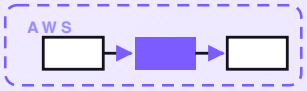
7-PART SERIES · FREE COMPANION



Shipping notifier

A serverless notifier that watches each order's shipping status — placed, shipped, out for delivery, delivered, delayed — and sends the right friendly update at every step, so the customer is always in the loop and nobody on your team has to lift a finger. When a delivery runs late, it warns the customer and the owner early. It cuts "where is my order?" emails, respects quiet hours, and never double-sends. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Shipping notifier

PART 1 OF 7

MAY 18, 2026 PART 1 OF 7 · SHIPPING NOTIFIER SERIES ~5 MIN READ

A shipping notifier on AWS for a few dollars a month

A small business ships more orders than anyone keeps in their head. The package that left the warehouse this morning and the customer doesn't know yet. The order that's out for delivery today and would love a heads-up. The one that was supposed to arrive Tuesday and is now sitting in a depot two states away while the customer wonders if it's lost. Keeping every customer in the loop is the kind of work that feels small until you have fifty orders in flight and an inbox full of "where is my order?" This post walks through the design of a small notifier that watches every order, sends the right friendly update at each step, and warns both the customer and you the moment a delivery runs late.

KEY TAKEAWAYS

- Three sources for orders: a Drive sheet from your store, a carrier webhook lane, and an inbox forwarding lane.
- Every order ends in one of five moves on each check: nothing, shipped, out for delivery, delivered, or delayed.
- One update per status, ever. The notifier writes down each send so it never repeats one.
- Updates respect quiet hours and let the customer unsubscribe; a delay warns both customer and owner.
- Designed on AWS for about $2.40/month at typical small-business volume.

The whole system on one page

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

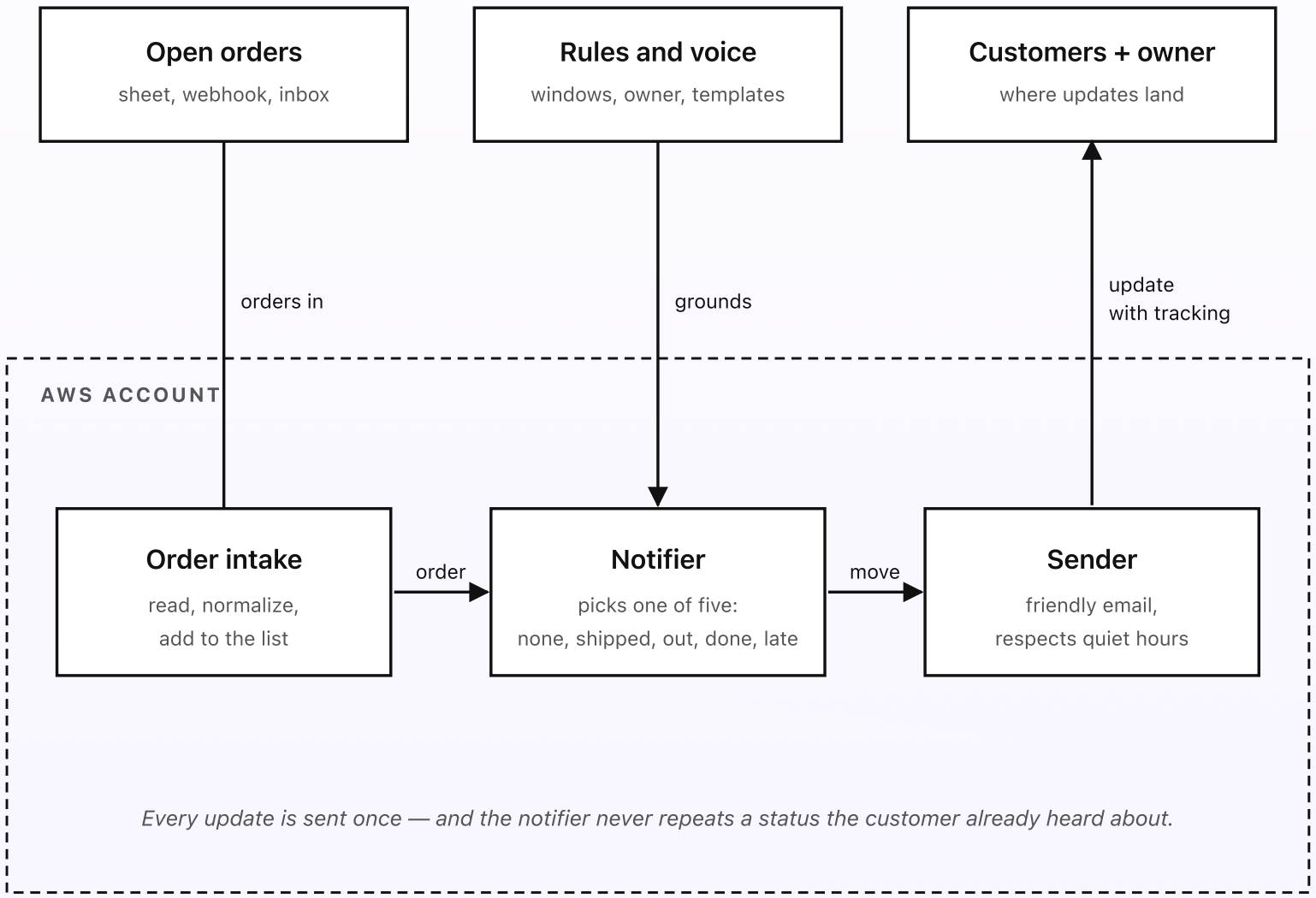

Open orders
sheet, webhook, inbox
Rules and voice
windows, owner, templates
Customers + owner
where updates land
orders in
grounds
update
with tracking
AWS ACCOUNT
Order intake
read, normalize,
add to the list
order
Notifier
picks one of five:
none, shipped, out, done, late
move
Sender
friendly email,
respects quiet hours
Every update is sent once — and the notifier never repeats a status the customer already heard about.

Fig 1. Three sources outside, three pieces inside AWS. Orders flow in from a Drive sheet, a carrier webhook lane, and an inbox forwarding lane. The Notifier runs on a schedule and picks one of five moves. The Sender emails the right update to the right customer at the right time.

What you set up once (the outside)

- **Open orders.** A Google Sheet in a Drive folder, one row per order: number, customer name, customer email, carrier, tracking number, current status (placed, shipped, out for delivery, delivered, delayed), expected delivery date, an unsubscribed flag, and a link to the order. You export this from your store once and let new orders and tracking updates flow in from two other lanes covered in Part 2 — a carrier webhook lane (the carrier posts each tracking change to a small endpoint) and an inbox-forwarding lane (forward a carrier email to a dedicated address and the notifier reads it into a proposed status change for one-tap approval).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the quiet hours (when not to email), the expected-delivery windows per carrier or shipping method (so the notifier knows when an order is running late), the owner who gets delay alerts, and the unsubscribe policy. The *voice* doc holds one update template per shipping step — the “it shipped,” the “it’s out for delivery,” the “it arrived,” and the “it’s running late” message.
- **Customers and owner.** The customer receives the friendly updates as their order moves along. The internal owner receives the alert when a delivery runs late. Updates land with the order number, the new status, a tracking link the customer can click to see details, and an unsubscribe link so anyone who’d rather not hear about it can opt out in one tap.

What runs on every check (the inside)

- **The order intake.** Three sources feed the list. The Drive sheet is the canonical store. New orders and tracking updates can also be added via the carrier webhook lane (the carrier posts each tracking change to a small endpoint the moment it happens) and the inbox forwarding lane (forward a carrier email to `tracking@your-company.com`, the notifier uses Bedrock Haiku 4.5 to read the email and pull out the order and the new status, then drops a one-tap approval card in the team's Slack to confirm before the row is updated).
- **The notifier.** Runs on a schedule (every 30 minutes during the day). Reads the open orders. For each one, it looks at the current status and compares it to the last status it told the customer about. Picks one of five moves. *Nothing*: status hasn't changed since the last update — do nothing. *Shipped*, *out for delivery*, *delivered*: the status moved forward — send the matching update once. *Delayed*: the order passed its expected delivery date without arriving — warn the customer and alert the owner. The notifier itself doesn't call a model on the check — the move logic is plain Python.
- **The sender.** Reads the voice doc, formats the update message for the chosen move, and sends it. Email goes through SES outbound. It honors quiet hours (no sends between 9pm and 8am local by default) and checks the unsubscribe flag before every send. Every send writes a row in DynamoDB so the next check can tell which updates already went out and never repeats one. A monthly summary writes a one-paragraph narrative: orders shipped, orders delivered, average days in transit, and how many ran late.

In plain words

Jordan ordered a pair of boots and chose standard shipping. The warehouse hands the parcel to the carrier on Monday afternoon. Within the half hour, the carrier's webhook tells the notifier the status is now *shipped*, and on the next check Jordan gets a friendly email: "Good news — your order #5120 is on its way! Track it here." Wednesday morning the parcel goes out for delivery; Jordan gets a quick "it's out for delivery today" note. Wednesday evening it's delivered, and a short "it arrived — enjoy!" goes out. Jordan never had to ask. Now picture the other order: it was due Tuesday and Wednesday comes and goes with no delivery. The notifier marks it delayed, sends Jordan a calm heads-up with what it knows, and tells the owner so a human can chase the carrier — before Jordan has to email asking where it is.

The cost of running this is about $2.40 a month at SMB volume. The cost of *not* running it is the inbox full of "where is my order?" emails, the customer who assumes the worst because nobody told them anything, and the late delivery you only find out about when it's already a complaint.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every update ships with full context — order number, new status, tracking link, unsubscribe link. The customer never has to dig.
- Five moves, always. Nothing, shipped, out for delivery, delivered, delayed. There is no sixth.
- One update per status. The same order never gets two “shipped” emails, even if the carrier reports it twice.
- Quiet hours and unsubscribes are respected. An update is a finite resource; bad timing burns it.
- A delay warns both the customer and the owner, so a human can act before it becomes a complaint.
- Every send is logged. Audit an order next month and you can see every update that went out.

Why this shape

Most small shops handle shipping updates in one of three ways: a person who copies tracking numbers into emails when they have time, the carrier’s own notifications that customers ignore because they look like spam, or nothing at all. The person works until they’re busy — and the busiest weeks are the ones with the most orders in flight. The carrier’s emails come from an address the customer doesn’t recognize, in a voice that isn’t yours. And “nothing at all” is how a happy customer turns into an anxious one who fills your inbox.

The setup above keeps the order list in a sheet the team already exports, but adds a small system that *looks at* that list on a schedule and acts only when an order has actually moved. Updates go out in your voice, with your branding, carrying a tracking link so checking is one click. They go out once per step, never repeated. They respect quiet hours so nobody gets a 2am ping. And they catch a late delivery early enough that a human can step in. The notifier is invisible most of the time; it only speaks up when there's genuinely something the customer wants to know.

The next four posts walk through each piece in turn: how an order gets watched, how a shipping update gets sent, how a message reaches the customer, and how a late delivery gets caught. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 18, 2026 PART 2 OF 7 · SHIPPING NOTIFIER SERIES ~4 MIN READ

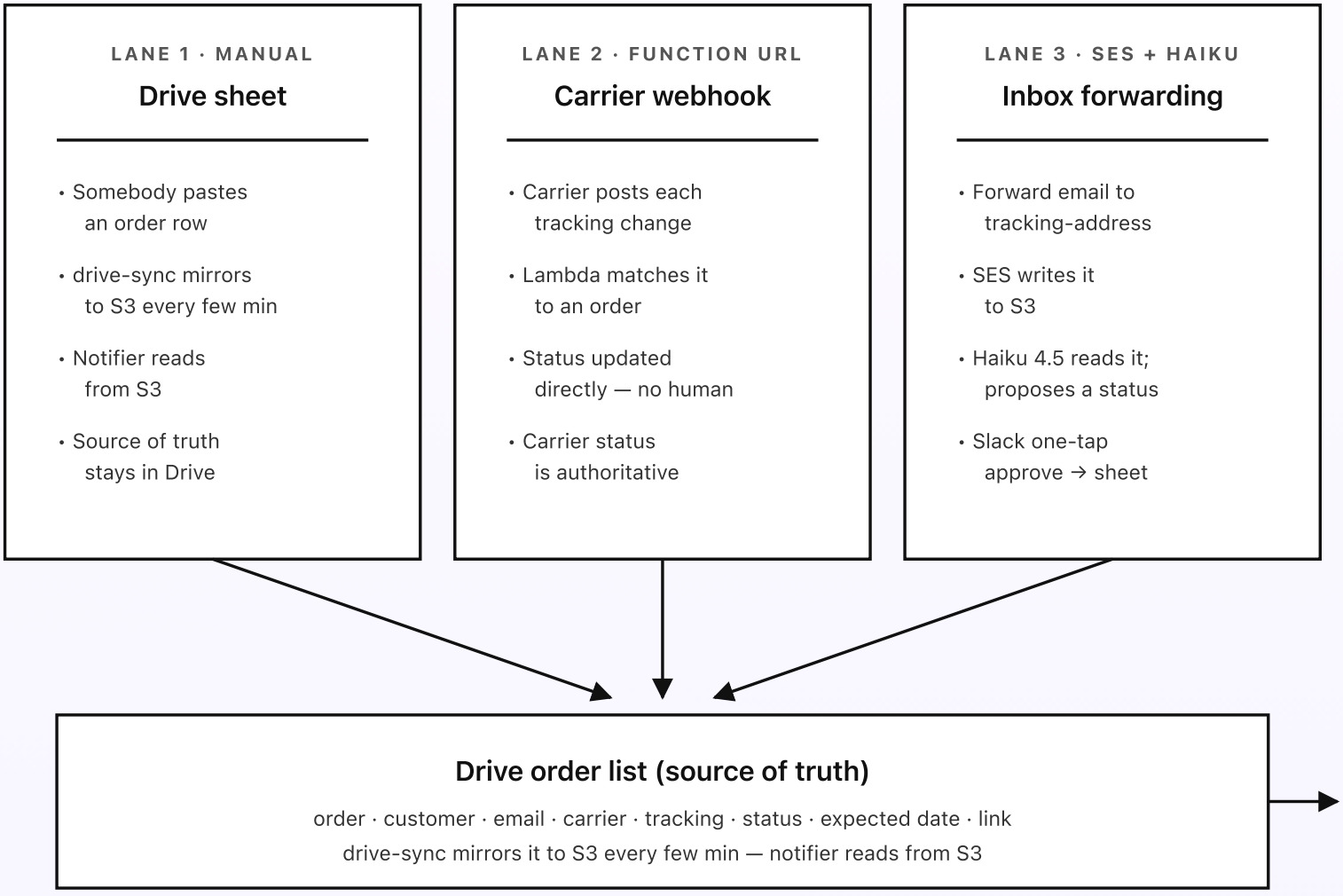
How an order gets watched

The notifier only watches what's in the order list. So the first job is making sure the list actually reflects what your business has in flight, and that each order's status stays current as the parcel moves. There are three ways an order gets in and stays up to date: somebody types or pastes a row in the Drive sheet, the carrier posts each tracking change to a small endpoint, or somebody forwards a carrier email and the notifier reads it. The first one is obvious. The other two exist because in real life nobody types a status update for the parcel that just changed depots.

KEY TAKEAWAYS

- Three intake lanes feed one order list: the Drive sheet, a carrier webhook, and an inbox-forwarding lane.
- The carrier webhook posts each tracking change to a Lambda Function URL the moment it happens.
- Forwarded carrier emails are read by Bedrock Haiku 4.5, which proposes the order and the new status.
- Every parsed status change goes to a team Slack for one-tap approval before it lands in the list.
- The Drive sheet stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one order list



The Drive sheet stays the source of truth — the webhook and the inbox lane keep it current.

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the carrier webhook updates it in real time and the inbox lane proposes status changes for human approval. The drive-sync Lambda mirrors the sheet to S3 so the notifier can read it without hitting Drive on every check.

Lane 1: the Drive sheet itself

The simplest lane. Open the order list in Drive, paste in the day's new orders (most stores can export a CSV), and save. The columns are short: order number, customer name, customer email, carrier, tracking number, current status, expected delivery date, and a link to the order. A small Lambda — `drive-sync` — runs every few minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://sn-orders-source/orders.csv` if the sheet has changed since the last sync. The notifier reads from S3, not Drive directly. That keeps Drive API calls predictable and gives you S3 versioning for free, so a bad bulk-edit can be rolled back in one click.

This lane covers the cases where you already have an order, you know its tracking number, and you can spend a few seconds pasting it in. Most orders go in this way at the start of their journey.

Lane 2: the carrier webhook (the lane that does the heavy lifting)

Most carriers can post a tracking update to a web address you give them every time a parcel's status changes — picked up, in transit, out for delivery, delivered. Set up a small Lambda Function URL (a plain web address that runs a tiny piece of code, no API Gateway needed) and hand it to the carrier as the place to post updates. When a parcel changes status, the carrier posts the tracking number and

the new status to that address. The Lambda matches the tracking number to an order in the list and updates the status field directly.

This lane has no human in the loop, on purpose. The carrier's own status is the authoritative truth about where a parcel is — there's nothing for a person to approve or correct. The Lambda verifies the request really came from the carrier (using a shared secret stored in Secrets Manager), matches the tracking number, and writes the new status. If a tracking number doesn't match any order, the update is parked in a small "unmatched" list and surfaced in the weekly digest so somebody can fix the row. The webhook is what keeps the order list current minute to minute without anyone touching it.

Lane 3: inbox forwarding (the catch-all)

Not every carrier offers a clean webhook, and some send updates only by email. For those, set up a dedicated inbound address — something like `tracking@your-company.com` — via Amazon SES. Anyone on the team forwards a carrier email to that address and the notifier takes it from there. SES writes the raw email to `s3://sn-raw-mime/`. The S3 write triggers a parser Lambda.

The Lambda calls Bedrock Haiku 4.5 to read the email and emit a structured change: which order it's about (matched by tracking number or order number in the text) and the new status. The model prompt is short: "Read this carrier email. Return JSON only: the tracking number and the new shipping status. Do not invent a status that isn't stated." The output goes to a small Slack interactive message that pings the team: the proposed change and three buttons — *approve*, *edit*, *discard*. On *approve*, a Lambda updates the status in the Drive sheet via the Sheets API. On *edit*, the team member gets a fillable modal pre-populated with the

proposal. On *discard*, the message is logged and the email moved to a discarded prefix in S3 for audit.

The reason this lane keeps a human in the loop — while the webhook lane doesn't — is that reading a free-text email is fuzzier than reading a structured webhook. A status the model misread is worse than one that never made it in: it would send the customer a "delivered" email for a parcel still in transit. So a person glances at the proposal before it goes anywhere.

Why the list stays the source of truth

Three lanes in, but only one place where the notifier actually looks. That's a deliberate constraint. If two lanes both wrote directly to the notifier's state, every "why did this email go out?" question would mean checking three places. Funneling everything through the Drive sheet means there is exactly one row per order, and any team member can read or edit any of it without learning a new tool. The convenience lanes are first-class for keeping status current, but they always pass through the sheet on the way.

Next post: how the notifier actually reads the order list, compares each status to what was last sent, and picks one of five moves.

PART 3 OF 7

MAY 18, 2026 PART 3 OF 7 · SHIPPING NOTIFIER SERIES ~5 MIN READ

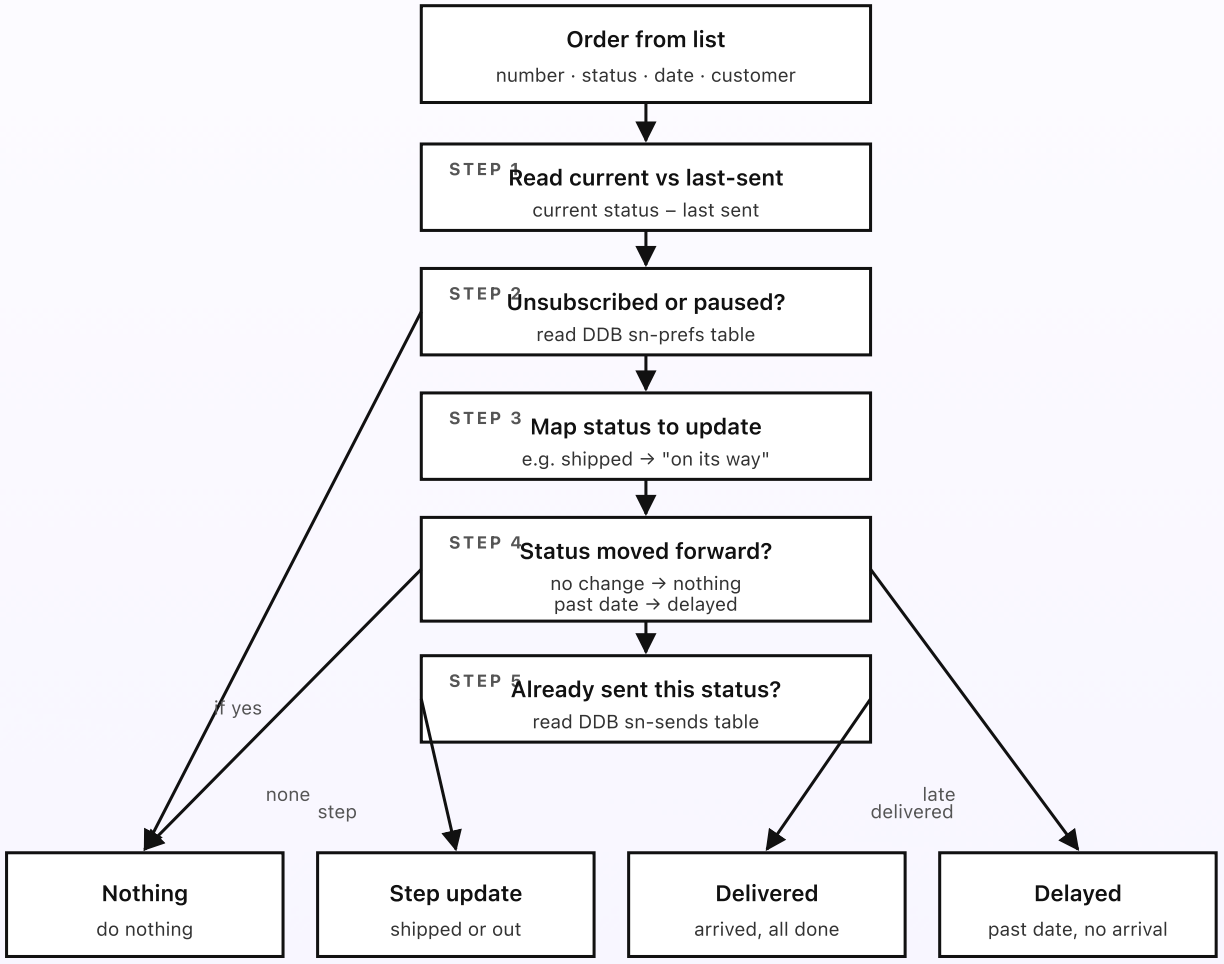
How a shipping update gets sent

Every 30 minutes during the day, an EventBridge Scheduler rule fires the notifier Lambda. The Lambda reads the order list, looks at one row at a time, compares the order's current status to the last status it told the customer about, and decides whether to do nothing or to send an update — and if so, which one. The whole decision is plain Python. No model. No guessing. Every setting lives in the rules doc, where the owner can edit it without a deploy.

KEY TAKEAWAYS

- The notifier runs on a schedule via EventBridge Scheduler, every 30 minutes during the day.
- It compares each order's current status to the last status it actually told the customer about.
- Five moves per order, every check: nothing, shipped, out for delivery, delivered, delayed.
- DynamoDB tracks the last status sent per order so the same update never goes out twice.
- The notifier itself never calls a model. The decision is entirely deterministic.

The decision flow, per order



The rules doc holds every setting — change a window and the next check uses the new value.

Fig 3. The notifier's decision tree, per order, per scheduled check. Five steps decide which of five moves applies. The rules doc holds every setting; the notifier only enforces them.

Status stages: placed, shipped, out for delivery, delivered, delayed

Every order moves through a small, fixed set of stages. *Placed* is the start — the customer has ordered but nothing has shipped, so there's nothing new to say. *Shipped* means the parcel is with the carrier and on its way. *Out for delivery* means it's on a vehicle and arriving today. *Delivered* means it's at the door. *Delayed* is the off-path stage: the order passed its expected delivery date without arriving. Each of these (except placed) has its own friendly update in the voice doc.

The stages exist for a reason. A "shipped" note sets the customer's expectation and gives them a tracking link. An "out for delivery" note tells them to keep an eye out today, which cuts the "is it coming?" question. A "delivered" note closes the loop and is a nice moment to say thanks. Different stages do different jobs; the updates reflect that.

Per-order overrides exist too. The order list has an optional column called `mute_until`. Put a date there and the notifier holds all updates for that one order until the date passes — useful for a pre-order that shipped early, or an order you're handling by hand.

Five moves, always

Every order, every check, lands in exactly one of five moves. The names are simple on purpose.

- **Nothing.** The status hasn't changed since the last update, or the customer has unsubscribed, or the order is muted. Do nothing. Most orders, most checks, are nothing.
- **Shipped.** The status just became shipped and no "on its way" email has gone out. Send the shipped update with the tracking link. Write a row to the `sn-sends` DynamoDB table marking that the shipped update fired.
- **Out for delivery.** The status became out for delivery and no "arriving today" email has gone out. Send that update. Write the send to `sn-sends`.
- **Delivered.** The status became delivered and no "it arrived" email has gone out. Send the delivered update with a friendly thank-you. Write the send to `sn-sends`.
- **Delayed.** The order passed its expected delivery date without a delivered status. Warn the customer and alert the owner. Mark the order as delayed in DynamoDB. Part 5 covers this move in full.

State that makes the decision deterministic

The notifier reads two DynamoDB tables every check. `sn-sends` records every update that's gone out: `(order_id, status, sent_date, sent_via)`. `sn-prefs` records each customer's preference: `(order_id, unsubscribed, mute_until)`. With those two tables, the move-decision logic is a few dozen lines of Python and zero guesswork. A given order with a given status and a given send

history always produces the same move. Re-running the check produces no extra emails (because the state in DynamoDB shows what already fired).

This is why the same order never gets two “shipped” emails. Carriers sometimes report the same status twice, or the webhook fires a retry. The notifier doesn’t care: it checks `sn-sends` first, sees the shipped update already went out, and lands at *nothing*.

Why the scheduled check uses no model

The notifier could call a model on the check to write a smarter update, or to decide whether to send at all. It doesn’t. Two reasons. First, the check should be the one part of the system that is utterly predictable — if the status moved to shipped and no shipped email went out, the email fires. A model in that loop introduces variance the team can’t reason about. Second, model calls cost money, and most orders on most checks have nothing new, so the call would be wasted.

Bedrock fires elsewhere — on the inbox forwarding lane in Part 2, and on the monthly summary mentioned in Part 6. Not on the scheduled check. The notifier itself is plain Python that reads a list and writes events.

Next post: how a message reaches the customer, how quiet hours and unsubscribes are honored, and what context every update carries.

PART 4 OF 7

MAY 18, 2026 PART 4 OF 7 · SHIPPING NOTIFIER SERIES ~5 MIN READ

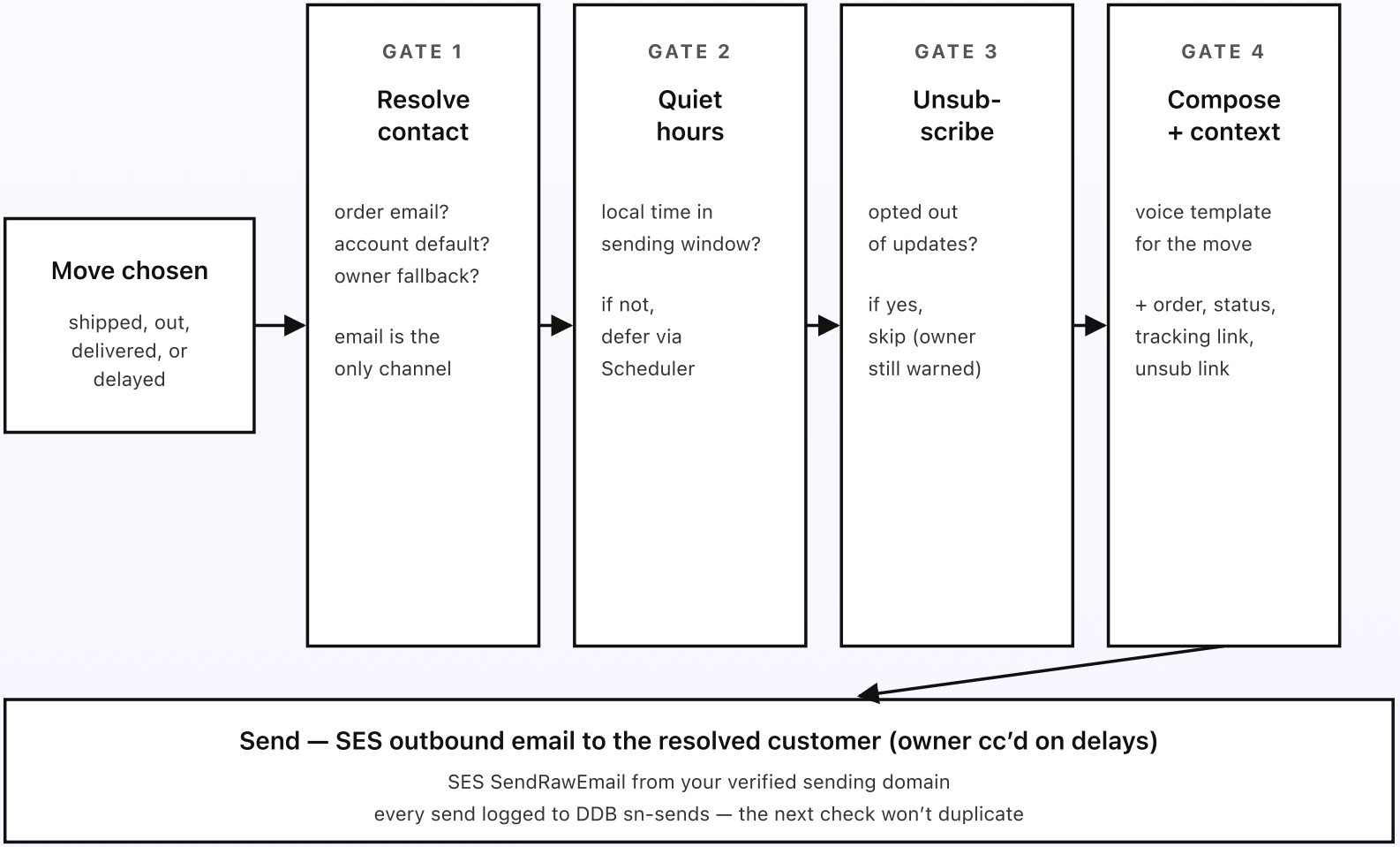
How a message reaches the customer

The notifier picked a move — shipped, out for delivery, delivered, or delayed. Now the sender Lambda has to figure out who to email, at what time of day, whether they even want the email, and with what context attached. Get any of those wrong and the update is worse than no update: a 2am email, a note to a customer who opted out, a “your order shipped” with no tracking link. Four small guardrails sit between the move and the actual send.

KEY TAKEAWAYS

- Contact resolution: the order's own email beats the account default beats the configured fallback.
- Email is the channel; the sender uses SES outbound with your verified sending domain.
- Quiet hours defer an update to the next morning so nobody gets a middle-of-the-night email.
- Every update ships with the order number, the new status, a tracking link, and an unsubscribe link.
- A delayed move also emails the owner; the customer still gets their warning too.

Four guardrails on every send



Every gate is a deterministic check — no model calls, no surprise behavior on a quiet Tuesday.

Fig 4. Four guardrails between the move and the sent update. Resolve the contact. Honor quiet hours. Check the unsubscribe. Compose with full context. Then send via SES and log it so the next check doesn't duplicate.

Gate 1: resolve the contact

Three places the sender Lambda looks for the customer's email, in order. First, the order list's per-order `customer_email` column — the email the customer gave at checkout, which is right almost every time. Second, an account-default contact in the rules doc (for shops that keep a separate contacts list keyed by customer name). Third, the configured owner fallback — so an update for an order with a missing email goes to a human instead of vanishing. The fallback should rarely fire; if it does, the weekly digest names every order that hit it so the row can be fixed.

Email is the only channel here, on purpose. Shipping updates are exactly the kind of message customers expect in their inbox, and email gives you a clean unsubscribe and a clear sender identity. The sender uses SES outbound with your verified sending domain so the update comes from you, in your voice — not from an address the customer doesn't recognize.

Gate 2: quiet hours

The check runs every 30 minutes, around the clock, because carriers deliver and scan parcels at all hours. That means a "delivered" status can land at 11pm, and an "out for delivery" scan can happen at 6am. Without a guard, the customer would get a ping in the middle of the night.

Gate 2 reads the rules doc's quiet-hours setting (default 9pm to 8am, configurable per business). If the current local time is in the quiet window, the sender creates a one-off EventBridge Scheduler rule that fires at the start of sending hours the next morning and exits without sending. The Scheduler invokes the same sender Lambda with the same payload at the deferred time, where Gate 2 will let it through. A delayed warning is the one exception you can configure to bypass quiet hours, since a customer would usually rather know late than not at all.

Gate 3: unsubscribe check

Every update carries an unsubscribe link. If a customer clicks it, a Function URL Lambda writes an `unsubscribed` flag to the `sn-prefs` table for that order. Gate 3 reads that flag before every send and, if the customer opted out, skips the send and logs it.

There's one carve-out. A *delayed* warning still reaches the owner even if the customer unsubscribed, because the owner needs to know a delivery is running late regardless of the customer's email preferences. The customer's own delayed warning still respects their unsubscribe — if they opted out, they won't get it. The owner's alert is internal and isn't a marketing message, so it isn't subject to the customer's opt-out.

Gate 4: compose with full context, then ship

The voice doc has one email template per move: a short, friendly message with placeholders for the order number, the new status, the carrier, and the tracking

link. The sender Lambda fills the placeholders, appends an unsubscribe link, wraps it in a small branded HTML email, and ships it via SES `SendRawEmail` from your verified sending domain. The sending identity and DKIM keys live in SES; nothing sensitive sits in the Lambda.

The tracking link is the most useful thing in the email. Rather than make the customer copy a tracking number into a carrier's site, the link goes straight to the carrier's tracking page for that parcel, pre-filled. Checking is one click.

A *delayed* move adds a second recipient: the owner named in the rules doc. The customer still gets their warning (unless they unsubscribed). But the owner now sees it too, with the order number, the expected date, and how many days late it is, so they have what they need to chase the carrier. The delayed template is calmer than you might expect — it tells the customer what's known, sets a new expectation, and avoids over-promising.

Every send — any move, customer or owner — writes a row to `sn-sends` in DynamoDB. The next check reads that row and knows not to send the same update again.

Why the guardrails exist

None of these gates are exotic. They're the kind of small care a thoughtful person would take if they were sending the updates by hand — check you have the right email, don't email at 2am, don't email someone who asked you to stop, include enough context that the customer doesn't have to ask a follow-up question. Putting them in code as four small sequential gates makes them part of the design, not a feature you're trusting the writer of any one email to remember.

Next post: how a late delivery gets caught — how the notifier spots an order that's past its date, warns the customer, alerts the owner, and logs the delay so nothing slips silently.

PART 5 OF 7

MAY 18, 2026 PART 5 OF 7 · SHIPPING NOTIFIER SERIES ~5 MIN READ

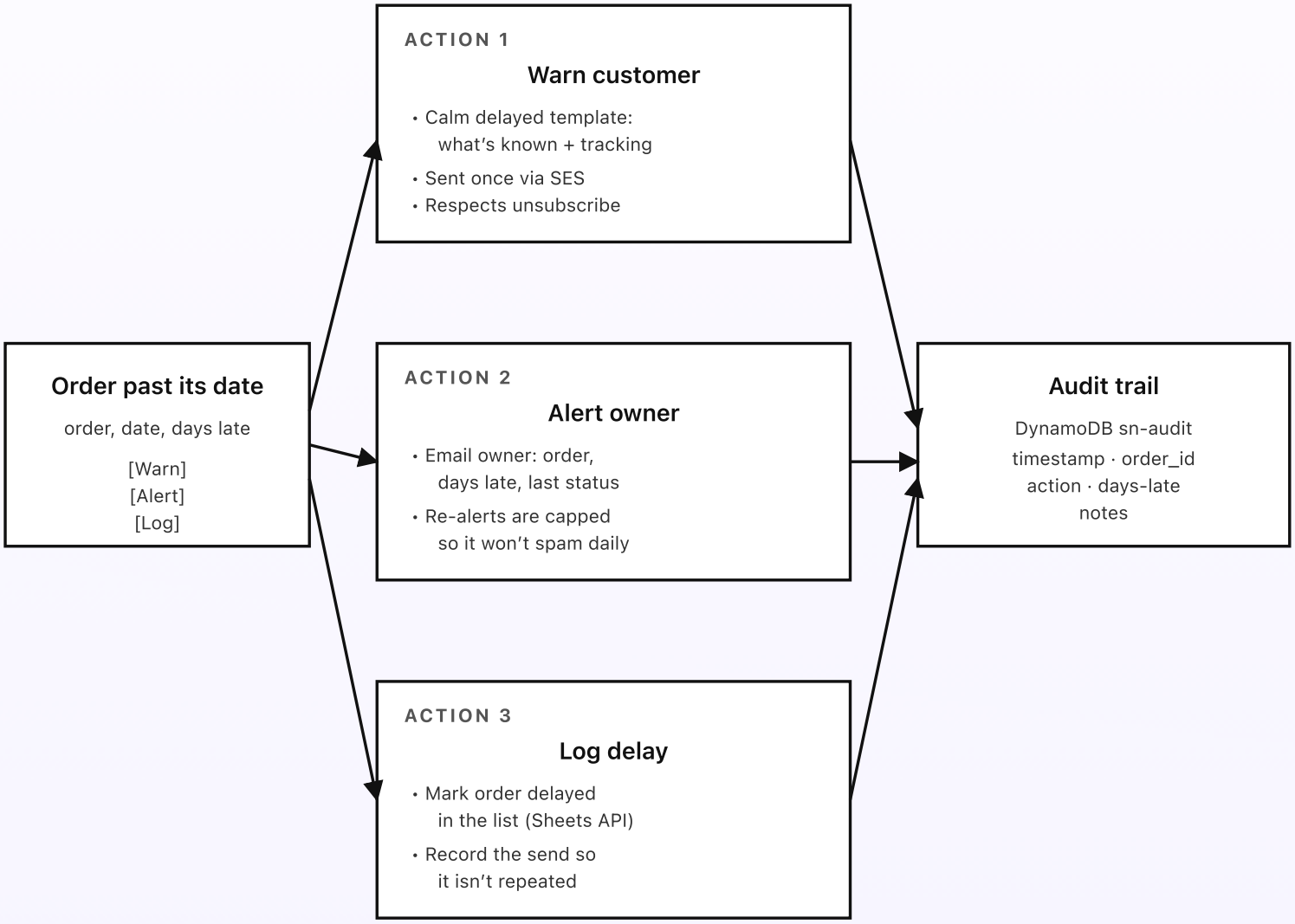
How a late delivery gets caught

An order was supposed to arrive Tuesday. It's Wednesday morning and the status still says out for delivery. What happens? The honest answer is "three things, in order." This post walks through what the notifier does when an order passes its expected date without arriving — warn the customer, alert the owner, log the delay — and how the order list, the send history, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Three things on a delay: *warn the customer* (calm heads-up), *alert the owner* (so a human can act), *log the delay*.
- Each step updates the order list via the Sheets API and writes an audit row.
- A delay marks the order so the customer warning is sent once, not on every check.
- The delay check is bounded — one customer warning per delay, with owner re-alerts capped.
- The whole flow is plain Python; the rules doc holds the expected-delivery windows.

Three things on a delay



The customer hears once, the owner can act, and the trail is complete — a delay never slips by silently.

Fig 5. Three things on a late delivery, three different effects. Warn the customer with a calm heads-up. Alert the owner so a human can chase the carrier. Log the delay so the customer isn't warned twice. Every action writes to the audit trail.

How a delay gets spotted

The notifier knows each order's expected delivery date — either set per order in the list, or worked out from the carrier and shipping method using the windows in the rules doc ("standard ground: 5 business days; express: 2"). On each check, for any order that isn't yet delivered, the notifier compares today's date against the expected date. If the order is past it, the move is *delayed*.

A small grace setting in the rules doc (default one day) keeps the notifier from crying wolf on a parcel that's a few hours behind. A delivery that's expected by end of Tuesday and arrives Wednesday morning is normal; the grace window absorbs it. Only orders genuinely past their window get flagged.

Action 1: warn the customer (the calm heads-up)

The first thing that happens is the customer hears about it — from you, before they have to ask. The sender composes the delayed template: what's known ("your order is taking a little longer than expected"), a new rough expectation if one can be worked out, and the tracking link so they can see for themselves. The tone is calm and honest — it doesn't over-promise a new date the carrier hasn't confirmed.

This warning goes out once per delay, not on every check. The moment it's sent, the notifier records it in the send history, so the next check sees the delayed warning already went out and doesn't repeat it. And it still respects the unsubscribe: a customer who opted out of updates won't get the warning, though the owner alert below still fires.

Action 2: alert the owner (so a human can act)

A customer warning tells the customer the order is late. It doesn't fix anything. The second thing that happens is the owner named in the rules doc gets an email with the order number, the days late, the last known status, and the tracking link — everything they need to call the carrier or open a claim. The point is to put a human on it early, while there's still time to do something, rather than waiting for the customer to complain.

Owner re-alerts are capped. A parcel can sit stuck in a depot for days, and the owner doesn't need a fresh email every 30 minutes about the same stuck order. The rules doc has a configurable `owner_realert_days` setting (default two). After the first alert, the owner is re-alerted at most that often until the order is delivered or resolved. This is the same care the customer warning gets: say it once, clearly, then stay quiet until something changes.

Action 3: log the delay (so nothing slips)

The third thing is bookkeeping, and it's what keeps the whole flow honest. The notifier marks the order's status as *delayed* in the order list via the Sheets API, so anyone looking at the sheet can see which orders are running late. It records the

customer warning and the owner alert in the `sn-sends` history so neither is repeated. And it writes an `action: delayed` row to `sn-audit` with the order, the days late, and a short note.

If the order later gets delivered — the carrier finally scans it — the next check sees the delivered status, sends the delivered update (which doubles as the "it finally arrived" close), and the delay is resolved. The audit trail keeps both rows: the day it was flagged delayed, and the day it arrived. The monthly summary in Part 6 reads those rows to report how many orders ran late and how late they were on average.

Every action is logged, every delay is accounted for

The `sn-audit` table records every delay flag, customer warning, and owner alert with the timestamp, the order, and the days late. If a customer ever asks "why didn't anyone tell me?" — or a teammate asks "did we warn them?" — the answer is one lookup away. The trail is the only memory the next person has when they pick up a stuck order someone else was handling.

This kind of accounting matters most for the orders that go wrong, which are exactly the ones a busy team is most likely to lose track of. A happy order needs no memory; a late one needs all of it.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why it's less than you'd guess.

PART 6 OF 7

MAY 18, 2026 PART 6 OF 7 · SHIPPING NOTIFIER SERIES ~3 MIN READ

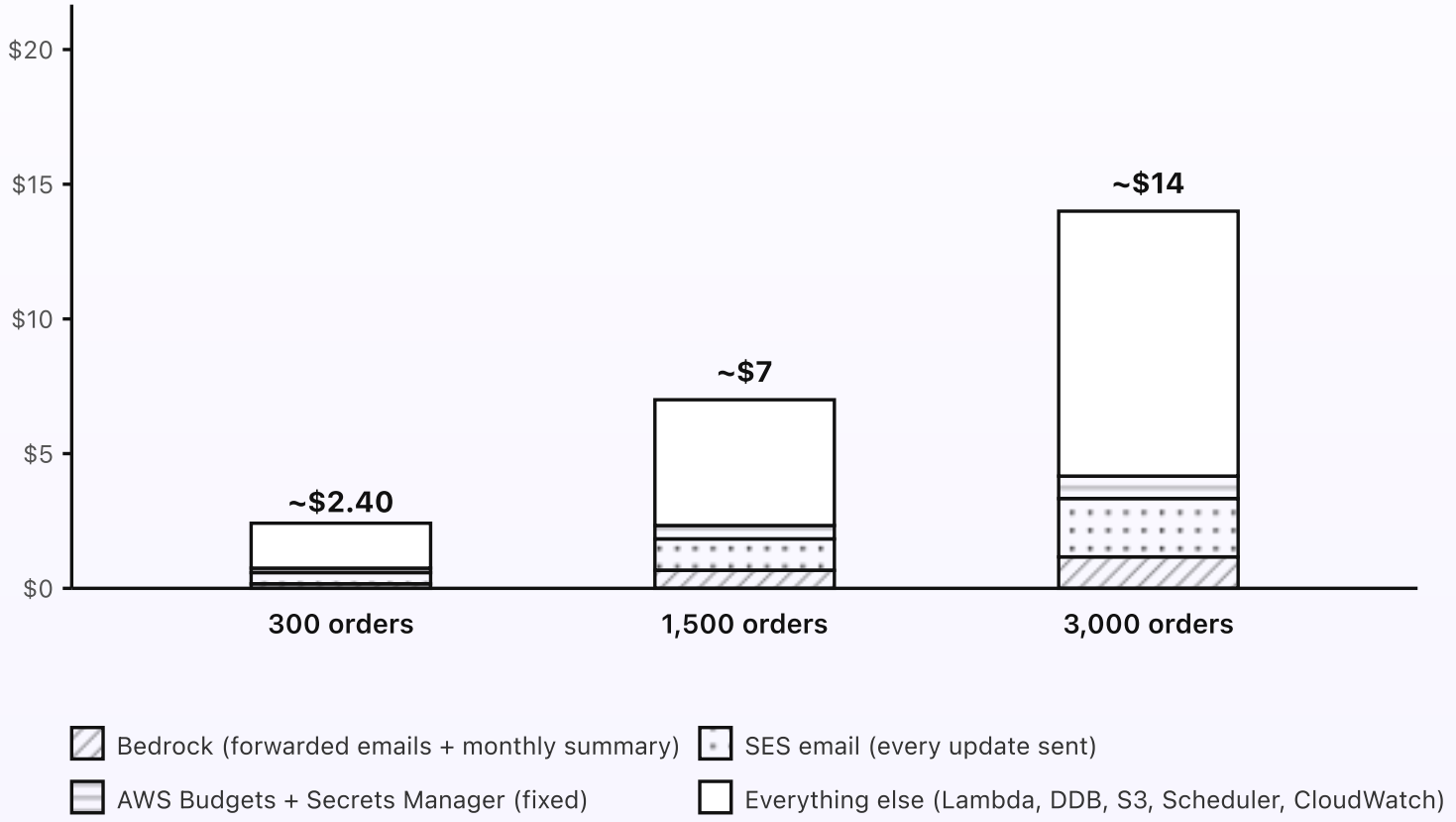
What the shipping notifier costs

The notifier is one of the cheapest systems in this whole series. The scheduled check reads a CSV from S3, does some simple comparisons, writes a few rows to DynamoDB, and sends a handful of emails. It calls no models on the check. Bedrock fires only when somebody forwards a carrier email and once a month for the summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (around 300 orders a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The scheduled check costs pennies — no model calls.
- Bedrock fires only on forwarded carrier emails (a few a month) and the monthly summary.
- At 1,500 orders the bill is around $7. At 3,000 orders it's around $14.

Cost at three volumes



The check and the update emails are the dominant cost — and even those are fractions of a cent per order.

Fig 6. Monthly cost at three order volumes. Bedrock is a small sliver because it only fires on forwarded carrier emails and the monthly summary. The dominant cost is the everything-else bucket (the scheduled check) plus the SES email for every update sent.

Where the dollars actually go

Lambda runtime (the bulk). The notifier runs every 30 minutes during the day. Each check reads the order list CSV from S3, iterates the rows, compares each order's status to what was last sent, and decides on a move. At 300 orders, that's a few hundred milliseconds per check. At 3,000 orders it's a couple of seconds. Add the sender Lambda firing for each update, the webhook Lambda handling carrier posts, the Function URL Lambda for unsubscribes, and the drive-sync Lambda every few minutes — the Lambda total still lands under a couple of dollars at all three volumes.

SES (every update sent). Outbound email is $0.10 per thousand sent. A typical order gets three to four updates over its life (shipped, out for delivery, delivered, and occasionally delayed). At 300 orders that's around a thousand emails a month — about ten cents. At 3,000 orders it's ten thousand emails, around a dollar. Inbound for the forwarding lane is the same $0.10 per thousand, and far fewer messages. Email is the second-biggest slice but still small.

DynamoDB on-demand. Three small tables: `sn-sends`, `sn-prefs`, `sn-audit`. Reads are dominant during each check (one read per order per check). Writes are sends, preference changes, and audit rows. Pennies a month at any of these volumes.

S3 + Storage. The mirrored order CSV plus the archived emails from any forwarded carrier messages. A few hundred KB total at SMB volume. Effectively free.

EventBridge Scheduler. The recurring check rule plus deferred-send rules from the quiet-hours gate. A few dozen invocations a day. Pennies.

Bedrock (only when something fires it). The scheduled check uses no Bedrock. The inbox forwarding lane fires Haiku 4.5 once per forwarded email: a few hundred input tokens (the email body) and a few output tokens (the status JSON), so a tiny fraction of a cent per read. At a few forwarded emails a month, Bedrock costs cents. The monthly summary is one larger call: write a paragraph that summarizes the month's shipments, deliveries, and delays; a couple of cents.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the carrier webhook and the unsubscribe link.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The notifier wakes up, runs for a second or two, and sleeps.
- **A Knowledge Base.** The order list is structured rows, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the check.** The scheduled decision is plain Python. Bedrock fires only on forwarded emails and the monthly summary.

How the cost scales

Lambda runtime grows roughly with order count, because every order is evaluated on every check. SES grows with order count too, since more orders mean more update emails. DynamoDB grows linearly. Bedrock is uncorrelated with order count — it only fires when somebody forwards an email or it's the first of the

month. So the bill at 6,000 orders is around $28; at 12,000 it's around $55. Past those volumes you'd probably move the check to only re-evaluate orders that aren't yet delivered, which trims the dominant cost — but that's an optimization for high-volume shops, not a redesign.

Set an AWS Budgets alarm at $20/month so anything unusual pages you before the bill matters. The notifier's normal-volume bill stays well under that ceiling.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, DynamoDB schemas, the carrier webhook flow, and EventBridge Scheduler config.

PART 7 OF 7

MAY 18, 2026 PART 7 OF 7 · SHIPPING NOTIFIER SERIES ~8 MIN READ

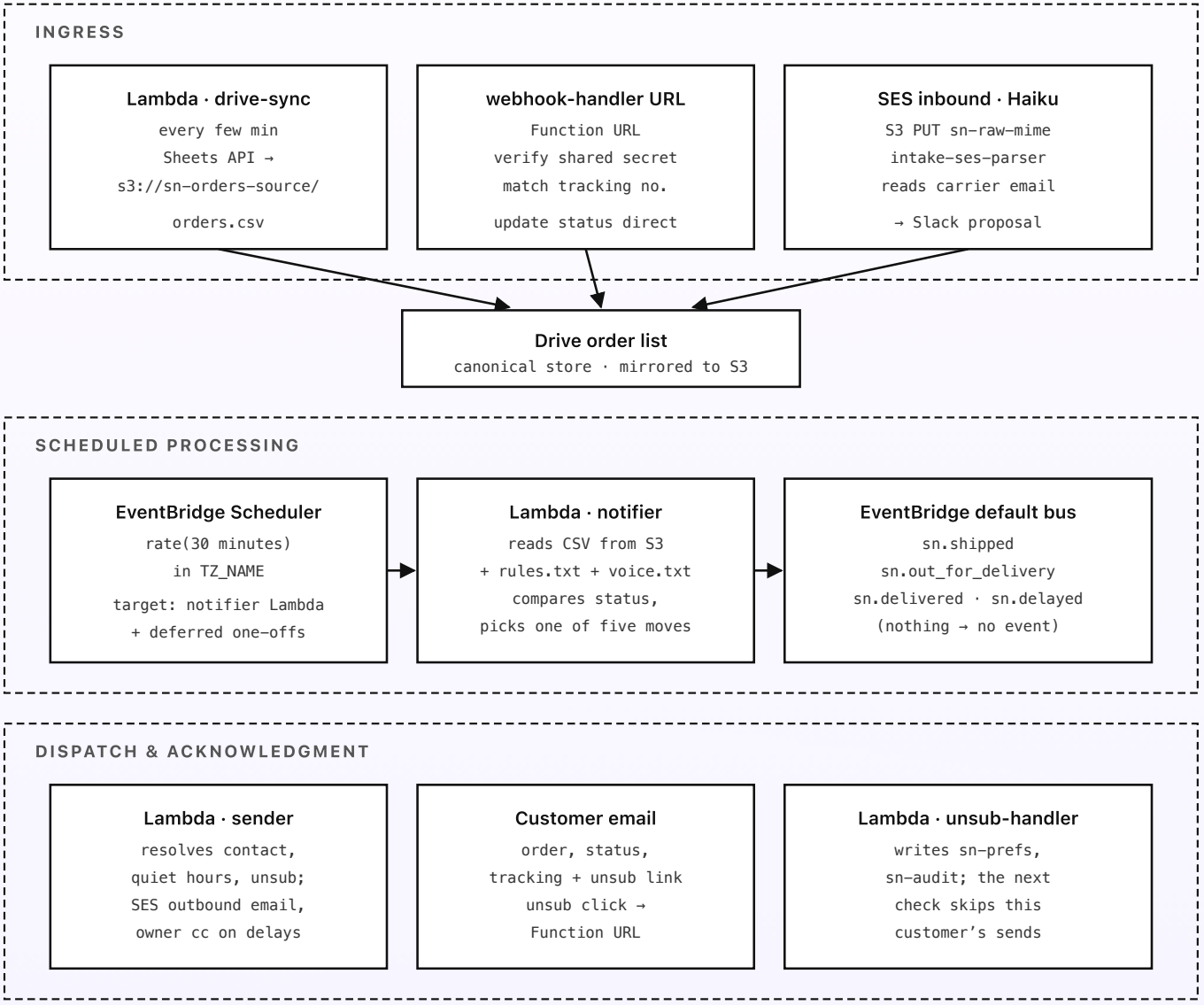
Engineering reference: the shipping notifier architecture

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

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound and outbound, Bedrock cross-Region inference, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is a customer missing a shipping update, not a regional outage. One AWS account dedicated to the notifier (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every update is sent once — and every send and delay is logged to sn-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the order list), scheduled processing (the recurring check emitting events), dispatch and acknowledgment (the update ships and the customer's unsubscribe is recorded). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

All Lambdas use the `arm64` architecture, the smallest memory size that meets latency targets (typically 256 MB), Python 3.14 runtime, and CloudWatch Logs at 7-day retention. Each function has its own least-privilege IAM role. None run inside a VPC.

- **`drive-sync`** — EventBridge Scheduler target, fires every few minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `sn/drive/sa`) to export the order list sheet as CSV and write to `s3://sn-orders-source/orders.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://sn-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`webhook-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a carrier-specific HMAC signature or shared secret on the request body (secret in Secrets Manager under `sn/carrier/secret`). Parses the carrier's tracking-update payload, matches the tracking number to an order via a GSI on the order list cache, and updates the status field in the Drive sheet via the Sheets API. Unmatched tracking numbers are written to an `sn-unmatched` list for the weekly digest. Memory: 256 MB. Timeout: 15 s.

- **`intake-ses-parser`** — S3 PUT trigger on `s3://sn-raw-mime/`. Parses MIME, extracts the text body of the forwarded carrier email, and calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to extract the tracking number and the new status. Posts the proposal to Slack via `chat.postMessage` with Approve/Edit/Discard buttons. Memory: 512 MB. Timeout: 30 s.
- **`notifier`** — EventBridge Scheduler target, every 30 minutes during the day (the schedule expression runs in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). Reads `s3://sn-orders-source/orders.csv` and the rules and voice docs. For each row, compares current status to last-sent, reads state from `sn-sends` and `sn-prefs`, decides on a move. Emits one event per row that needs an update: `sn.shipped`, `sn.out_for_delivery`, `sn.delivered`, or `sn.delayed`, with the order context as the event payload. Orders with nothing new emit nothing. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`sender`** — EventBridge rule on the four move events. Resolves contact, checks quiet hours and the unsubscribe flag, formats the update from the voice template, and ships via SES `SendRawEmail` from the verified sending identity. On a quiet-hours defer, creates a one-off EventBridge Scheduler rule that re-invokes `sender` at the start of the next sending window. On a `sn.delayed` event, adds the owner as a recipient. Writes a row to `sn-sends` after a successful send. Memory: 256 MB. Timeout: 30 s.
- **`unsub-handler`** — Lambda Function URL, public with `AuthType: NONE`; the unsubscribe link carries a signed token so only the real recipient can opt out their own order. Writes the opt-out to `sn-prefs` and an audit row to `sn-audit`. Returns a small confirmation page. Memory: 256 MB. Timeout: 15 s.

- `digest` — EventBridge Scheduler target, weekly Sunday 6pm. Reads `sn-sends` and the `sn-unmatched` list for the past week; sends a digest message to a configured Slack channel summarizing updates sent, orders in flight, and any unmatched tracking numbers. No Bedrock; the message is a plain summary table. Memory: 256 MB.
- `summary` — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `sn-sends` and `sn-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph narrative (orders shipped, delivered, average days in transit, how many ran late); emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB** · `sn-sends` — one row per update sent. PK `(order_id, status)`; attributes: `sent_date`, `sent_via` (customer/owner), `recipient`, `move` (shipped/out_for_delivery/delivered/delayed). On-demand. No TTL.
- **DynamoDB** · `sn-prefs` — one row per order's notification preference. PK `order_id`; attributes: `unsubscribed` (bool), `mute_until` (date, optional), `updated_by`. On-demand.
- **DynamoDB** · `sn-audit` — one row per write action of any kind. PK `(order_id, ts)`; attributes: `action`, `days_late` (if delay), `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB** · `sn-unmatched` — webhook tracking updates that matched no order. PK `tracking_no`; attributes: `status`, `received_at`, `raw`. On-demand. TTL 30 days.

- **S3 · `sn-orders-source`** — mirrored CSV from the Drive order list sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 3 years.
- **S3 · `sn-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3 · `sn-raw-mime`** — raw inbound MIME from forwarded carrier emails. Lifecycle to Glacier at 30 days; expiry at 1 year.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `intake-ses-parser` for reading forwarded carrier emails, and `summary` for the monthly narrative. Claude Sonnet 4.6 isn't used here — neither task needs the heavier reasoning, so Haiku 4.5 is the right cost-for-quality fit.
- **Embeddings.** Not used. The order list is structured rows; deterministic lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The notifier itself doesn't call Bedrock; the parsing lane fires a few times a month at most.

EventBridge Scheduler config

- **`sn-status-check`** — `rate(30 minutes)` with a FlexibleTimeWindow; target: `notifier` Lambda. A daytime-only window is enforced in code, not the schedule, so a late-night delivered scan still gets deferred rather than dropped.

- `sn-drive-sync` — `rate(5 minutes)`. Target: `drive-sync` Lambda.
- `sn-weekly-digest` — `cron(0 18 ? * SUN *)` in TZ. Target: `digest` Lambda.
- `sn-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `sender` when a quiet-hours defer is needed. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `tracking.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `sn-inbound-rules`: one rule with recipient `tracking@your-company.com` → spam scan → S3 PUT to `s3://sn-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the customer updates: verify a sending identity at `orders@your-company.com` with DKIM and SPF on the parent domain, and a list-unsubscribe header on every message. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **notifier role:** `s3:GetObject` on the orders, rules, and voice keys; `dynamodb:Query` + `GetItem` on `sn-sends`, `sn-prefs`; `events:PutEvents` on the default bus. *No* `bedrock:*`.

- **sender role:** `events:CreateSchedule` for the deferred-send one-offs; `ses:SendRawEmail` from the verified sending identity; `dynamodb:PutItem` on `sn-sends`; `s3:GetObject` on the voice template; `secretsmanager:GetSecretValue` only if a per-tenant sender config is used.
- **webhook-handler role:** `secretsmanager:GetSecretValue` on the carrier secret; `secretsmanager:GetSecretValue` on the Sheets-API service-account secret; outbound network access to `sheets.googleapis.com`; `dynamodb:PutItem` on `sn-unmatched`.
- **intake-ses-parser role:** `s3:GetObject` on `sn-raw-mime`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token.
- **unsub-handler role:** `dynamodb:PutItem` on `sn-prefs` and `sn-audit`; `secretsmanager:GetSecretValue` on the token-signing secret.
- **drive-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the orders and rules buckets; outbound network to `www.googleapis.com`.

Carrier webhook flow

Most carriers (and aggregators like a tracking API) can POST a tracking-update payload to a URL on each scan event. The `webhook-handler` Function URL is that URL. On each request it verifies the carrier's signature (an HMAC over the body with a shared secret, or a bearer token, depending on the carrier), parses the tracking number and the new status, normalizes the carrier's status vocabulary into the system's five stages (a small mapping table in the rules doc handles

carrier-specific status names), and updates the matching order's status in the Drive sheet via the Sheets API.

Because the webhook writes the carrier's authoritative status with no human in the loop, the only safety checks are the signature verification and the tracking-number match. A request that fails the signature is rejected with a 401 and logged. A request whose tracking number matches no order is written to `sn-unmatched` so a person can reconcile it from the weekly digest, rather than being silently dropped.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch metric for alerting.
- **Alarms:** notifier Lambda failures > 0 in an hour (the check is the one piece that has to run); sender failure rate > 1% in 24h; webhook signature-verification failures > 5/hour (might mean the carrier secret rotated).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $20/month threshold, alarm at 80% and 100%, posts to SNS topic `sn-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive and Sheets APIs live in Secrets Manager under `sn/drive/sa` (one service account with scopes for both APIs). Carrier

webhook secrets live under `sn/carrier/*`. Slack bot token lives under `sn/slack/bot-token`. The unsubscribe token-signing secret is under `sn/unsub/signing-secret`. The configured timezone, quiet-hours window, expected-delivery windows per carrier, owner contact, and grace setting all live in Parameter Store under `/sn/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions + OIDC + AWS SAM, no long-lived keys. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `sn-orders-source` and `sn-rules-source` so a bad Drive edit can be rolled back in one click, and version the EventBridge Scheduler timezone setting so you don't accidentally start running the check in UTC after a CI rotation. Total deployable surface: around eight Lambdas, four DDB tables, three S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SES rule set, and one Budgets alarm.

That's the full system. Six narrative posts and this engineering reference. If you want to talk about adapting it for your business, see Work with me.