fills the gap.

PART 4 OF 7

JUNE 20, 2026 PART 4 OF 7 · ORDER STATUS RESPONDER SERIES ~7 MIN READ

How a status reply gets written

By this point the responder knows the order and has the live tracking in hand. All that's left is to say it like a person would. This post is about the only place a model is allowed near the system: the single Bedrock call that turns real tracking facts into one friendly reply with an ETA — and the fences that keep it honest.

KEY TAKEAWAYS

- One Bedrock Haiku 4.5 call per enquiry writes one reply — no chains, no follow-ups, no back-and-forth.
- The model is handed the already-fetched tracking facts and told to use only those; it never queries anything itself.
- If there are no facts to report, there is nothing to write — the order is escalated instead of dressed up.
- The reply goes back out on the same channel the customer used: SES for email, the provider for SMS, the widget for chat.
- Every reply is logged with the exact facts it was built from, so any message can be traced back to a real carrier scan.

The only place a model runs

This is the single spot in the whole system where a language model does anything. The matching was plain Python; the tracking lookup was a real API call. By the time we get here, the hard facts already exist: an order, a customer name, a carrier, a normalised status, the last scan and its location, and an ETA where the carrier gave one. None of that is in doubt. What's left is purely a writing job — turn those facts into one message that sounds like a person from your shop, not a status code.

So the responder makes exactly one Bedrock Haiku 4.5 call. It hands the model the facts and the voice from your settings doc, and asks for one short reply. That's the entire interaction — no multi-step reasoning, no tool use, no second call to "check" anything. The model is a writer, not a researcher. It cannot reach the carrier, it cannot read the orders, and it cannot decide a status. It can only phrase the facts it was given.

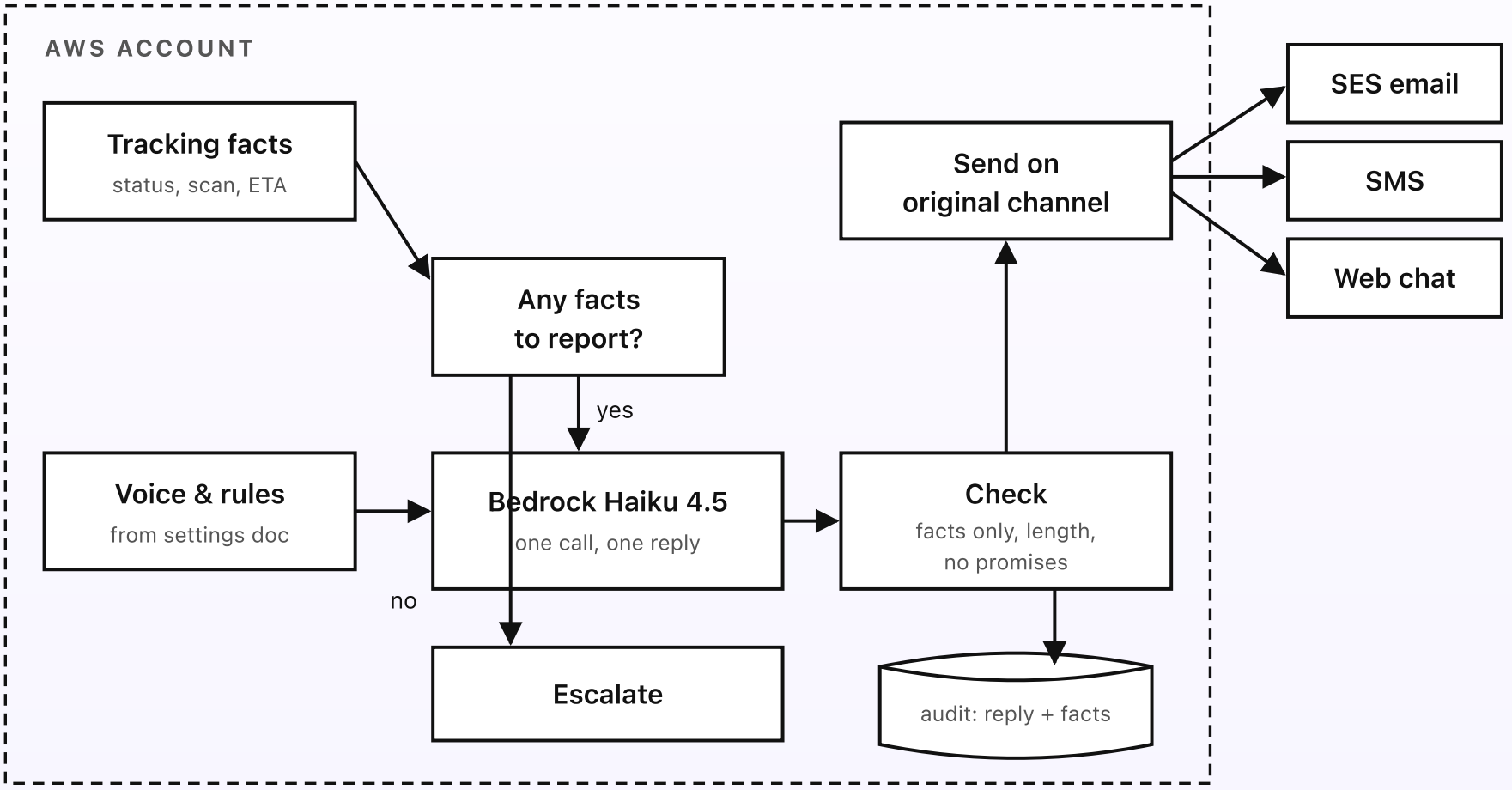
Facts in, words out

The prompt is built in code and is almost entirely facts. It carries the customer's first name, the order number, the items in plain language, the normalised status, the last scan time and place, and the ETA. The instructions around those facts are short and strict: use only what you're given, never state a delivery date the carrier didn't provide, never apologise for a delay that isn't in the facts, keep it to a few sentences, and match the house voice.

The result is a reply that reads naturally but can't drift from the truth, because there's nothing in the prompt to drift toward. A few worked examples:

- **Out for delivery.** “Hi Priya — good news, your order 10482 is out for delivery today with DPD and should arrive by this evening. You’ll get a one-hour window from them by text. Anything else I can help with?”
- **In transit with an ETA.** “Hi Tom — your order 10590 is on its way with Royal Mail. It was scanned at the Bristol hub last night and is due to arrive Thursday. I’ll keep an eye on it — shout if it’s not with you by then.”
- **Delivered.** “Hi Jo — order 10333 shows as delivered yesterday at 2:14pm, left in the porch as you’d asked. If it’s not where you expected, let me know and I’ll dig in.”

Notice what isn’t there. None of them invents a reason for a delay, promises a date the carrier didn’t set, or offers a refund or a reship — the responder has no power to do any of that, so it never says it. When the facts are thin (in transit, no ETA from the carrier), the reply is honest about that too: “it’s moving but the carrier hasn’t given a delivery date yet” rather than a made-up “3–5 days”.



The model only writes words from facts it was handed — if there are no facts, the order is escalated, not dressed up.

Fig 4. One Bedrock call turns the real tracking facts and your voice into one reply. A check confirms it stuck to the facts, the reply goes out on the channel the customer used, and the reply plus its facts are logged.

One reply, and a paper trail

Two rules keep the writer honest in practice. The first is the one-reply rule: the responder answers once and stops. It is not a chatbot holding a conversation — it gives the status and an offer to help further, and if the customer writes back with anything beyond “thanks”, that next message is treated as a fresh enquiry and, more often than not, escalated, because a second question usually means the simple lookup didn’t satisfy them. One good answer beats a model improvising its way through a back-and-forth.

The second is the paper trail. Before the reply is sent, a quick check confirms it’s within length, contains no date or promise that wasn’t in the facts, and reads as a status reply rather than a commitment. Then the reply *and* the exact facts it was built from are written to the audit table. If a customer ever says “your system told me Tuesday”, you can pull the record and see precisely which carrier scan and ETA produced that line — or confirm it never said any such thing. The reply is sent on the same channel it arrived on: a reply via SES for email, back through the SMS provider for a text, and into the open widget session for chat.

DESIGN RULES THAT SHAPED THE REPLY WRITER

- One call, one reply. The model writes a single message and stops; it is not a conversation engine.
- Facts in, words out. The prompt is the real tracking facts; the model phrases them and nothing else.
- No facts, no reply. An empty hand means escalate, never improvise.
- No promises it can’t keep. The responder reports status; it never offers refunds, reships, or dates the carrier didn’t set.
- Same channel back. Email gets an email, a text gets a text, chat gets chat.
- Log the facts, not just the reply. Every message can be traced to the carrier scan that produced it.

PART 5 OF 7

JUNE 20, 2026 PART 5 OF 7 · ORDER STATUS RESPONDER SERIES ~7 MIN READ

How a stuck order gets escalated

The responder is judged less by the easy replies it sends than by the hard ones it knows to hand over. This post is about the escalation lane: which enquiries never get an automatic answer, how a person receives them with the message, the matched order, and the raw tracking attached, and the daily sweep that catches orders going quiet before the customer has to chase.

KEY TAKEAWAYS

- Five things never get an automatic reply: no match, an ambiguous match, an upset customer, a carrier exception, and a no-movement parcel.
- An escalation is a single email to your support address with the message, the matched order, and the raw tracking attached.
- A daily EventBridge sweep re-checks in-flight orders and escalates the ones going quiet before the customer has to chase.
- The responder can flag, attach, and explain — it never refunds, reships, or cancels. A person decides every one of those.
- Escalations are de-duplicated, so one stuck order raises one ticket, not a new one every time it's asked about.

What never gets an automatic reply

A responder is only as trustworthy as the things it refuses to answer. The whole point of the automatic reply is the easy 80% — "is it coming?", "yes, Thursday" — and the whole point of escalation is to pull the other 20% out cleanly, before a thin or wrong reply makes a small problem worse. Five cases route straight to a person and never get an automatic answer:

- **No match.** The matcher couldn't tie the message to any order — no number in the text and no order under that email or phone. A person can spot in seconds that it's a new customer, a typo, or an order placed under a different address.

- **Ambiguous match.** The sender has several open orders and the message doesn't say which. Rather than answer about the wrong parcel, the responder hands over all the candidates and lets a person ask the obvious clarifying question.
- **Upset customer.** The message reads as a complaint — frustration, "ridiculous", "cancel", "refund", a third chase. Even with a clean status to report, a curt tracking line is the wrong response; a human picks these up.
- **Carrier exception.** The tracking came back as an *exception* — failed delivery, held parcel, address problem, return to sender. There's a real status, but it's one that needs a decision, not a quote.
- **No movement.** The carrier has no scan, or no scan in days. The parcel may be lost. Telling the customer "it's on its way" would be a lie, so it goes to a person to chase the carrier.

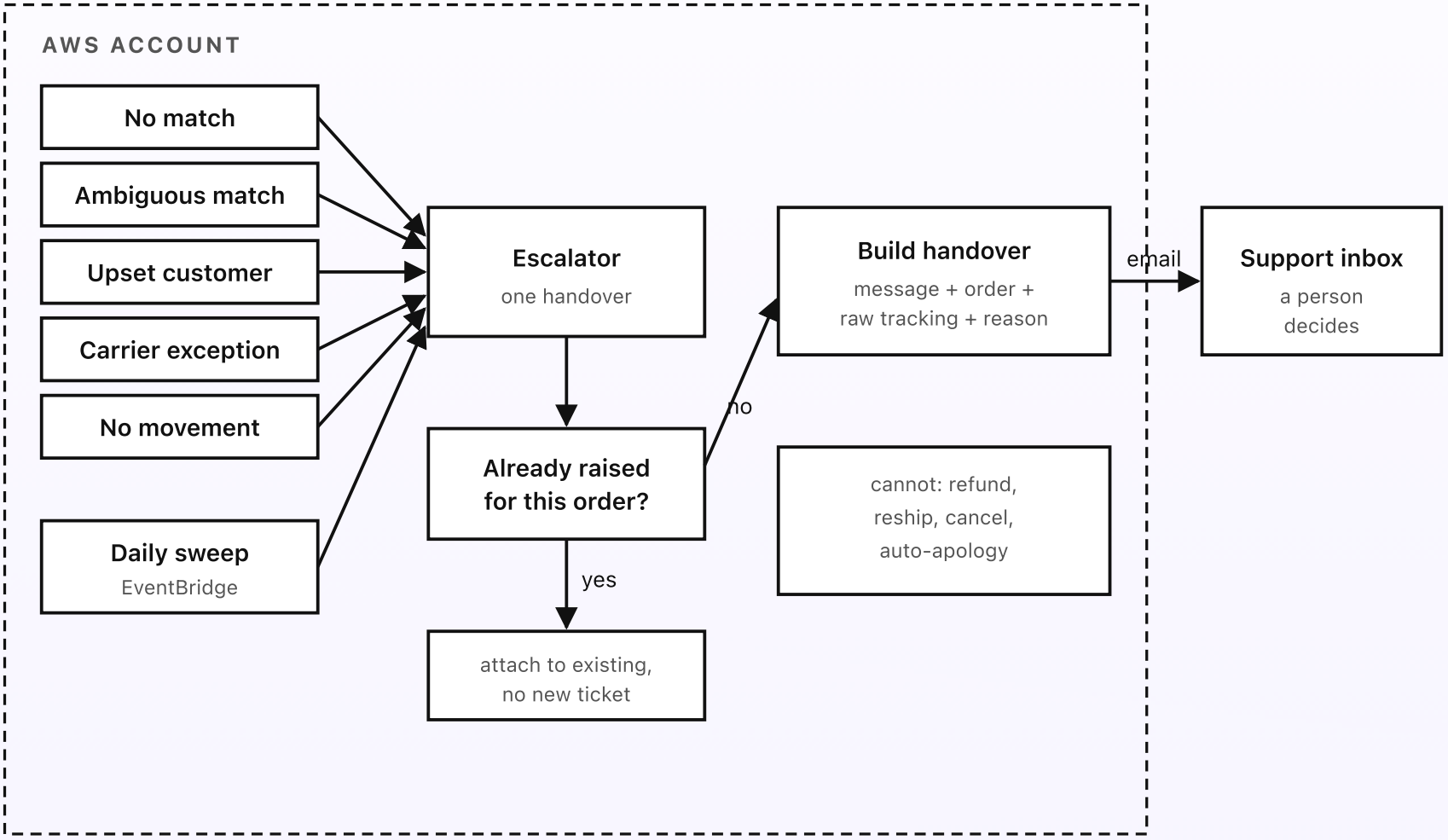
The first two come from the matcher (Part 2); the last two come from the tracking lookup (Part 3); the upset-customer case is a light tone check on the message text, deliberately tuned to over-escalate rather than risk answering a complaint with a status code.

What an escalation looks like

An escalation isn't a vague alert — it's everything a person needs to act, in one place. The escalator Lambda sends a single email to your support address (or drops a row into a shared inbox) carrying: the original message and the channel it came on, the order the system matched it to (if any), the raw and normalised carrier tracking, and a one-line reason it was escalated. The support person opens

it and has the full picture without looking anything up — they can reply to the customer, call the carrier, or start a refund, all from the same screen.

What the escalator deliberately can't do is act. It writes no refund, books no reship, cancels no order, sends no apology on its own. Those are decisions with money and goodwill attached, and the system's job is to put them in front of a person with the context to make them — not to make them itself. This is the same guardrail as the rest of the system, drawn at the riskiest edge: the responder proposes and informs; a human decides.



The responder flags, attaches, and explains — it never refunds, reships, or cancels. A person decides what happens next.

Fig 5. Five triggers and a daily sweep funnel into the escalator. It de-duplicates by order, builds one context-rich handover, and emails it to support. It cannot refund, reship, or cancel — a person decides.

The sweep, and not crying wolf

Two details stop the escalation lane from being either too quiet or too noisy. The first is a daily sweep. Customers shouldn't have to be the ones who notice a parcel went quiet. An EventBridge Scheduler rule fires the `osr-sweep` Lambda once a day; it walks the in-flight orders, re-checks the carrier on any that haven't moved, and escalates the ones stuck past a threshold from your settings doc — say, no scan in five days. That turns a future angry "where is it?!" into a quiet "we noticed this one's stuck" that your team can get ahead of.

The second is de-duplication. A genuinely stuck order tends to get asked about repeatedly, and the sweep will keep finding it. The escalator keeps an open-escalation record per order, so the first trigger raises one ticket and every later signal about the same order attaches to it rather than spawning a new one. Your support person sees one thread for one problem — the customer's three chases and the sweep's daily nudge all gathered together — not an inbox full of duplicates that trains them to ignore the lot.

DESIGN RULES THAT SHAPED ESCALATION

- Five clear triggers. No match, ambiguous, upset, exception, no movement — each never gets an automatic reply.
- Hand over the whole picture. Message, matched order, and raw tracking in one place, so a person acts without looking anything up.
- Over-escalate tone. A possible complaint goes to a human even when there's a clean status to report.
- Notice it first. A daily sweep catches stuck orders before the customer has to chase.
- One problem, one ticket. De-duplicate by order so the team isn't buried in repeats.
- Flag, never act. No refunds, reships, or cancellations — those are always a person's call.

PART 6 OF 7

JUNE 20, 2026 PART 6 OF 7 · ORDER STATUS RESPONDER SERIES ~6 MIN READ

What the order status responder costs

A responder that costs more than the time it saves is a toy. This post is the cost breakdown: every AWS service this system touches, what each one adds up to at around 500 enquiries a month, and why the total lands near $2.80 — plus what happens to the bill when the volume goes up tenfold.

KEY TAKEAWAYS

- About $2.80/month at roughly 500 enquiries, and the fixed cost is essentially zero — nothing runs when no one is asking.
- The single biggest line is Bedrock: one small Haiku 4.5 call per reply. Everything else is cents.
- The one fixed cost worth naming is Secrets Manager, at $0.40 per secret per month for the carrier and SMS keys.
- At ten times the volume (around 5,000 enquiries) the bill lands near $20 — it scales with use, not with idle time.
- Carrier tracking API fees are separate and depend on your courier contracts; they aren't an AWS cost.

Where the money goes

The responder is serverless end to end, so there's no instance ticking over at 3am and no idle bill. You pay for a message only when a message arrives. At a typical small-shop volume — call it 500 enquiries a month, of which perhaps 450 get an automatic reply and the rest escalate — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Bedrock (Claude Haiku 4.5)	One reply-writing call per answered enquiry (~450)	$1.35
Secrets Manager	Two secrets — carrier API key, SMS-provider key ($0.40 each)	$0.80
DynamoDB (on-demand)	Orders mirror, tracking cache, threads, audit — small reads and writes	$0.20
CloudWatch Logs	Function logs, 7-day retention	$0.15
SES (inbound + outbound)	Receiving email enquiries and sending email replies	$0.12
Lambda (Python 3.14, arm64)	Intake, match, tracking, replier, escalator, sweep, drive-sync	$0.10
S3	Raw inbound email and the Drive-sheet mirror file	$0.05

AWS service	What it does here	Monthly
SQS + DLQ	Buffering between intake and processing	$0.02
EventBridge Scheduler	15-minute Drive sync, daily stuck-order sweep	$0.01
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~500 enquiries/month**	**$2.80**

The shape of that bill is the point. The reply writer — the one place a model runs — is the largest single cost, and it's still well under half the total. The matching and the tracking lookup, which do the actual work of answering the question, cost almost nothing, because they're plain Python and a single API call. The only fixed line worth naming is Secrets Manager: $0.40 per secret per month, whether you handle one enquiry or ten thousand, which is why two carrier and provider keys quietly make up the bulk of what you pay before any message arrives.

The one real fixed cost

It's worth dwelling on Secrets Manager, because it's the only thing here that costs money while the system sleeps. Two secrets at $0.40 each is $0.80 a month no matter what — more than a quarter of the bill at this volume. If you used more carriers you'd add more secrets; if you used one, you'd shave $0.40 off. Everything else on the list is genuinely usage-priced and rounds to zero at idle,

which is exactly what you want from a system that only does work when a customer asks a question.

One cost that is *not* on this table: the carriers' own tracking APIs. Some couriers include tracking calls in your shipping contract; others meter them. Those fees sit with your carrier accounts, not with AWS, and they vary too much to put a single number on — but the brief cache from Part 3 is there precisely so a busy day of repeat "any update?" messages doesn't turn into a pile of billable carrier calls.

Fig 6. The monthly bill at about 500 enquiries. Bedrock and Secrets Manager are two-thirds of it; everything that does the actual work rounds to cents.

What ten times the volume costs

Push this to a busy shop — 5,000 enquiries a month, ten times the volume — and the bill lands somewhere near $20, not $28. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules stay at a cent, and AWS Budgets stays free. What scales is the genuinely usage-priced work —

roughly $13.50 of Bedrock for ten times the replies, a couple of dollars more of DynamoDB, a bit more SES, Lambda, and logs. Even then, the dominant cost is still the one model call per reply, and the thing answering the actual question — match plus tracking — remains close to free.

The honest way to read this: the AWS bill is rounding error against the alternative. A person answering 500 “where’s my order?” messages a month by hand is hours of dull lookups; the same hours at 5,000 is a part-time job. $2.80, or even $20, buys those hours back — and the few messages that genuinely need a human still get one, with the full context already gathered.

DESIGN RULES THAT SHAPED THE COST

- Pay per message, not per hour. No always-on compute means no idle bill.
- Spend the model sparingly. One Haiku call per reply, and only to write — never to match or to track.
- Cheap work stays cheap. The match and the tracking lookup are plain Python and one API call.
- Know your one fixed cost. Secrets Manager is the only line that bills while the system sleeps.
- Cache to protect the carrier bill. The tracking cache keeps repeat asks from becoming repeat charges.

PART 7 OF 7

JUNE 20, 2026 PART 7 OF 7 · ORDER STATUS RESPONDER SERIES ~7 MIN READ

Engineering reference: the order status responder architecture

This is the order status responder with the friendly labels removed: the real resource names, the runtime, the table key schemas, the inbound mail rules, the schedules, 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.
- Four DynamoDB tables, all on-demand: orders mirror, tracking cache (with TTL), threads, and an append-only audit log.
- Inbound is SES for email and two Lambda Function URLs for the SMS and web-chat webhooks — no API Gateway.
- Two EventBridge Scheduler rules: a 15-minute Drive sync and a daily stuck-order sweep.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by the replier. 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; inbound HTTP arrives on Lambda Function URLs, email arrives through SES, and work is buffered on a single SQS queue.

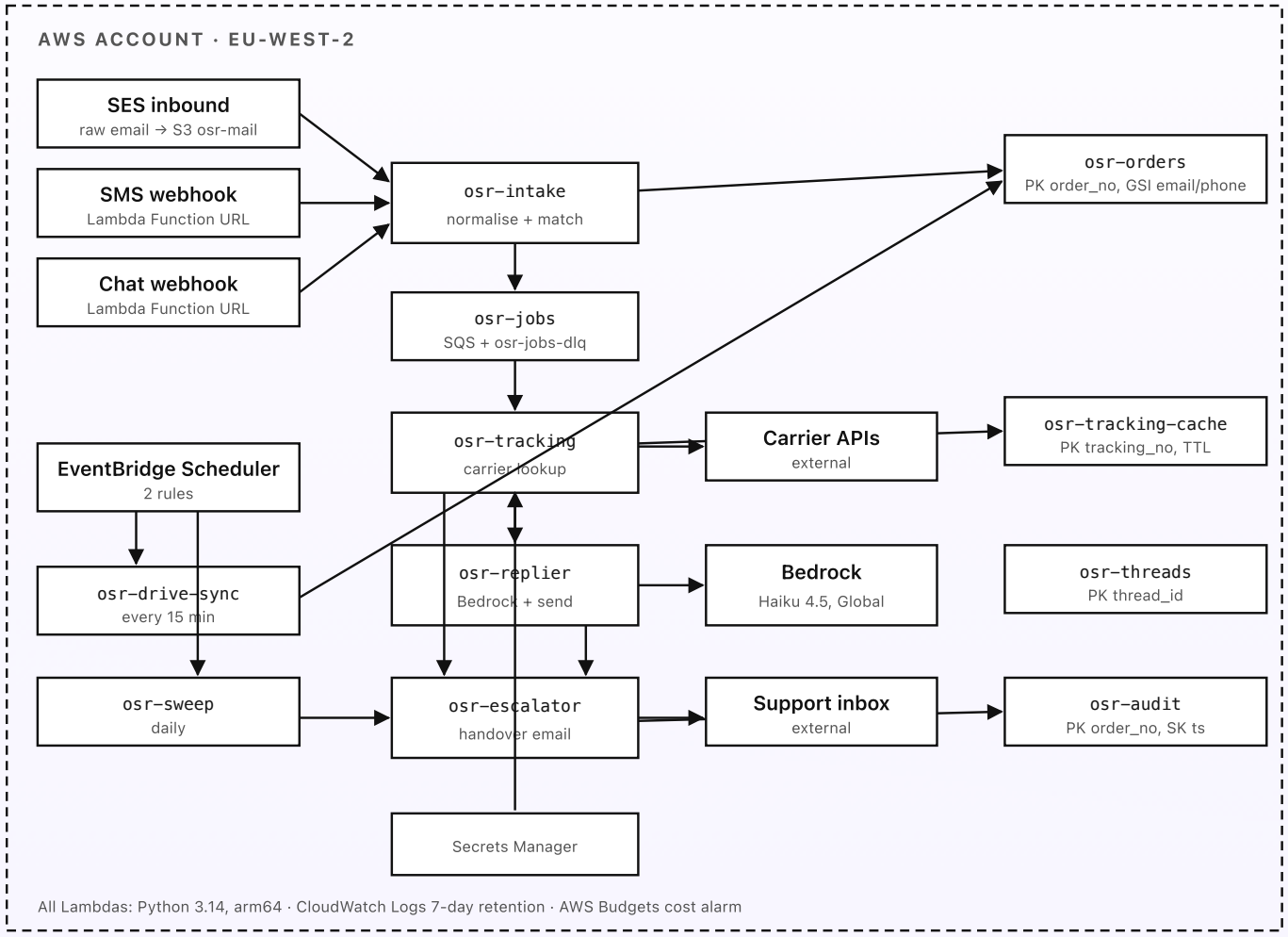

AWS ACCOUNT · EU-WEST-2
SES inbound
raw email → S3 osr-mail
SMS webhook
Lambda Function URL
Chat webhook
Lambda Function URL
osr-intake
normalise + match
osr-orders
PK order_no, GSI email/phone
osr-jobs
SQS + osr-jobs-dlq
osr-tracking
carrier lookup
Carrier APIs
external
osr-tracking-cache
PK tracking_no, TTL
EventBridge Scheduler
2 rules
osr-drive-sync
every 15 min
osr-replier
Bedrock + send
Bedrock
Haiku 4.5, Global
osr-threads
PK thread_id
osr-sweep
daily
osr-escalator
handover email
Support inbox
external
osr-audit
PK order_no, SK ts
Secrets Manager
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm

Fig 7. The order status responder drawn for engineers: three inbound edges, an SQS-buffered chain of seven Lambdas, four DynamoDB tables, Bedrock called only by the replier, 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 (`osr-jobs`, with `osr-jobs-dlq` as its dead-letter queue after five attempts) decouples intake from the slower carrier and model calls.

- `osr-intake` — the only public surface. Backs the SMS and web-chat Function URLs and the SES inbound rule; normalises the message, runs the order match (number, then sender), and enqueues a job. Unmatched and ambiguous go straight to `osr-escalator`.
- `osr-tracking` — SQS-triggered. Reads the matched order, checks `osr-tracking-cache`, and on a miss calls the carrier adapter with a key from Secrets Manager; normalises to the five states and writes the cache.
- `osr-replier` — makes the single Bedrock call, validates the draft against the facts, sends on the original channel (SES out, or the SMS/chat provider), and writes `osr-audit`.
- `osr-escalator` — builds the handover (message + order + raw tracking + reason), de-duplicates against open escalations in `osr-threads`, and emails the support inbox via SES.
- `osr-drive-sync` — scheduled. Pulls the Drive sheet, writes a snapshot to S3, and upserts rows into `osr-orders`.

- `osr-sweep` — scheduled. Re-checks in-flight orders against the carrier and escalates those past the no-movement threshold.
- `osr-channel-out` — thin sender helper invoked by the replier and escalator; wraps SES and the SMS/chat provider so the calling functions don't each hold provider logic.

Data stores, schedules, and mail

- **DynamoDB (all on-demand).** `osr-orders` — PK `order_no`, GSIs on `email` and `phone` for the sender fallback. `osr-tracking-cache` — PK `tracking_no`, with a `ttl` attribute set a few minutes out so cached results expire automatically. `osr-threads` — PK `thread_id`, the open-escalation and conversation state used for de-duplication. `osr-audit` — PK `order_no`, SK `ts`, append-only, holding each reply and the exact facts it was built from.
- **S3.** `osr-mail` — raw inbound email from the SES receipt rule, plus the latest Drive-sheet snapshot written by `osr-drive-sync`. Lifecycle expiry on the raw-mail prefix after 30 days.
- **SES.** One inbound receipt rule on the support/order address that writes to `osr-mail` and invokes `osr-intake`; verified domain and DKIM for outbound replies and escalation mail.
- **EventBridge Scheduler.** Two rules — `osr-drive-sync` at `rate(15 minutes)`, and `osr-sweep` at a daily `cron` (early morning, before business hours).
- **Secrets Manager.** One secret per carrier API and one for the SMS provider; fetched at call time, never in env vars or the sheet.

- **Bedrock.** Model id `anthropic.claude-haiku-4-5` via the Global cross-Region inference profile, invoked only by `osr-replier`.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `osr-intake` can read `osr-orders` and send to `osr-jobs`; it cannot read the audit table or call Bedrock. `osr-tracking` can read the cache, write the cache, and read only the carrier secrets — not the SMS-provider secret. `osr-replier` is the only role with `bedrock:InvokeModel`, and it's scoped to the one Haiku profile; it can send via SES and write `osr-audit` but cannot delete from any table. `osr-escalator` can write `osr-threads` and send SES, nothing more. The scheduled functions hold the narrow Drive and carrier permissions they need and no inbound surface at all. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call for capacity and is not a data store. An AWS Budgets alarm watches the monthly spend and notifies if it drifts above a few dollars — the early signal that a carrier API or a loop is misbehaving.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Seven small Lambdas beat one that does everything; the queue decouples the slow calls.
- One public surface. Only `osr-intake` is reachable from outside, on Function URLs and the SES rule.
- Least privilege, per role. Only the replier can call Bedrock; only tracking reads carrier secrets.
- State in DynamoDB, blobs in S3. Tables for orders, cache, threads, and audit; S3 for raw mail and the sheet snapshot.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called once per reply.
- A budget alarm is a smoke detector. The cheapest way to learn something looped is a Budgets alert.