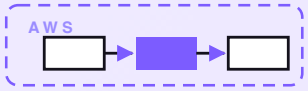
7-PART SERIES · FREE COMPANION




# Cart recovery

A serverless system that wins back online shoppers who added items but didn't check out. When a cart is left behind, it waits a sensible amount of time, then sends a friendly, well-timed reminder (and maybe a second) with what they left — and stops the instant they buy or unsubscribe. Never pushy: it respects quiet hours and sends one gentle nudge, not a barrage. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

# Cart recovery

PART 1 OF 7

MAY 23, 2026 PART 1 OF 7 · CART RECOVERY SERIES ~5 MIN READ

# A cart recovery system on AWS for a few dollars a month

A small online store loses more sales to second thoughts than to anything else. The shopper who filled a cart with three things and then got distracted by a phone call. The one who got to the shipping page and decided to “come back later.” The one who was just price-checking and never meant to buy yet but really might. None of them are lost causes — most of them simply forgot. This post walks through the design of a small system that notices a cart was left behind, waits a sensible amount of time, and sends one friendly reminder with what they left — and stops the second they buy or ask to be left alone.

KEY TAKEAWAYS

- Three sources for carts: a storefront webhook, a nightly Drive export a human can read, and a saved-link tag lane.
- Every cart ends in one of four moves on each wake-up: still shopping, first reminder, second reminder, or give up.
- Per-size waits: a small cart waits 4 hours then 24; a big cart waits 1 hour then 20.
- Reminders are friendly, respect quiet hours, carry the items and a return link, and stop on checkout.
- Designed on AWS for about $2.20/month at typical small-business volume.

## The whole system on one page

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

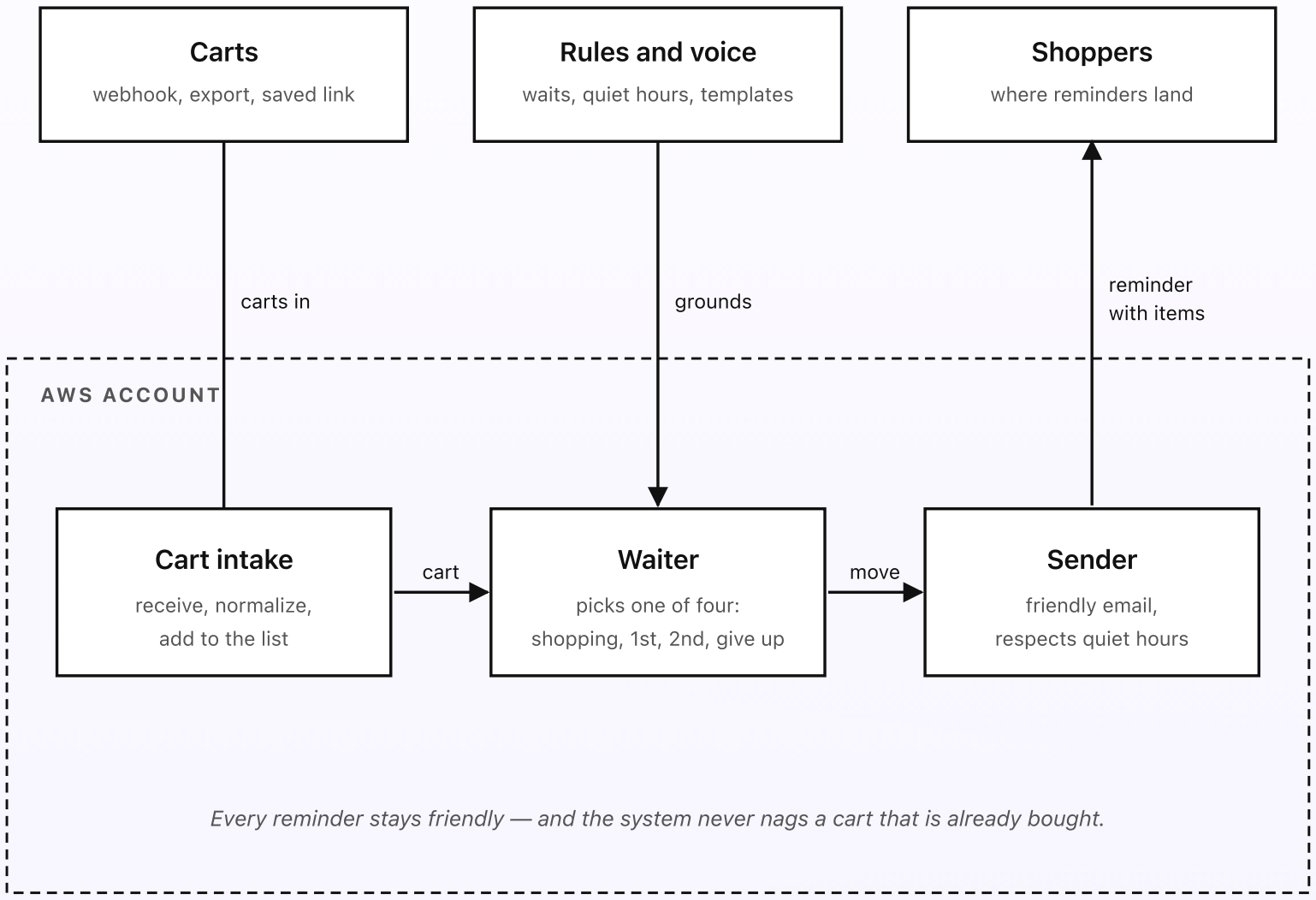

Carts
webhook, export, saved link
Rules and voice
waits, quiet hours, templates
Shoppers
where reminders land
carts in
grounds
reminder
with items
AWS ACCOUNT
Cart intake
receive, normalize,
add to the list
cart
Waiter
picks one of four:
shopping, 1st, 2nd, give up
move
Sender
friendly email,
respects quiet hours
Every reminder stays friendly — and the system never nags a cart that is already bought.

*Fig 1. Three sources outside, three pieces inside AWS. Carts flow in from a storefront webhook, a nightly Drive export, and a saved-link tag lane. The Waiter wakes once per cart and picks one of four moves. The Sender emails the right reminder to the right person at the right time.*

## What you set up once (the outside)

- **Carts.** Your storefront posts a cart event to a small webhook the moment a shopper adds an item, changes the cart, or checks out. Each event carries the cart id, the shopper's email (once they've entered it), the items, the cart total, and a timestamp. That's the main lane. Two others, covered in Part 2, fill the gaps — a nightly Drive export (the day's carts land in a sheet a human can scan) and a saved-link tag lane (carts that start from a saved link the shopper emailed themselves get marked so the reminder copy can match).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the wait for each cart size — how long to wait before the first reminder, and how long before the second. A small cart might wait 4 hours, then 24; a big cart waits 1 hour, then 20, because a bigger basket is worth a faster, gentler nudge. The doc also sets the quiet hours (no sends between 9pm and 8am local by default), the give-up point (two reminders, then stop), and the do-not-disturb rule (skip anyone who got a reminder in the last few days). The *voice* doc holds one reminder template per step — the warm first nudge and the slightly warmer second.
- **Shoppers.** The people who left a cart behind. Reminders land in their email inbox with the items they left, the cart total, a return link straight back to the filled cart, and a one-click unsubscribe. Nobody who unsubscribes ever hears from the system again.

## What runs per cart (the inside)

- **The cart intake.** Three sources feed the list. The storefront webhook is the main one — the moment a cart changes, a small Function URL Lambda writes or updates one row per cart in DynamoDB and schedules a wake-up for when the first reminder might be due. New carts also arrive via the nightly Drive export (the day's carts mirrored to a sheet for a human to read) and the saved-link tag lane (carts started from a saved link get a small flag so the copy can say "still saving these?" instead of "you left these behind").
- **The waiter.** Wakes once per cart, at the time the intake scheduled. Reads the cart. Computes time-since-abandon. Compares against the per-size wait in the rules doc. Picks one of four moves. *Still shopping*: the cart changed again or hasn't waited long enough — do nothing, reschedule. *First reminder*: crossed the first wait with no checkout — send a friendly nudge with the items and a return link. *Second reminder*: crossed the second wait, still no checkout — send one slightly warmer follow-up. *Give up*: two reminders sent and still no checkout — stop; the cart is logged and left alone. The waiter doesn't call a model to decide — the move logic is plain Python.
- **The sender.** Reads the voice doc, fills the reminder template for the chosen move, and asks Bedrock Haiku 4.5 to polish just the opening line into the store's voice (with a plain fallback if it's unavailable). Email goes through SES outbound. It honors quiet hours and the do-not-disturb list. Every send writes a row to DynamoDB so the next wake-up knows a reminder already went out. A monthly summary writes a short paragraph: carts seen, reminders sent, carts recovered, dollars won back.

## In plain words

A shopper fills a cart with $120 of goods and gets to the shipping page, then closes the tab to take a call. An hour passes (the cart is on the bigger side, so the wait is short). The system emails them: “Still thinking it over? Your cart’s still here — the linen shirt, the two candles, and the tote, $120 in all. *[return to cart]*” with a one-click unsubscribe at the foot. They don’t open it that evening. Twenty hours later, one gentle follow-up: “Last nudge — we saved your cart in case you still want it.” This time they click through and check out. The moment that checkout event lands, the system clears the chain — no third email, no nagging. If they’d unsubscribed instead, that would have stopped everything too.

The cost of running this is about $2.20 a month at SMB volume. The cost of *not* running it is every full cart that quietly evaporates because nobody ever reminded the shopper it was still there.

**DESIGN RULES THAT SHAPED EVERY DECISION**

- Every reminder ships with full context — the items, the total, a return link, an unsubscribe. The shopper never has to dig.
- Four moves, always. Still shopping, first reminder, second reminder, give up. There is no fifth.
- At most two reminders, ever. One gentle nudge, not a barrage. The give-up point is a hard stop.
- Quiet hours and the do-not-disturb list are respected. A reminder is a finite resource; bad timing burns it.
- Checkout stops the chase on the next wake-up. The system never nags a cart that is already bought.
- Every send and every stop is logged. Look back next month and you can see exactly what went out and why.

## Why this shape

Most small stores handle abandoned carts in one of three ways: a plugin that blasts everyone the same three emails on the same schedule, a vague plan to “set something up one day,” or nothing at all. The plugin works until it annoys people — three emails for a $9 cart is how you train shoppers to mark you as spam. The plan never happens. And nothing at all is how a store leaves real money on the table every single day.

The setup above keeps the cart data where your store already writes it, but adds a small system that *wakes up per cart* and acts only when a reminder is genuinely worth sending. Reminders go out after a sensible wait, sized to the cart. They are friendly by default and never escalate into pressure. They carry a return link so finishing the purchase is one click. They stop the moment the shopper buys, and the moment anyone asks to be left alone. The system is invisible on the carts that come back on their own; it only shows up for the ones that would otherwise have been forgotten.

The next four posts walk through each piece in turn: how an abandoned cart gets spotted, how a cart reminder gets timed, how a cart reminder reaches the shopper, and how a cart recovery stops on checkout. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 23, 2026 PART 2 OF 7 · CART RECOVERY SERIES ~4 MIN READ

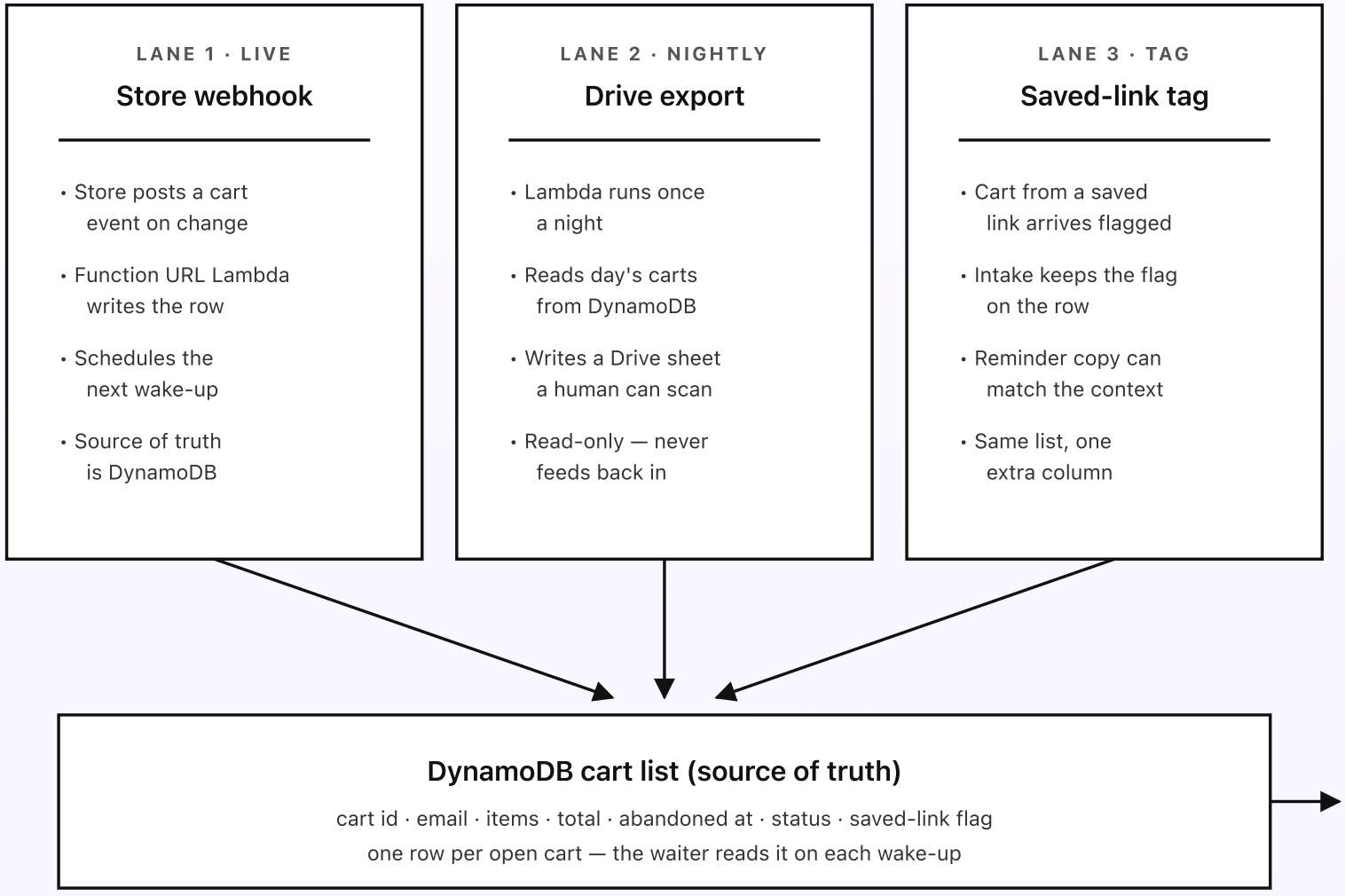
# How an abandoned cart gets spotted

The system can only win back a cart it knows about. So the first job is catching every cart the moment it forms, and noticing the instant it's left behind. There are three ways a cart gets onto the list: the store tells the system directly as it happens, the day's carts land in a sheet a human can read, and carts that started from a saved link get a small tag. The first one does almost all the work. The other two exist because real stores have gaps, and a human still likes to glance at the day.

KEY TAKEAWAYS

- Three intake lanes feed one cart list: the storefront webhook, a nightly Drive export, and a saved-link tag lane.
- The webhook is a Lambda Function URL; every add, update, and checkout posts one event.
- Each event writes or updates one row per cart in DynamoDB and schedules the next wake-up.
- The nightly Drive export mirrors the day’s carts to a sheet a human can scan.
- The DynamoDB cart list stays the canonical store. The other lanes are conveniences that write into it.

## Three lanes into one cart list




*The cart list stays the source of truth — the other lanes are conveniences that read it or tag it.*

*Fig 2. Three lanes converge on one DynamoDB cart list. The list is the source of truth; the export lane is a read-only convenience and the tag lane just adds context. The webhook does almost all the work, writing one row per cart the moment a cart changes.*

## Lane 1: the storefront webhook (the one that does the work)

The main lane. Your store calls a small endpoint — a Lambda Function URL — every time a cart changes: an item added, a quantity changed, the email entered at checkout, or the order placed. Most ecommerce platforms can post these events out of the box; if yours can't, a few lines of theme code can. Each event carries the cart id, the items, the cart total, the shopper's email (once they've typed it), and a timestamp.

The intake Lambda takes the event, writes or updates one row per cart in DynamoDB, and schedules the next wake-up via EventBridge Scheduler — one timer per cart, set to when the first reminder might be due. If the cart changes again before then, the row is updated and the timer is reset, so a shopper who's still actively adding items never gets a premature nudge. A checkout event flips the row to `bought` and cancels the timer outright. This lane is the source of truth; everything the waiter reads, it reads from here.

## Lane 2: nightly Drive export (the one a human reads)

The cart list lives in DynamoDB, which is great for the system and not great for a person who just wants to glance at the day. So a small Lambda runs once a night, reads the day's carts, and writes them to a Google Sheet in a Drive folder you control — one row per cart with the items, the total, whether it was recovered, and

when. The owner can open it over coffee and see how the day went without ever touching AWS.

The sheet is read-only as far as the system is concerned. It never feeds back in. That keeps the rule simple: the system acts on DynamoDB, and the sheet is a window onto it, not a second copy that can drift out of sync.

### Lane 3: saved-link tag

Some shoppers save a cart on purpose — they email themselves the link, or tap “save for later,” meaning to come back when payday lands. A reminder that says “you left these behind” reads slightly wrong for that shopper; “still saving these?” reads right. So when a cart arrives from a saved link, the event carries a small flag, and the intake Lambda keeps that flag on the row. It changes nothing about the timing — it just lets the copy in Part 4 match the situation.

This is the most optional of the three lanes. A store that doesn’t distinguish saved carts loses nothing; a store that does gets reminders that feel a touch more human.

## Why the cart list stays the source of truth

Three lanes in, but only one place the waiter actually looks. That’s deliberate. If the export sheet could write back, or the webhook and the tag lane each kept their own state, every “why did this reminder go out?” question would mean checking three places. Funneling everything through one DynamoDB list means there is exactly one row per cart, the timing is computed from one timestamp, and

the audit trail has one shape. The convenience lanes are first-class for catching carts and reading them, but the list is where the system lives.

Next post: how the waiter actually reads a cart, computes time-since-abandon, and picks one of four moves.

PART 3 OF 7

MAY 23, 2026 PART 3 OF 7 · CART RECOVERY SERIES ~5 MIN READ

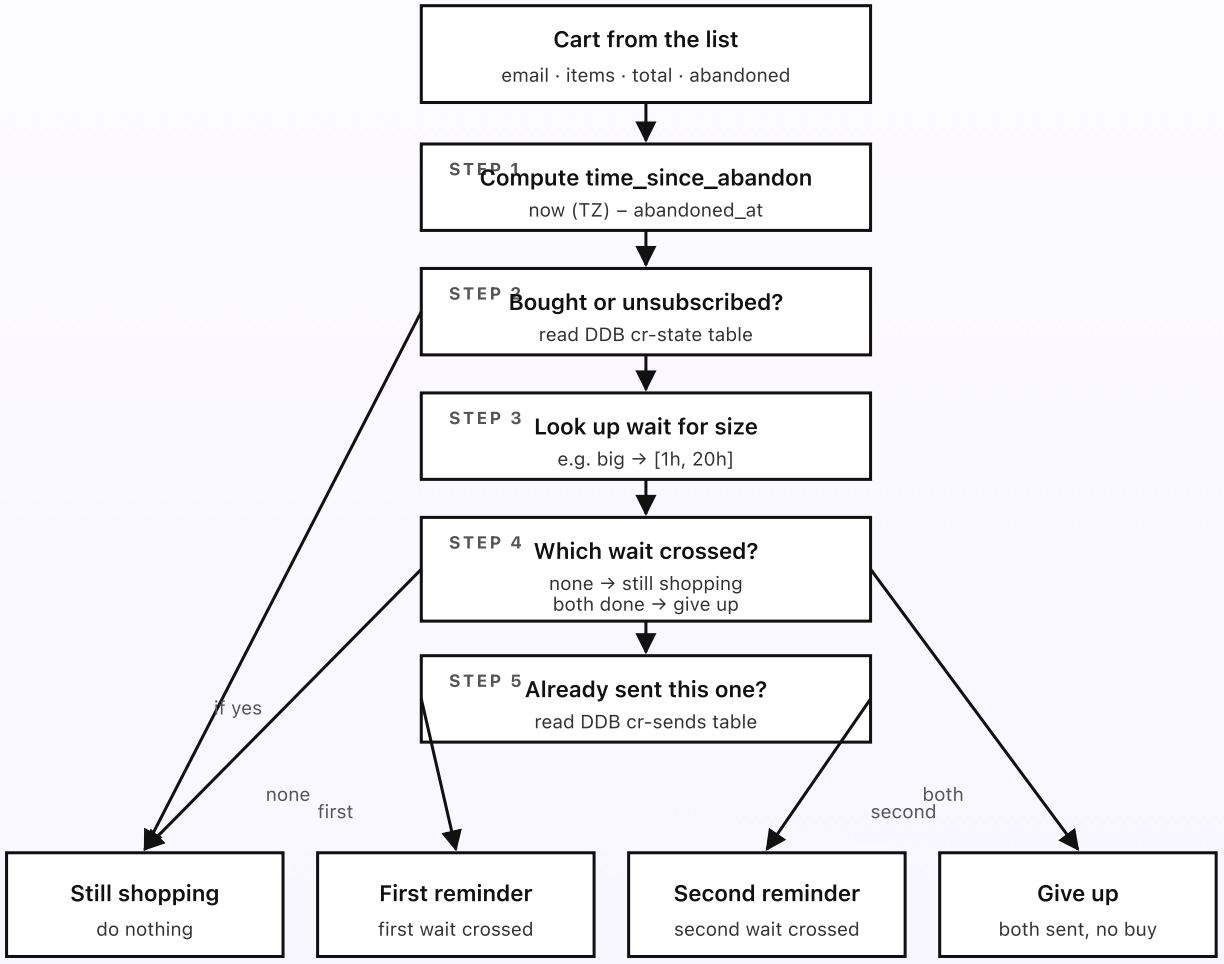
# How a cart reminder gets timed

Each cart has its own timer. When a cart is left behind, the intake schedules a wake-up for when the first reminder might be due. At that moment an EventBridge Scheduler one-off fires the waiter Lambda for just that cart. The Lambda reads the row, works out how long it's been since the cart was abandoned, and decides whether to do nothing or to send — and if so, which reminder. The whole decision is plain Python. No model. Every wait lives in the rules doc, where the owner can edit it without a deploy.

KEY TAKEAWAYS

- One EventBridge Scheduler wake-up per cart, set to when the next reminder might be due.
- Per-size waits live in the rules doc — a small cart waits 4h then 24h, a big cart waits 1h then 20h.
- Four moves per cart, every wake-up: still shopping, first reminder, second reminder, give up.
- DynamoDB tracks last-sent and bought status so reminders aren't duplicate spam.
- The waiter itself never calls a model. The decision is entirely deterministic.

## The decision flow, per cart




*The rules doc holds every wait — change one and the next wake-up uses the new value.*

*Fig 3. The waiter's decision tree, per cart, per wake-up. Five steps decide which of four moves applies. The rules doc holds every wait; the waiter only enforces them.*

## Waits: 4h/24h isn't magic, it's in the doc

The rules doc has one short line per cart size. Each names the two waits in plain prose: "Small carts (under $40): remind after 4 hours, then after 24. Medium carts: 2 hours, then 22. Big carts (over $150): 1 hour, then 20." The numbers are how long to wait after the cart was abandoned before each reminder fires. The first number is the first reminder. The second number is the last reminder — after it, the system gives up. There is no third.

The sizes exist for a reason. A small cart can wait — the shopper is in no hurry and a fast email feels pushy for a $20 basket. A big cart is worth catching while the intent is still warm, so the first nudge comes sooner. None of the waits are aggressive; even the fastest gives the shopper a full hour to come back on their own first. The point is a gentle reminder, not a pounce.

Per-cart overrides exist too. The cart row has an optional `wait_override` field. Set two values there and the waiter uses your numbers instead of the size default for that one cart. This is the right escape hatch for a launch day or a flash sale where the rhythm is different.

## Four moves, always

Every cart, every wake-up, lands in exactly one of four buckets. The names are simple on purpose.

- **Still shopping.** Not enough time has passed yet, the cart changed again, or the cart was already bought or unsubscribed. Do nothing, and reschedule the wake-up for the next wait. Most carts that come back do so here, on their own.
- **First reminder.** The first wait passed with no checkout and no reminder yet. Send a friendly nudge with the items and a return link. Write a row to the `cr-sends` DynamoDB table marking that the first reminder fired, and schedule the wake-up for the second wait.
- **Second reminder.** The second wait passed with no checkout and the first reminder already went out. Send one slightly warmer follow-up — the last one. Write the send to `cr-sends`. No further wake-up is scheduled.
- **Give up.** Both reminders went out and the cart still wasn't bought. Stop. Mark the cart `closed` in DynamoDB and leave the shopper alone. Two emails is the whole budget; a third would cross from helpful into nagging, so the system never sends it.

## State that makes the decision deterministic

The waiter reads two DynamoDB tables every wake-up. `cr-sends` records every reminder that's gone out: `(cart_id, step, sent_at, channel)`. `cr-state` records the cart's status: `(cart_id, status, bought_at, unsubscribed)`. With those two tables, the move-decision logic is a few dozen lines of Python and zero magic. A given cart with a given abandoned-at time, a given size, and a given send history always produces the same move. If a wake-up fires twice by

accident, the state in `cr-sends` shows the reminder already went out, so no duplicate is sent.

A cart that changes resets its abandoned-at time and reschedules — the shopper is active again, so the clock restarts. A checkout flips `cr-state` to `bought` and the next wake-up short-circuits to still shopping. Part 5 covers the stop-on-checkout flow in detail.

## Why the timing decision uses no model

The waiter could call a model to guess the perfect moment to send, or to decide whether a given shopper is “worth” a reminder. It doesn’t. Two reasons. First, the timing should be the one part of the system that is utterly predictable — if the doc says remind after 4 hours and there’s no checkout, the reminder fires. A model in that loop introduces variance the owner can’t reason about, and timing bugs in email are the kind that get you marked as spam. Second, model calls cost money, and most carts come back on their own, so the call would be wasted most of the time.

Bedrock fires elsewhere — to polish one line of the reminder copy in Part 4, and on the monthly summary in Part 6. Not on the timing decision. The waiter itself is plain Python that reads a clock and writes events.

Next post: how a reminder finds the right shopper, how quiet hours and the do-not-disturb list are honored, and what the email actually carries.

PART 4 OF 7

MAY 23, 2026 PART 4 OF 7 · CART RECOVERY SERIES ~5 MIN READ

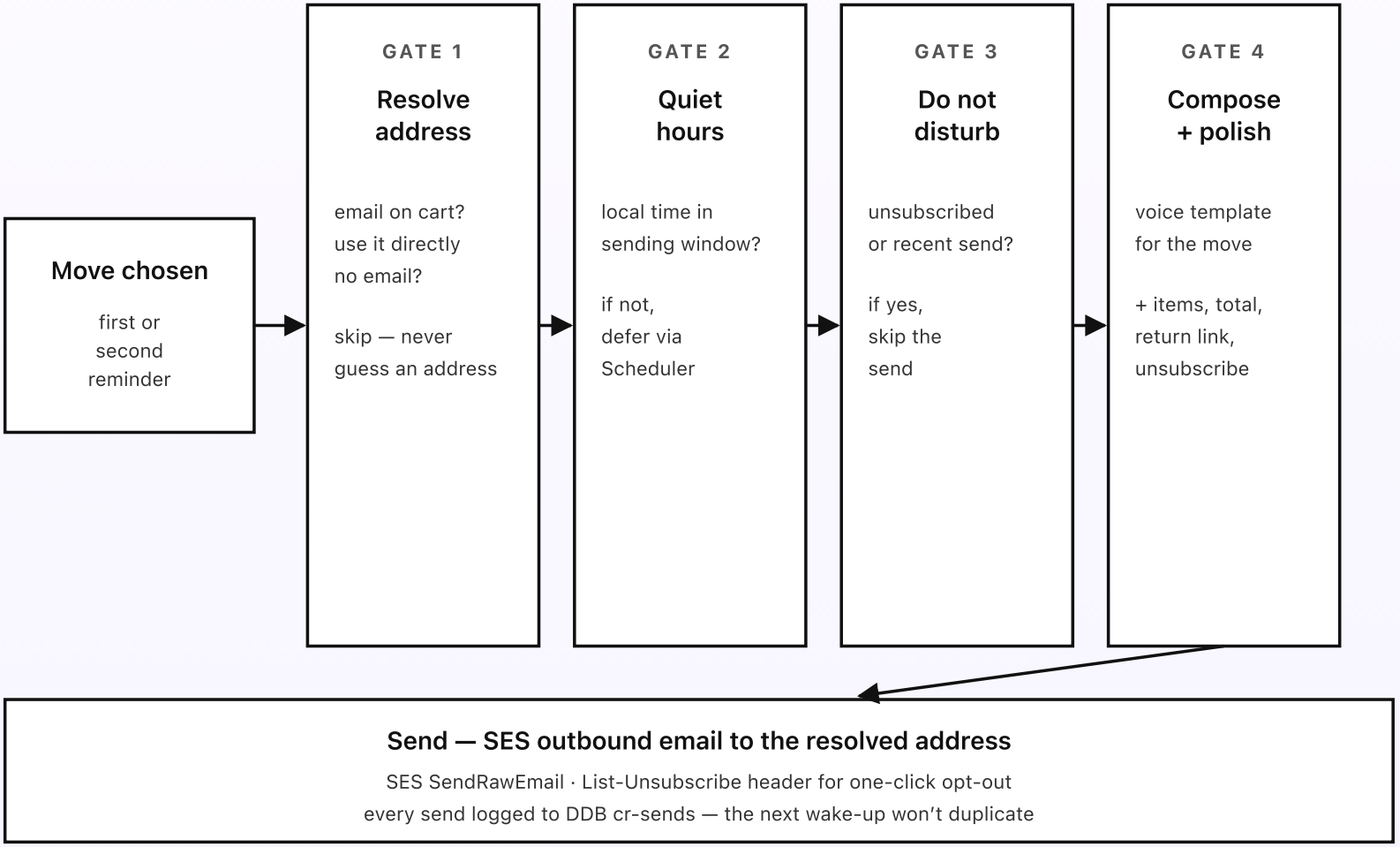
# How a cart reminder reaches the shopper

The waiter picked a move — first reminder or second reminder. Now the sender has to figure out who to email, whether now is a decent time of day, whether this shopper has had enough already, and what the message should actually say. Get any of those wrong and the reminder is worse than none at all: a 3am ping, a second email to someone who unsubscribed, a generic blast that reads like spam. Four small guardrails sit between the move and the actual email.

KEY TAKEAWAYS

- Address resolution: a cart with no email can’t be reminded — it’s skipped, not guessed.
- Quiet hours defer the send to the next decent hour instead of pinging overnight.
- A do-not-disturb check skips anyone who unsubscribed or got a reminder very recently.
- Bedrock Haiku 4.5 polishes one line into your store’s voice, with a plain fallback if it’s down.
- Every email carries the items, a return link, and a one-click unsubscribe; every send is logged.

## Four guardrails on every send




*Every gate is a deterministic check — the only AI is one polished line, with a plain fallback.*

*Fig 4. Four guardrails between the move and the sent reminder. Resolve the address. Honor quiet hours. Respect do-not-disturb. Compose with the items, polish one line, then ship via SES and log the send so the next wake-up doesn't duplicate.*

## Gate 1: resolve the address

The sender needs an email to send to, and the only honest source is the cart row itself. If the shopper entered their email at checkout before leaving, it's on the row and the sender uses it. If they never reached the email field — an anonymous cart — there's nothing to send to, and the cart is simply skipped. The system never buys, guesses, or matches an address from somewhere else. A reminder is only worth sending to someone who actually handed you their email; anything else is the kind of guessing that erodes trust fast.

Carts with an email but no name still get a warm, generic greeting ("Hi there"). The copy never pretends to know more about the shopper than the cart actually told you.

## Gate 2: quiet hours

A cart abandoned at 11pm shouldn't produce an email at 11pm, even if the wait technically elapsed. Gate 2 reads the rules doc's quiet-hours setting (default 9pm to 8am, in the shopper's timezone where known, otherwise the store's). If the current local time is inside the quiet window, the sender creates a one-off EventBridge Scheduler rule that fires at the next decent minute and exits without sending. The Scheduler re-invokes the same sender with the same payload at the deferred time, where Gate 2 will let it through.

This is also why the waits in Part 3 are deliberately not razor-thin — a small buffer means the quiet-hours defer rarely has to push a send more than a few hours, so a cart abandoned late at night still gets its reminder first thing in the morning rather than days later.

## Gate 3: do not disturb

Two things can make a send unwelcome even when the timing is fine. First, the shopper may have unsubscribed — from this cart, or from all reminders. Gate 3 checks the unsubscribe list and skips anyone on it, no exceptions. Second, the same person may have abandoned several carts in a short span; sending three reminders in two days, even for different carts, feels like a barrage. So Gate 3 also skips a send if that email got any reminder in the last few days (the window is configurable in the rules doc).

This gate is what turns “one reminder per cart” into “one gentle nudge per shopper.” The promise on the box — never pushy, one nudge not a barrage — lives mostly here.

## Gate 4: compose, polish one line, then ship

The voice doc has one email template per move: a short, warm message with placeholders for the greeting, the items, the cart total, the return link, and the unsubscribe link. The sender fills the placeholders, then makes exactly one Bedrock Haiku 4.5 call — to rewrite just the opening line into your store’s voice, given the items and whether the cart was a saved-link cart. The model gets a tight prompt and a one-sentence budget; if it’s slow or unavailable, the sender falls

back to the plain template line and ships anyway. The AI never touches the items, the total, the link, or the unsubscribe — only the tone of one sentence.

The composed email goes out via SES `SendRawEmail`, with a proper `List-Unsubscribe` header so the shopper's mail client shows a one-click opt-out at the top, and a visible unsubscribe link in the footer that hits a Function URL. Both record the opt-out the same way. A second reminder uses the slightly warmer template and gently references that the cart's still saved — never that "you ignored our last email."

Every send — first or second — writes a row to `cr-sends` in DynamoDB. The next wake-up reads that row and knows not to send the same reminder again.

## Why the guardrails exist

None of these gates are exotic. They're the kind of small care a thoughtful shopkeeper would take if they were emailing each shopper by hand — only write to someone who gave you their address, don't email at midnight, leave alone anyone who asked to be left alone or who you just wrote to, and say something warm and specific rather than a form blast. Putting them in code as four small sequential gates makes them part of the design, not something you're trusting one email template to remember.

Next post: how a cart recovery stops the moment the shopper checks out — and the three ways a human can stop it by hand.

PART 5 OF 7

MAY 23, 2026 PART 5 OF 7 · CART RECOVERY SERIES ~5 MIN READ

# How a cart recovery stops on checkout

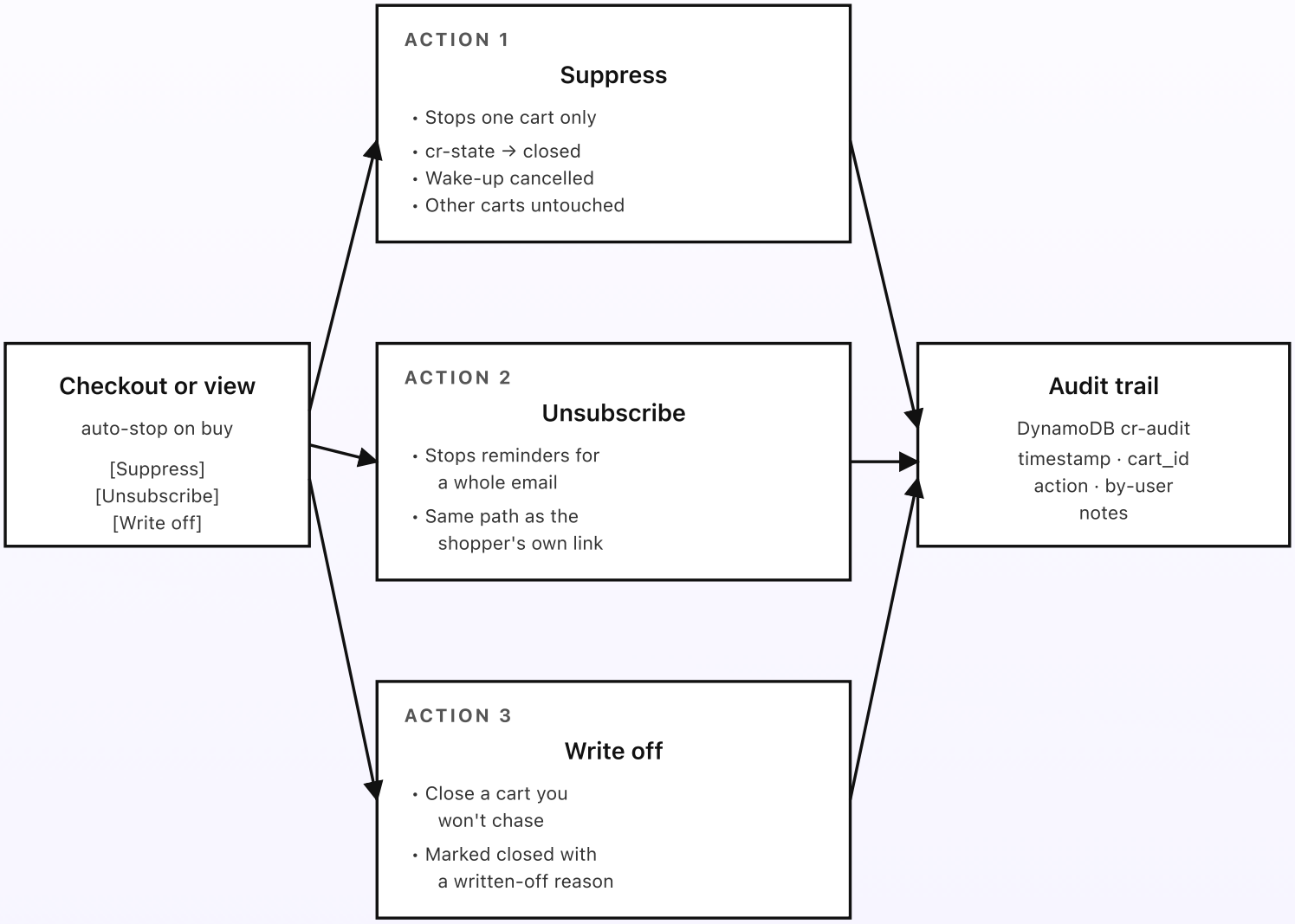
A reminder lands in a shopper's inbox at 8:03am. The linen shirt and the candles are still in the cart. They click through and buy. What happens next is the most important thing the whole system does: nothing. No third email, no "thanks for finally paying." This post walks through how the chase stops the moment a shopper checks out — and the three ways a human can stop it by hand: suppress one cart, unsubscribe an email, write it off.

---

**KEY TAKEAWAYS**

- A checkout event flips the cart to *bought* and cancels its wake-up — no reminder goes out.
- Three owner actions: *suppress* (stop one cart), *unsubscribe* (stop an email), *write off* (close it out).
- Each action updates the cart list and writes an audit row.
- The shopper's own unsubscribe link is the same path as the owner's unsubscribe action.
- Every stop is recorded so you can always see why a cart went quiet.

# The automatic stop, plus three by hand




*A checkout stops the chase automatically — the manual actions are for when a human steps in.*

*Fig 5. The automatic stop on checkout, plus three manual actions. Suppress stops one cart. Unsubscribe stops an email. Write off closes a cart the owner won't chase. Every stop, automatic or manual, writes to the audit trail.*

## The automatic stop (the one that matters most)

The whole point of the system is that it disappears the moment it's no longer needed. When a shopper checks out, your store posts a checkout event to the same webhook from Part 2. The intake Lambda finds the matching cart row, flips its status in `cr-state` to `bought`, and cancels the scheduled wake-up. If a wake-up somehow still fires — the cancel and the buy raced — the "bought or unsubscribed?" check from Part 3 catches it and routes to still shopping, so no reminder goes out either way.

This is the single most important rule in the design. A shopper who just paid you should never get an email implying they didn't. Belt and braces — the status flip and the wake-up check both guard it — because the cost of getting it wrong is a customer who feels nagged the instant after they bought.

## Action 1: suppress (stop one cart)

Sometimes a cart shouldn't be chased even though it's technically open. The shopper called and said they'll pay by invoice. The order's being handled offline. The cart is a test the owner placed themselves. The owner opens the day's cart sheet, finds the row, and taps *Suppress*.

Suppress submits to a Function URL Lambda. Two things happen: the cart's status in `cr-state` flips to `closed`, and its scheduled wake-up is cancelled. Crucially, suppress is scoped to one cart — the same shopper's other carts, now or later, are untouched. It's the narrowest stop available, for the case where this one cart is the exception.

## Action 2: unsubscribe (stop an email)

When a shopper clicks the unsubscribe link in a reminder, that's the same action as the owner choosing *Unsubscribe* on a row — both add the email to the unsubscribe list. The do-not-disturb gate from Part 4 reads that list on every single send, so once an address is on it, no reminder ever reaches it again, for any cart, now or in the future.

The shopper's one-click link and the owner's button hitting the same path is deliberate. There is exactly one unsubscribe list and one way onto it, so "did this person opt out?" always has a single, honest answer. An unsubscribe is forever unless the shopper themselves opts back in; the system never quietly re-adds anyone.

## Action 3: write off (close it out)

Some carts are simply not worth chasing. A $4 cart of out-of-stock items. A cart from an address that's bounced before. A duplicate. *Write off* closes the cart out: the Lambda marks it `closed` in `cr-state` with a written-off reason, cancels any wake-up, and that's the end of it. Unlike suppress ("don't chase this, but it's a real cart"), write-off is the owner saying "this cart was never going to convert." The

distinction matters for the monthly summary in Part 6, which counts written-off carts separately so the recovery rate isn't dragged down by carts nobody ever meant to chase.

## Every stop is logged, every stop is clear

The `cr-audit` table records every stop — the automatic one on checkout and all three manual ones — with the user (or "system" for the automatic stop), the timestamp, and a snapshot of the cart row before and after. Two months from now, when someone asks "why did this cart never get a second reminder?", the answer is one lookup away: bought on the 14th, or written off by Sam on the 15th, or unsubscribed by the shopper at 9:02am. No mystery, no guessing.

This kind of clarity is what lets the owner trust the system to run unattended. The reminders go out on their own, the stops happen on their own, and when a human does want to know what happened, the trail is right there.

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.

PART 6 OF 7

MAY 23, 2026 PART 6 OF 7 · CART RECOVERY SERIES ~3 MIN READ

# What the cart recovery costs

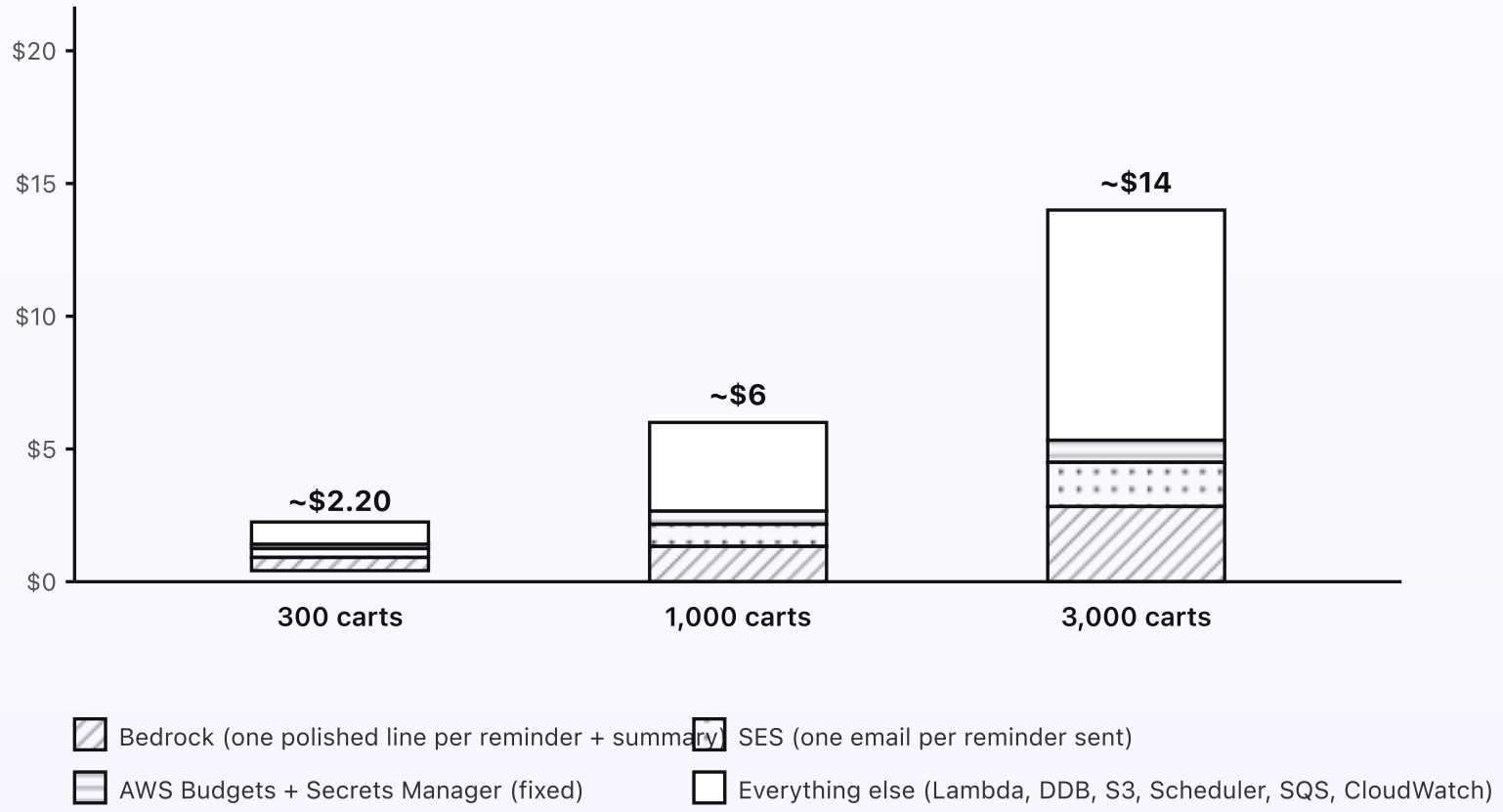
This is one of the cheapest systems in the whole series. Each wake-up reads one cart row, does a little time arithmetic, writes a couple of rows to DynamoDB, and sometimes sends one email. It calls no model to decide. Bedrock fires only to polish one line of a reminder and once a month for the recovery summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

---

**KEY TAKEAWAYS**

- Around $2.20/month at typical SMB volume (around 300 abandoned carts a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The per-cart wake-up costs fractions of a cent — no model calls.
- Bedrock fires only to polish one reminder line and on the monthly summary.
- At 1,000 carts the bill is around $6. At 3,000 carts it's around $14.

## Cost at three volumes




*The per-cart wake-up is the dominant cost — and even that is fractions of a cent per cart.*

*Fig 6. Monthly cost at three abandoned-cart volumes. Bedrock and SES are small slivers because they only fire on a reminder that actually sends. The dominant cost is the everything-else bucket: the per-cart wake-ups and the webhook handling every cart event.*

## Where the dollars actually go

**Lambda runtime (the bulk).** The webhook Lambda fires on every cart event (an active shopper can produce a dozen), and the waiter wakes once or twice per abandoned cart. Each invocation is a few hundred milliseconds of date arithmetic and a couple of DynamoDB calls. Add the nightly export Lambda and the unsubscribe handler, and the Lambda total still lands well under a dollar at all three volumes. Most cart events come from active shoppers and never lead to a reminder at all, so the work is cheap.

**DynamoDB on-demand.** Three small tables: `cr-state`, `cr-sends`, `cr-audit`. Writes happen on each cart event and each send; reads happen on each wake-up. Pennies a month at any of these volumes.

**S3 + Storage.** The nightly export and a little config. A few hundred KB total at SMB volume. Effectively free.

**EventBridge Scheduler.** One wake-up per abandoned cart, plus the occasional quiet-hours defer. A few hundred to a few thousand invocations a month. Pennies.

**SES.** One email per reminder actually sent. At 300 abandoned carts a month, only a fraction get a reminder (most come back on their own), so a few hundred emails at $0.10 per thousand — a few cents. Even at 3,000 carts it's well under a dollar.

**Bedrock (only when a reminder sends).** The timing decision uses no Bedrock. Each sent reminder fires one Haiku 4.5 call to polish a single line: a couple of hundred input tokens and a few dozen output, so a tiny fraction of a cent per reminder. The monthly summary is one larger call — write a paragraph on carts seen, recovered, and dollars won back — a couple of cents. With the deterministic

fallback, even a Bedrock outage costs nothing extra; the reminder just ships with the plain line.

**SQS.** A small queue with a dead-letter queue sits between the webhook and the intake so a burst of cart events never drops anything. Requests are pennies at this volume; the DLQ is there for safety, not cost.

## What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the 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 system only runs when a cart event or a wake-up arrives.
- **A Knowledge Base.** Carts are structured rows, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the decision.** The timing decision is plain Python. Bedrock fires only to polish one line and on the monthly summary.

## How the cost scales

Lambda runtime grows roughly linearly with cart events, because every event is handled and every abandoned cart gets a wake-up or two. DynamoDB grows linearly too. Bedrock and SES grow only with reminders actually sent, which is a fraction of carts. So the bill at 5,000 abandoned carts is around $22; at 10,000 it's

around $40. Past those volumes you'd batch the wake-ups (group carts due in the same window into one Lambda run) to shave the per-cart overhead, but that's an optimization for a busy store — not a redesign.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The system'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 SQS and DLQ wiring, and EventBridge Scheduler config.

PART 7 OF 7

MAY 23, 2026 PART 7 OF 7 · CART RECOVERY SERIES ~8 MIN READ

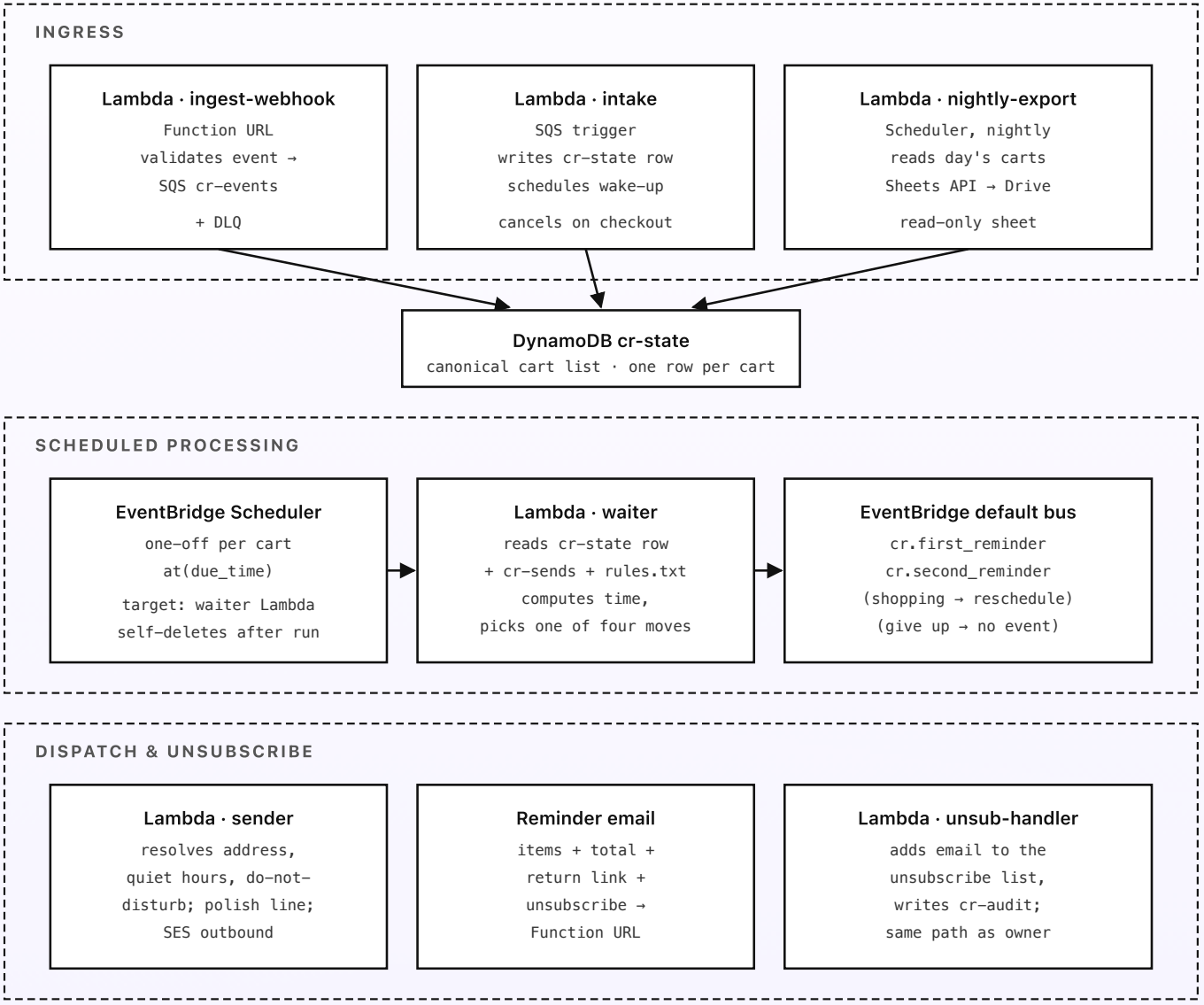
# Engineering reference: the cart recovery architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SQS and DLQ wiring, EventBridge Scheduler config, the DynamoDB schemas, and the unsubscribe flow. Read alongside the previous six posts; this one's the build sheet.

## Region and account shape

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

## Topology




*A checkout stops the chase automatically — and every interaction is logged to cr-audit.*

*Fig 7. AWS topology, in three regions of the diagram: ingress (the webhook through a queue into the cart list), scheduled processing (the per-cart wake-up emitting events), dispatch and unsubscribe (the reminder ships and the opt-out 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.

- **`ingest-webhook`** — Lambda Function URL, public with `AuthType: NONE`; verifies a shared-secret HMAC header from the storefront on the raw body. Validates the cart event shape and pushes it to the SQS queue `cr-events`, then returns 200 fast so the storefront isn't blocked. Decoupling via SQS means a burst of cart traffic (a flash sale) never overruns the intake and nothing is dropped — failures land in the DLQ. Memory: 256 MB. Timeout: 10 s.
- **`intake`** — SQS trigger on `cr-events`, batch size up to 10. For each event: upserts the cart row in `cr-state` keyed by `cart_id`; on an add/update, (re)schedules the per-cart wake-up via an EventBridge Scheduler one-off at the first wait; on a checkout event, flips status to `bought` and deletes the pending schedule; on a saved-link event, sets the `saved_link` flag. Idempotent on `(cart_id, event_id)` so an SQS redelivery is a no-op. Memory: 256 MB. Timeout: 30 s.

- **`nightly-export`** — EventBridge Scheduler target, once a night. Scans `cr-state` for the day's carts and writes them to a Google Sheet via the Sheets API (service-account credentials in Secrets Manager under `cr/drive/sa`). The sheet is read-only to the system; it never feeds back. Memory: 256 MB. Timeout: 60 s.
- **`waiter`** — EventBridge Scheduler one-off target, fired per cart at the due time. Reads the `cr-state` row and `cr-sends` history, loads `s3://cr-rules-source/rules.txt`, computes `time_since_abandon`, and decides on a move: `still_shopping` (reschedule for the next wait), `first_reminder` / `second_reminder` (emit the matching event), or `give_up` (mark `closed`, emit nothing). Emits to the EventBridge default bus. Memory: 256 MB. Timeout: 30 s. *No Bedrock calls.*
- **`sender`** — EventBridge rule on the two reminder events. Resolves the address, checks quiet hours and the do-not-disturb list, formats the email from the voice template, and makes one Bedrock Haiku 4.5 call to polish the opening line (with a deterministic fallback on timeout or error). Ships via SES `SendRawEmail` with a `List-Unsubscribe` + `List-Unsubscribe-Post` header for one-click opt-out. On a quiet-hours defer, creates a one-off Scheduler rule that re-invokes `sender` at the next sending minute. Writes a row to `cr-sends` after a successful send. Memory: 512 MB. Timeout: 30 s.
- **`unsub-handler`** — Lambda Function URL, public with `AuthType: NONE`; serves the unsubscribe link from the email and the `List-Unsubscribe-Post` one-click. Adds the email to the unsubscribe list (a `cr-state` partition or a small `cr-unsub` table) and writes `cr-audit`. The owner's suppress/unsubscribe/write-off actions from the export sheet hit the same handler with a signed admin token. Memory: 256 MB. Timeout: 15 s.

- `summary` — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `cr-sends`, `cr-state`, and `cr-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph recovery narrative (carts seen, reminders sent, recovered, dollars won back, written off); emails it via SES to the configured stakeholder list. Memory: 512 MB.

## Storage

- **DynamoDB · `cr-state`** — one row per cart, the canonical list. PK `cart_id`; attributes: `email`, `items`, `total`, `abandoned_at`, `status` (open/bought/closed), `saved_link`, `wait_override`, `schedule_name`. On-demand. A GSI on `email` supports the do-not-disturb and unsubscribe lookups.
- **DynamoDB · `cr-sends`** — one row per reminder sent. PK `(cart_id, step)`; attributes: `sent_at`, `channel`, `move` (first_reminder/second_reminder), `recipient`. On-demand. A GSI on `recipient` backs the "recent send to this email" do-not-disturb check.
- **DynamoDB · `cr-audit`** — one row per write action of any kind, including the automatic stop on checkout. PK `(cart_id, ts)`; attributes: `action`, `by_user` (or `system`), `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB · `cr-unsub`** — the unsubscribe list. PK `email`; attributes: `unsubscribed_at`, `source` (shopper/owner). On-demand. No TTL.
- **S3 · `cr-rules-source`** — the rules and voice docs mirrored from Drive as plain text. Versioning enabled, so a bad edit rolls back in one click.

- **SQS · `cr-events`** — the cart-event buffer between the webhook and the intake. Standard queue; visibility timeout sized to the intake timeout; redrive to `cr-events-dlq` after 5 attempts.
- **SQS · `cr-events-dlq`** — dead-letter queue for events the intake couldn't process. A CloudWatch alarm on queue depth > 0 pages the admin.

## 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: `sender` for the one-line reminder polish (with a deterministic fallback), and `summary` for the monthly recovery narrative. Heavier reasoning isn't needed anywhere, so Sonnet 4.6 isn't wired in — Haiku 4.5 covers both paths.
- **Embeddings.** Not used. Carts are 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 timing decision doesn't call Bedrock; the polish fires only on a reminder that actually sends.

## EventBridge Scheduler config

- **Per-cart wake-ups** — created on the fly by `intake` with `at(YYYY-MM-DDTHH:MM:SS)` expressions in `TZ_NAME`, target `waiter`, with `--action-after-completion DELETE` so each rule self-cleans. `waiter` reschedules for the second wait when it sends the first reminder.

- `cr-nightly-export` — `cron(0 2 * * ? *)` in TZ. Target: `nightly-export` Lambda.
- `cr-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **Quiet-hours defers** — created on the fly by `sender` when a send falls in the quiet window. Use `at(...)` with `--action-after-completion DELETE`.

## SES outbound and the webhook

- Verify a sender identity at `shop@your-store.com` with DKIM and SPF on the parent domain; out of sandbox by request. A custom `MAIL FROM` subdomain keeps alignment clean for deliverability.
- SES configuration set `cr-sends-config`: event destination to CloudWatch for bounces and complaints; a complaint rate over threshold auto-adds the address to `cr-unsub`.
- The storefront webhook posts to the `ingest-webhook` Function URL. The shared secret used for the HMAC header lives in Secrets Manager under `cr/webhook/secret`; rotate it without redeploying the storefront by accepting two valid secrets during a rotation window.

## IAM (least privilege per Lambda)

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

- **ingest-webhook role:** `sqs:SendMessage` on `cr-events`; `secretsmanager:GetSecretValue` on the webhook secret. Nothing else.

- **intake role:** `sqs:ReceiveMessage` + `DeleteMessage` on `cr-events`; `dynamodb:PutItem` + `UpdateItem` on `cr-state`; `scheduler:CreateSchedule` + `DeleteSchedule` for the per-cart wake-ups; `iam:PassRole` on the Scheduler target role.
- **waiter role:** `s3:GetObject` on the rules and voice keys; `dynamodb:GetItem` + `Query` on `cr-state`, `cr-sends`; `scheduler:CreateSchedule` for the second-wait reschedule; `events:PutEvents` on the default bus. *No* `bedrock:*`.
- **sender role:** `scheduler:CreateSchedule` for quiet-hours defers; `secretsmanager:GetSecretValue` on no secret beyond config; `bedrock:InvokeModel` on the Haiku ARN; `ses:SendRawEmail` from the verified sender; `dynamodb:PutItem` on `cr-sends`; `dynamodb:Query` on `cr-unsub` and the `cr-sends` recipient GSI.
- **unsub-handler role:** `dynamodb:PutItem` on `cr-unsub` and `cr-audit`; `dynamodb:UpdateItem` on `cr-state` for suppress/write-off; `scheduler:DeleteSchedule` to cancel a wake-up on suppress.
- **nightly-export and summary roles:** `dynamodb:Scan` / `Query` on the relevant tables; `secretsmanager:GetSecretValue` on the Google service-account secret; `ses:SendRawEmail` (summary only); `bedrock:InvokeModel` on the Haiku ARN (summary only); outbound network to `www.googleapis.com`.

## Unsubscribe and stop flow

There is exactly one unsubscribe list (`cr-unsub`) and one way onto it: the `unsub-handler`. The shopper's one-click link, the RFC 8058 `List-Unsubscribe-Post` request their mail client sends, and the owner's *Unsubscribe* action from the export sheet all hit the same handler. Suppress and write-off go through the same

handler too, but they target `cr-state` (one cart) rather than `cr-unsub` (an email). The automatic stop on checkout is handled in `intake` — status flip plus schedule delete — and double-guarded by the bought-or-unsubscribed check in `waiter`, so a racing wake-up never sends to someone who just bought.

## 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:** `cr-events-dlq` depth > 0 (a cart event failed to process); sender failure rate > 1% in 24h; SES complaint rate over the safe threshold; waiter errors > 0 in an hour.
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `cr-cost-alarm` subscribed to the on-call admin's email.

## Config and secrets

Service-account credentials for the Drive and Sheets APIs live in Secrets Manager under `cr/drive/sa`. The storefront webhook secret is `cr/webhook/secret`. The configured timezone, quiet-hours window, do-not-disturb window, and the size-to-wait map all live in Parameter Store under `/cr/config/`, with the human-editable copy mirrored from the Drive rules doc to `s3://cr-rules-source/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

## Deploy

GitHub Actions with OIDC into a deploy role — no long-lived keys — running AWS SAM. The opinionated bits: turn on S3 versioning for `cr-rules-source` so a bad Drive edit rolls back in one click, give the SQS queue a generous redrive policy so a transient intake error retries before the DLQ, and version the EventBridge Scheduler timezone setting so you don't accidentally start firing wake-ups in UTC after a CI rotation. Total deployable surface: around seven Lambdas, four DDB tables, one S3 bucket, two SQS queues, one EventBridge rule on the default bus (plus the per-cart Scheduler one-offs), one SES configuration 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 store, see Work with me.