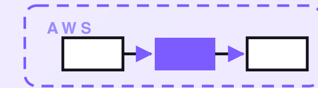
7-PART SERIES · FREE COMPANION



Back-in-stock notifier

A shopper lands on a sold-out product, taps “notify me”, and joins a queue no one usually manages well — so the stock quietly comes back, a few people who happened to check that morning grab it, and everyone who actually asked to be told never hears a thing. This is the design of a small serverless system that closes that gap: it captures each notify-me request against the exact SKU, watches the store’s inventory feed for that item coming back above a threshold, and then texts or emails the waiting list *in the order they joined* — each message carrying a short-lived reserve link so the first people to respond actually get the stock. It notifies each person once per restock, honours opt-out, caps the batch to the quantity that came in so it never oversells, and expires stale requests and unclaimed holds so the line keeps moving. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

Workflow guide $19 · Deployable AWS CDK starter $79 · Bundle $89

Free lite starter + this PDF · paid tiers at
shop.allanninal.dev/w/back-in-stock-notifier

CONTENTS

Back-in-stock notifier

01 A back-in-stock notifier on AWS for a few dollars a month
02 How a back-in-stock request gets captured
03 How a restock gets detected
04 How the waiting list gets notified
05 How a reservation gets held
06 What the back-in-stock notifier costs
07 Engineering reference: the back-in-stock notifier architecture

PART 1 OF 7

JULY 2, 2026 PART 1 OF 7 · BACK-IN-STOCK NOTIFIER SERIES ~10 MIN READ

A back-in-stock notifier on AWS for a few dollars a month

A sold-out product is a small, silent leak: the customer who wanted it taps away, the stock trickles back a fortnight later, and whoever happens to be looking that morning gets it while everyone who actually asked to be told hears nothing. This post walks through the design of a small serverless system that turns “notify me” into a fair, orderly queue — first come, first told, with a short window to buy — and never promises more than it can deliver.

KEY TAKEAWAYS

- A shopper taps “notify me” on a sold-out item; the request is stored against that exact SKU and joins a per-item queue.
- When the store’s inventory reports that SKU back above a threshold, the waiting list is texted or emailed in the order they joined.
- Each notice carries a short-lived reserve link, so the first people to respond actually lock the stock before it sells out again.
- It notifies once per restock, honours opt-out, and caps the batch to the units available so it never oversells.
- Designed on AWS for about $2.10/month at roughly 120 requests a month. One Bedrock call phrases each notice; everything else is plain Python.

The whole system on one page

Before any code, here’s the shape of what we’re designing. Every shop that sells popular things sells out of them, and a sold-out product page is a quiet leak: the customer who wanted it taps away, the stock trickles back a fortnight later, and whoever happens to be looking that morning gets it while everyone who actually asked to be told hears nothing. The system below catches that moment on both ends: it remembers who asked, for what, and in what order — and when the item comes back, it tells them, oldest first, with a short window to buy.

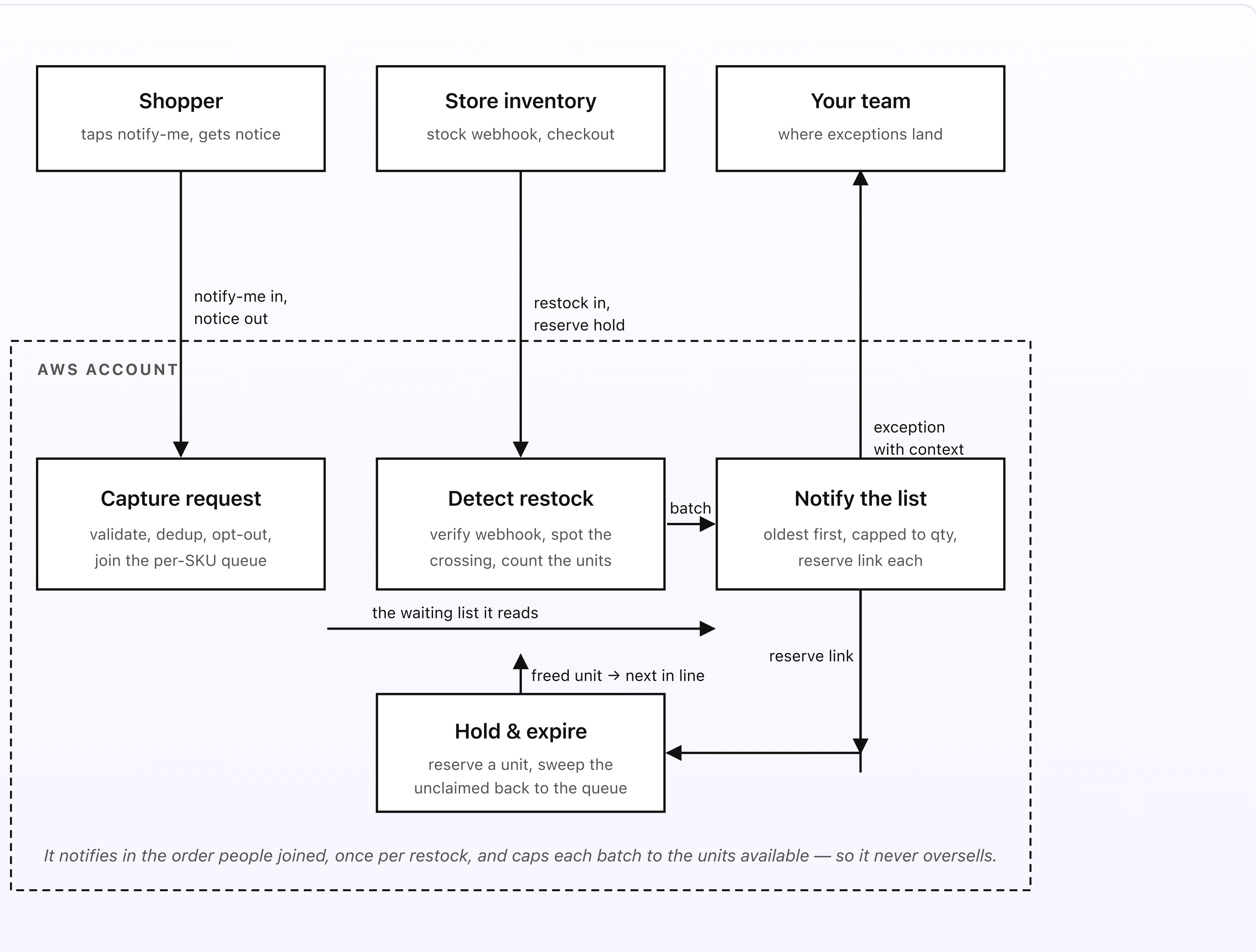

Shopper
taps notify-me, gets notice
Store inventory
stock webhook, checkout
Your team
where exceptions land
notify-me in,
notice out
restock in,
reserve hold
AWS ACCOUNT
exception
with context
Capture request
validate, dedup, opt-out,
join the per-SKU queue
Detect restock
verify webhook, spot the
crossing, count the units
batch
Notify the list
oldest first, capped to qty,
reserve link each
the waiting list it reads
reserve link
freed unit → next in line
Hold & expire
reserve a unit, sweep the
unclaimed back to the queue
It notifies in the order people joined, once per restock, and caps each batch to the units available — so it never oversells.

Fig 1. Two things outside, four pieces inside AWS. A shopper joins the queue through Capture; when Store inventory fires a restock, Detect counts the units and Notify messages that many people oldest-first with a reserve link each; Hold locks the stock and sweeps any unclaimed unit back to the next person in line.

What you set up once (the outside)

- **The store inventory and storefront.** Whatever platform already runs the shop — it needs to do two things: fire a webhook when a SKU's stock level changes (most e-commerce platforms emit an inventory or product-update event), and hold the checkout the reserve link points to. You point the platform's webhook at one AWS URL and store its signing key in Secrets Manager. This is the trigger for every notice; the catalogue supplies the product names, and the checkout is where a reserved unit is actually bought. It's covered in Part 3.
- **A "notify me" control and a small settings doc.** A button on your sold-out product pages that posts the shopper's contact and the SKU to a second AWS URL — nothing more than a name, a phone or email, and the product id. Alongside it sits a small settings doc: the store voice for the notices, the restock threshold per item (how many units in stock counts as "back"), how long a reserve link stays live, and the escalation rules. You already know these numbers; this just puts them where the system can read them.
- **Your team.** The person who picks up anything the system deliberately won't decide on its own — a restock that looks wrong (a thousand units appearing at 3am), a SKU with no matching catalogue entry, a bounce that won't deliver. They get a message with the item, the queue length, and what happened. The system opens the door for the customer; a human handles the odd cases behind it.

What runs on every restock (the inside)

- **Capture request.** The notify-me button posts to one Lambda Function URL. The function validates the request, keeps exactly one active request per person per item (a second tap doesn't create a second place in the queue), checks the opt-out list, and appends the request to the per-SKU waiting list stamped with the moment it joined. This is Part 2.
- **Detect restock.** The store posts a stock-level webhook to a second Function URL. The function verifies the signature, decides whether this is a genuine crossing from below the threshold to above it (rather than a level wobbling around the line), works out how many units came in, and fires exactly one notify batch for that restock. This is Part 3.
- **Notify the list.** A worker reads the per-SKU queue oldest-first, caps the batch to the units available so it can never invite more people than there is stock, phrases one notice in your voice with a single Bedrock call, and sends each person their own short-lived reserve link — in the order they joined. This is Part 4.
- **Hold and expire reservation.** A reserve-link click holds a single unit with a write that can never take the count below zero, then hands the shopper to checkout. A scheduled sweep expires holds that aren't used inside the window and requests that have gone stale, returning any freed unit to the next person waiting. This is Part 5.

In plain words

Loom & Last is a small independent homeware-and-footwear shop in Frome. Their walnut dining chair sells out constantly. One Tuesday, Della lands on the sold-out

chair page and taps “notify me”; the system records her request against that SKU and puts her at position seven in the queue. Over the next two weeks another dozen people do the same. A fortnight later a delivery lands and the shop’s inventory posts the chair back in at four units. The system sees the crossing, works out there are four to sell, and texts the first four people in the queue — Della is one of them: “Good news — the walnut dining chair is back at Loom & Last. You’re near the front of the list. Here’s a link that holds one for you for the next 15 minutes: loomandlast.co.uk/r/….” Della taps it, the chair is held, she checks out. Nobody at the shop touched a thing.

The fifth person in the queue, Marcus, hears nothing yet — there were only four chairs, and the system will not invite a fifth person for four units. But one of the four never taps their link. Fifteen minutes later the expiry sweep notices that hold went unused, returns that chair to the pool, and texts Marcus: he’s next in line, and now there’s one for him. The queue moved by exactly one, in order, and at no point were there five live reserve links for four chairs. One restock, four units, told fairest-first, and never oversold.

DESIGN RULES THAT SHAPED EVERY DECISION

- First come, first told. People are notified in the exact order they joined the queue — oldest request first, no jumping.
- Never more invites than units. Each restock's batch is capped to the quantity that came in, so the system can't oversell.
- A notice, then a hold. The reserve link locks a single unit for a short window; the first to respond genuinely get the stock.
- Once per restock. Each person is told at most once per restock event; a flapping stock level doesn't spam anyone.
- The model only writes words. The queue order, the cap, the dedup, and the holds are deterministic; Bedrock just phrases the notice.
- The line keeps moving. Unclaimed holds and stale requests expire and pass to the next person, and opt-out is honoured for good.

Why this shape

Most shops handle back-in-stock one of three ways: they don't — the item quietly returns and sells to whoever's looking; they blast everyone who ever asked the instant a single unit lands, so forty people race for four and thirty-six are annoyed; or they bolt on an app that emails the whole list at once with no reservation, which is the same scramble with a nicer template. The first wastes the demand you already captured. The second and third turn a happy moment into a disappointment for most of the list, and risk overselling when several people buy the same last unit in the same minute.

The shape above fixes exactly that. It leans on the inventory feed the store already emits as the trigger, keeps the notify-me request small and de-duplicated so the queue is honest, and adds a notifier that respects the order people joined and never invites more people than there are units. The reserve link is the quiet hero: it converts “it’s back, go!” into “it’s held for you for fifteen minutes”, which is what makes first-come-first-told actually mean something. The odd cases — a restock that looks wrong, an item with no catalogue entry, a message that won’t deliver — are pulled out and put in front of a person.

The next four posts walk through each piece in turn: how a request gets captured, how a restock gets detected, how the waiting list gets notified, and how a reservation gets held and expired. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JULY 2, 2026 PART 2 OF 7 · BACK-IN-STOCK NOTIFIER SERIES ~8 MIN READ

How a back-in-stock request gets captured

Before anything can be sent, the system has to remember who asked, for what, and when — cleanly, once each, in order. This post is about that first step: how a “notify me” tap on a sold-out product becomes a single, de-duplicated place in a per-item queue, stamped with the moment it joined so the line stays fair when the stock finally comes back.

KEY TAKEAWAYS

- The trigger is a “notify me” tap that posts the shopper’s contact and the SKU to one Lambda Function URL — no API Gateway.
- The request is validated and normalised before anything is stored, so a malformed phone or a missing SKU never enters the queue.
- One active request per person per item: a second tap on the same product updates the existing place, it doesn’t create a new one.
- The opt-out list is checked at capture, so anyone who has unsubscribed is never quietly added back to a waiting list.
- Every accepted request is stamped with the moment it joined, and that timestamp is the queue — oldest first when the stock returns.

From a tap to a place in the queue

Everything starts with a small, deliberate action the shop rarely captures well: a customer standing on a sold-out product page who still wants the thing. Most storefronts show “out of stock” and leave it there. This system puts a “notify me” control on that page, and when it’s tapped, the browser posts a tiny payload — a name, a phone number or email, the SKU, and the channel the shopper picked — to a single Lambda Function URL. There’s no API Gateway in front of it; a Function URL is a plain HTTPS endpoint on the function itself, which is all a form post needs and the cheapest way to receive one.

The capture function's job is not to send anything — nothing goes out for days or weeks, until the item returns. Its job is to record the request cleanly and exactly once, and to stamp it with the moment it joined so that when the stock does come back, the queue is fair and unambiguous. Most of this post is about the checks between "a tap arrived" and "a place in the queue exists", because a waiting list is only worth anything if it's honest.

Prove it's well-formed, then prove it's new

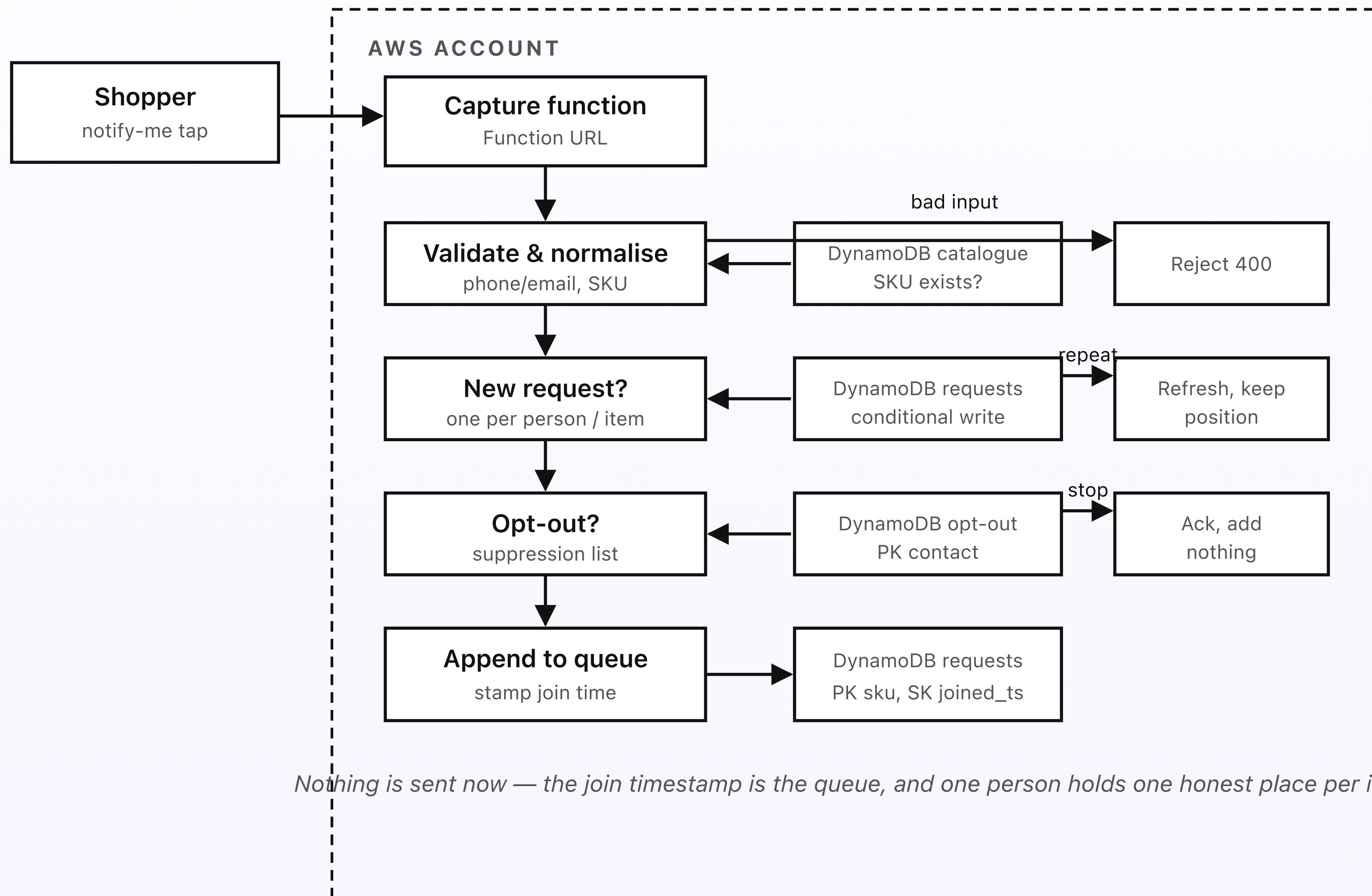
The first check is validity. A Function URL is public, so the function treats every field as untrusted: the phone number is parsed and normalised to E.164 (the `+44…` international form) or the email is validated, the SKU is checked against the store catalogue so a typo or a stale product id can't create a ghost queue, and the channel must be one the shop supports. Anything that fails is rejected at the door with a clear error and nothing is written. A light rate limit per source keeps a script from stuffing the list. Only a clean, real request gets past this line.

The second check is duplication, and it's what keeps the queue honest. People tap "notify me" twice, or come back a week later and tap it again, and none of that should buy them a second place in the line or a second text when the item returns. So the system keeps **one active request per person per item**: it writes the request keyed on the SKU and the contact, using a conditional write to DynamoDB, so a repeat tap finds the existing request and simply refreshes it rather than adding a duplicate. Crucially, a repeat tap does *not* reset the join time — the original position is preserved, because moving someone to the back for asking twice would be its own small unfairness.

Opt-out is checked before you join

One more gate runs before a request is accepted: the opt-out list. Anyone who has ever replied STOP to a text or unsubscribed from an email is recorded as suppressed, and the capture function checks that list before adding them to any queue. If a suppressed contact taps “notify me”, the tap is acknowledged politely but no waiting-list entry is created and nothing will ever be sent — because the cleanest message to handle is the one you correctly decide never to send. This is the same suppression list the notifier consults again at send time, so opt-out is enforced at both ends: you can’t be added if you’ve opted out, and you can’t be messaged even if you slipped through.

Only once a request is well-formed, matched to a real SKU, de-duplicated, and cleared against opt-out does the function commit it to the waiting list.



Nothing is sent now — the join timestamp is the queue, and one person holds one honest place per item.

Fig 2. The capture gate. A notify-me tap is validated and matched to a real SKU, de-duplicated so one person holds one place per item, and cleared against the opt-out list — then appended to the per-SKU queue stamped with the moment it joined, which is the order it'll be told in.

The timestamp is the queue

There is no separate "queue" data structure to keep in sync — the join timestamp *is* the queue. Each accepted request is written to the requests table partitioned by SKU and sorted by the moment it joined, so reading the waiting list for the walnut dining chair is a single query that comes back oldest-first, for free. When Part 4 needs to know who's next, it doesn't compute an order; it just reads in that natural order and stops when it has enough. Storing the ordering as the sort key rather than deriving it later is what makes "first come, first told' a property of the data instead of a promise the code has to keep.

Each request also carries a small amount of state that later parts lean on: which restock (if any) it was last notified for, so nobody is told twice for the same event; the channel the shopper chose; and a time-to-live so a request that's sat unwanted for months eventually falls off the list rather than lingering forever. The one case that doesn't flow straight through is a SKU the catalogue doesn't recognise — a discontinued line, a bad id from an old cached page. Rather than create a queue for a product that may never return, the function records the attempt and, if it looks like real demand, drops a note to a person to check, instead of silently swallowing it. Everything else becomes a clean place in the line, ready for a restock to call.

DESIGN RULES THAT SHAPED THE CAPTURE STEP

- One public surface for capture. A single Lambda Function URL receives the notify-me tap; there is no API Gateway.
- Validate before you store. A bad phone, email, or SKU is rejected at the door — ghost entries never reach the queue.
- One person, one place. A conditional write keeps a single active request per person per item; repeat taps refresh, they don't duplicate.
- Asking twice never costs your spot. A repeat tap keeps the original join time, so nobody is pushed back for being keen.
- Opt-out is checked here too. A suppressed contact is acknowledged but never added, so the list only holds people who can be told.
- The timestamp is the order. Requests are stored sorted by join time, so the queue is a property of the data, not a calculation.

PART 3 OF 7

JULY 2, 2026 PART 3 OF 7 · BACK-IN-STOCK NOTIFIER SERIES ~8 MIN READ

How a restock gets detected

The system doesn't poll the shop or guess; it waits for the store's own inventory to say a SKU is back. This post is about that trigger: how a raw stock-level webhook becomes a confident "this item genuinely restocked, by this many units, once" — and why telling a real restock from a stock level wobbling around the threshold is the whole job.

KEY TAKEAWAYS

- The trigger is a stock-level webhook from the store's inventory system, posted to one Lambda Function URL — no polling, no API Gateway.
- The webhook is verified by its signature before anything runs, so nobody can spoof a restock and make the system message the list.
- A restock is the *crossing* from below the threshold to above it — not every stock message, so a level wobbling near the line fires nothing.
- Each genuine restock gets a stable id and records the quantity available, which is what caps the batch so the system never oversells.
- One notify batch is enqueued per restock, idempotently, so provider retries or duplicate events can't notify the queue twice.

Waiting for the shop to say it's back

The system never guesses whether an item has returned, and it never polls the shop asking. It waits for the store's own inventory to tell it. Most e-commerce platforms emit an event when a SKU's stock level changes — an inventory-level or product-update webhook carrying the SKU, the new quantity, and a timestamp. That POST lands on a single Lambda Function URL, separate from the capture endpoint in Part 2 because it's a different trust relationship: this one comes from the shop platform, signed, and it's the trigger for every notice the system will ever send.

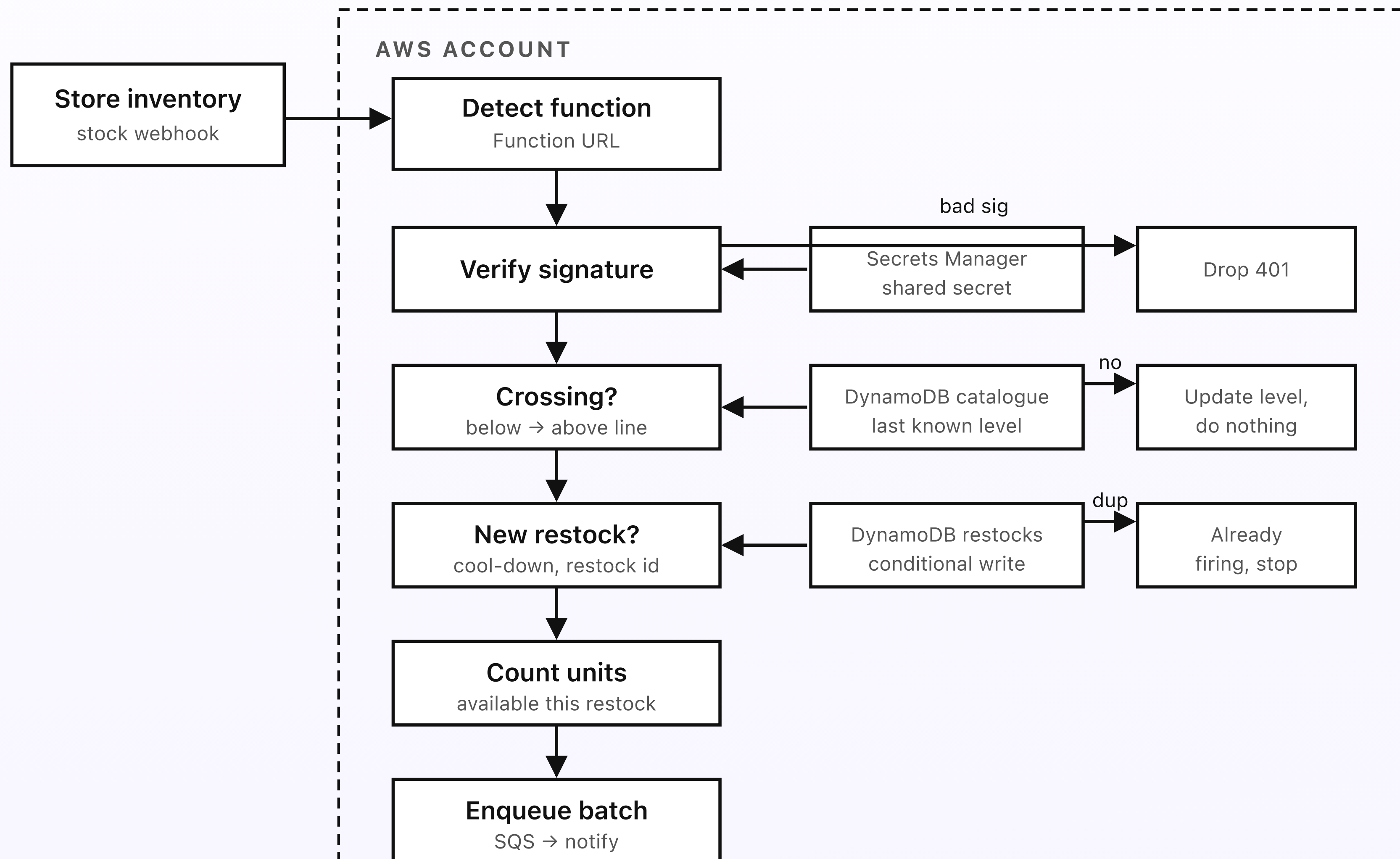
The detect function's job is deceptively narrow. It is not "stock changed, so message everyone". Stock levels change constantly — a sale drops the count, a return bumps it, a stocktake corrects it, the platform re-sends an event it thinks failed. The one moment worth acting on is the specific one where an item that was genuinely out (or below the threshold) comes back above it. Telling that single crossing apart from all the ordinary noise is the whole job of this step, and getting it wrong in either direction is expensive: miss the crossing and the queue never hears; over-fire and you spam people for a stock level twitching around zero.

Prove it's real, then prove it's a crossing

The first check is authenticity. The Function URL is public, so the function verifies the platform's signature — a hash of the request body signed with a shared secret held in Secrets Manager — before it trusts a single field. A bad signature is dropped with a `401` and nothing else happens. Without this, anyone who found the URL could post a fake "the walnut chair is back, quantity 500" and make the

system message the whole list; with it, only the shop’s platform can trigger a restock.

The second check is the one that matters most: is this a genuine restock, or just noise? The function reads the SKU’s last known level from the catalogue table, compares it to the new quantity, and applies the threshold from the settings doc (often simply “was at or below zero, now at or above one”, but a shop can set a higher bar — don’t bother the list until at least three are in). Only a transition from *below* the threshold to *above* it counts as a restock. A count that was already above the line and just went higher isn’t a restock — those people were told last time. A level that dips to zero and bounces back to zero fires nothing. To stop a flapping SKU (one that oscillates around the threshold as concurrent orders and restocks land) from firing repeatedly, the function also applies a short cool-down: once it has fired for a SKU, it won’t fire again for that SKU until the level has clearly settled and a sensible interval has passed.



Only a genuine crossing fires, exactly once — carrying the unit count that caps the batch so the list is never overs

Fig 3. Detecting a restock. A signed stock webhook is verified, checked for a genuine below-to-above crossing against the last known level, de-duplicated with a minted restock id and a cool-down, and its unit count recorded — then exactly one notify batch is enqueued.

Counting the units, and firing once

A confirmed crossing does two more things before it hands off. First, it records how many units came in — the quantity from the event, clamped to what the catalogue now says is genuinely sellable. This number is the whole reason the system can promise never to oversell: it is the cap the notifier in Part 4 obeys, the maximum number of people who will be invited for this restock. Four chairs in means at most four notices go out; the fifth person waits. Second, it mints a stable **restock id** for this event and writes it with a conditional write to the restocks table, so the very act of recording the restock is what makes firing idempotent. If the platform re-sends the same event, or two near-simultaneous webhooks describe the same crossing, the conditional write fails on the second one and no second batch is enqueued.

Only once a restock is verified, confirmed as a real crossing, de-duplicated, and counted does the function enqueue exactly one notify-batch job on SQS — carrying the SKU, the restock id, and the unit count — for Part 4 to work through. Everything slow (reading the whole waiting list, calling the model, sending messages) happens off that queue, not in the webhook, so the store's platform gets a fast acknowledgement and never waits on the shop's behalf. The odd case here — a restock with an implausible quantity, or a SKU the catalogue doesn't know — isn't acted on blindly; it's recorded and escalated to a person, because

messaging a queue for a mistake is far worse than a moment's delay while someone checks.

DESIGN RULES THAT SHAPED THE DETECT STEP

- Wait to be told. The store's inventory webhook is the trigger; the system never polls and never guesses a return.
- Verify before you trust. A bad signature is dropped with a 401 — the URL being public is fine because the secret isn't.
- A restock is a crossing, not a change. Only a genuine move from below the threshold to above it fires; ordinary stock noise doesn't.
- Count the units up front. The quantity available is recorded now, because it's the cap that stops the notifier ever overselling.
- Fire exactly once. A minted restock id written conditionally makes the batch idempotent — retries and duplicates enqueue nothing.
- When it looks wrong, a person. An implausible quantity or an unknown SKU escalates rather than messaging the queue on a mistake.

PART 4 OF 7

JULY 2, 2026 PART 4 OF 7 · BACK-IN-STOCK NOTIFIER SERIES ~8 MIN READ

How the waiting list gets notified

A restock with four units and thirty people waiting is where fairness is won or lost. This post is about the notifier: how it reads the queue in the exact order people joined, messages only as many as there are units to sell, phrases each notice in the store's voice, and hands each recipient a short-lived reserve link — so the first to respond genuinely get the stock and nobody is oversold.

KEY TAKEAWAYS

- The notifier reads the per-SKU queue oldest-first, so people are told in the exact order they joined — no jumping.
- The batch is capped to the units available for this restock, so the system never invites more people than there is stock to sell.
- One Bedrock Haiku 4.5 call phrases the notice per restock; code personalises it and injects each recipient's reserve link.
- Each recipient is marked notified for this restock id, so a retry or a re-fired batch can never text the same person twice.
- Opt-out is checked again at send time, and the reserve link — not the model — is what makes first-come, first-served real.

The moment fairness is won or lost

A restock with four units and thirty people waiting is where a back-in-stock system either keeps its promise or breaks it. Blast all thirty and you've started a scramble: four win, twenty-six are told the thing they wanted is gone again before they've finished reading, and if several buy the last unit in the same minute you've oversold. The notifier exists to make that moment orderly. By the time it runs, Part 3 has already handed it everything it needs on the queue: the SKU, the restock id, and the number of units available. Its whole task is to turn that into the right notices, to the right people, in the right order — and to stop.

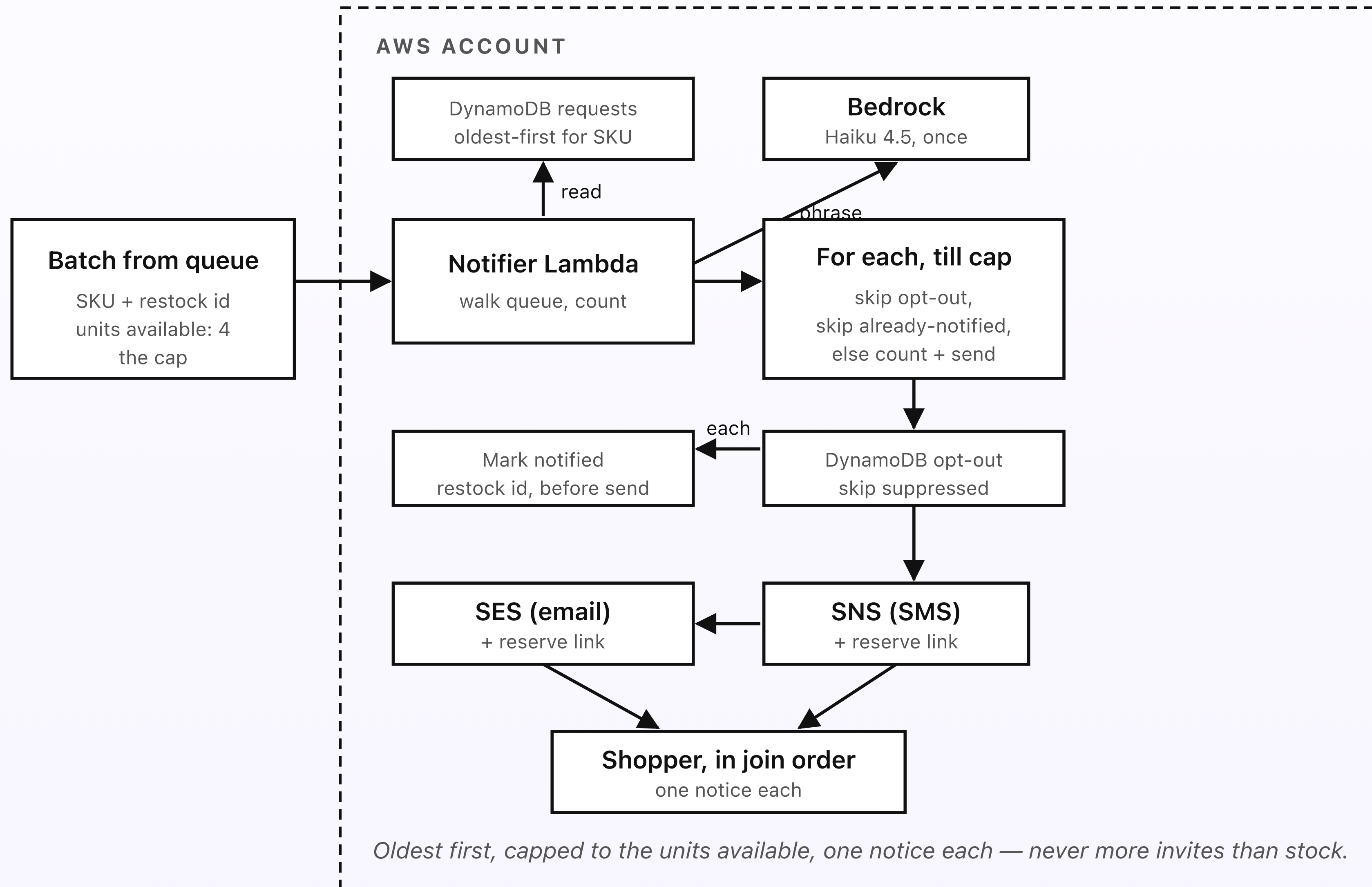
It reads the waiting list for that SKU straight from the requests table, which is stored sorted by join time, so the query comes back oldest-first with no sorting to do. It walks that list from the front, and it counts. The unit count is a hard cap: for four chairs it will notify at most four people, then stop, no matter how many are waiting behind them. This is the single rule that makes overselling impossible at the notice stage — you cannot oversell a queue you never invited past the number of units you have.

Oldest first, capped, and skipping the suppressed

Walking the queue isn't quite "take the first four", because a couple of the people at the front may no longer be reachable. As it goes, the notifier checks each candidate against two things: the opt-out list (someone who unsubscribed after joining is skipped and never counts against the cap), and whether they've already been notified for *this* restock id (so a re-run doesn't double-message anyone). A skipped person doesn't consume one of the four slots — the notifier simply moves to the next eligible person — so four reachable people are always told for four units, not "four positions, some of them dead ends". It fills the cap with people who can actually act on it.

For each person it does notify, it writes a marker recording that this request was notified for this restock id *before* the message goes out, using a conditional write. That ordering matters: the marker is what makes the whole batch idempotent. If the SQS message is redelivered, or the batch runs twice, the second pass sees the marker and skips — so exactly one notice reaches each person per restock, even though the underlying queue delivery is at-least-once. The reserve link the message carries is minted here too, one per recipient, tied to that person and that

restock, and live for only a short window (Part 5 is entirely about what that link does).



Oldest first, capped to the units available, one notice each — never more invites than stock.

Fig 4. Notifying the list. The notifier reads the queue oldest-first, phrases the notice once with Bedrock, and walks the front of the queue skipping opt-outs and the already-notified; each eligible person is marked and sent one message with their own reserve link, and the loop stops the moment the cap of available units is filled.

One notice, phrased once

Wording is the one part a shop actually cares about, so this is where the single model call lives — and it's one call per *restock*, not one per person. The notice for "the walnut dining chair is back" reads the same for everyone in the batch, so Bedrock's Claude Haiku 4.5 is asked once to phrase it in the store's voice, handed only the facts it may use: the product name, the store name, and a placeholder where the reserve link will go. It is told to write one short message that says the item is back, that the recipient is near the front of the list, and that the link holds one for a short window — and to use only those facts. Then code takes that single phrasing and personalises the cheap parts — the first name, the reserve link — per recipient. Phrasing once and templating the rest keeps the model cost to a fraction of a penny per restock rather than per message, and keeps every notice in a batch consistent.

The reserve link is never written by the model. The model emits a token like `{{reserve}}`, and the code substitutes each recipient's real, signed reserve URL afterwards — because a language model is perfectly capable of subtly mistyping a URL, and a broken reserve link is the one error that quietly wastes the whole notice. The draft is validated before anything sends: under the SMS length limit, exactly one link, an opt-out hint, no invented promises about price or delivery. If the draft fails, or Bedrock is slow, the notifier falls back to a fixed template —

"Good news — {product} is back at {store}. You're near the front of the list; this link holds one for you for {window} minutes: {link}. Reply STOP to opt out." — which is plainer but always correct and always on time. The model supplies the warmth; the code guarantees the link, the cap, and the order.

DESIGN RULES THAT SHAPED THE NOTIFIER

- Oldest first, always. The queue is read in join order and walked from the front; nobody is reordered or jumped.
- The cap is the unit count. At most one notice per available unit — the loop stops when the cap is filled, so it can't oversell.
- Skip, don't spend a slot. An opt-out or already-notified person is skipped without using up one of the units.
- Mark before you send. A conditional "notified for this restock" marker makes the batch idempotent — one notice each, even on retry.
- Phrase once, personalise in code. One Haiku call per restock; the name and the reserve link are injected deterministically.
- The link is the code's job. The model emits a token; the real signed reserve URL is substituted per recipient so it can never be mangled.

PART 5 OF 7

JULY 2, 2026 PART 5 OF 7 · BACK-IN-STOCK NOTIFIER SERIES ~8 MIN READ

How a reservation gets held

A notice without a hold is just a starting gun for a scramble. This post is about the reservation: how clicking the reserve link locks a single unit for a short window — safely, without ever taking stock below zero — and how a scheduled sweep returns any hold that isn't used to the next person waiting, so the queue keeps moving and stale requests fall away.

KEY TAKEAWAYS

- Each notice carries a short-lived, signed reserve link; clicking it holds a single unit for that person for a set window.
- The hold is a conditional write that decrements the available count and can never take it below zero — so it can't oversell.
- The link is single-use and tied to one request and one restock, so it can't be forwarded, replayed, or shared to jump the queue.
- An EventBridge Scheduler sweep expires holds that aren't converted in time, returning the unit to the pool for the next person.
- The same sweep expires stale requests, so a queue full of people who've moved on doesn't block the ones who still want the item.

A notice without a hold is a starting gun

The notice in Part 4 does its job only if being told actually means getting the thing. If the message just said "it's back, go and buy it", then four notices for four chairs would still be a race — the fastest tapper wins, being first in the queue counts for nothing, and if two people reach checkout with the last chair in the same few seconds the shop oversells. The reserve link is what turns the notice from a starting gun into a genuine reservation. It carries a signed token unique to one recipient and one restock, and clicking it does one thing: it holds a single unit for that person for a short window — long enough to get through checkout, short enough that the queue keeps moving.

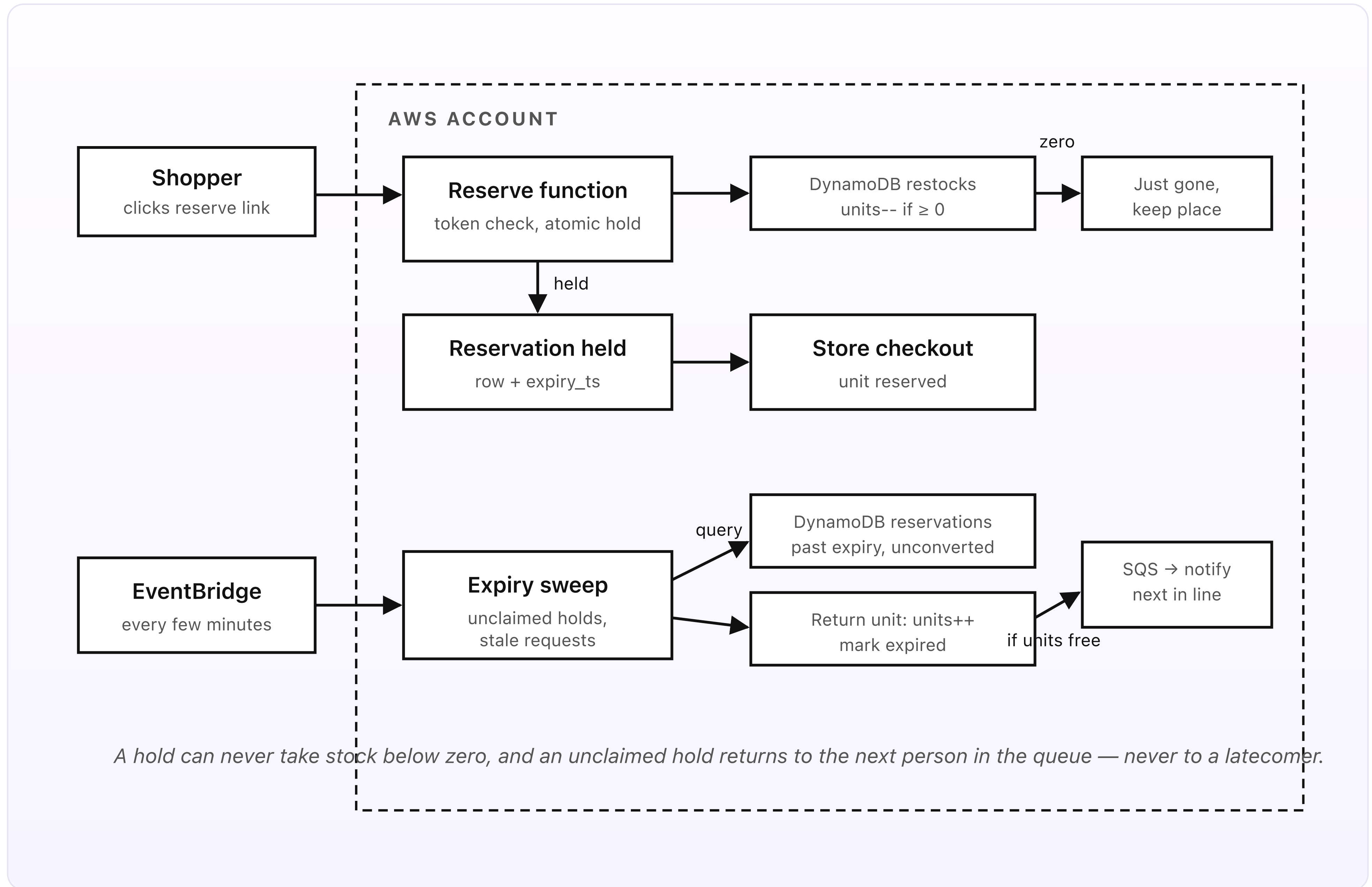
The link lands on its own Lambda Function URL — the third public surface, separate from capture and the inventory webhook, because it's a different job with a different shape: a customer clicking a link in a text and expecting to end up at checkout. The function validates the token's signature (so a guessed or edited link is rejected), checks it hasn't already been used or expired, and only then attempts the hold. Everything about this step is built around one guarantee: the shop can promise a held unit to whoever clicks, without ever promising the same unit twice.

The hold that cannot oversell

The hold itself is a single conditional write, and it's the linchpin of the never-oversell promise. The restock record for that SKU carries a count of units still available to hold. When a reserve link is clicked, the function decrements that count with a condition that it stays at or above zero, and writes a reservation row keyed to the token with an expiry timestamp a few minutes out. If the count is

already zero — every unit in this restock is spoken for — the condition fails, the hold is refused, and the shopper is shown an honest “just gone, we’ll keep your place” rather than a false promise. Because the decrement is atomic, two people clicking their links in the same instant can never both take the last unit: DynamoDB serialises the two writes, one succeeds, one sees zero and is refused. There is no window in which the shop believes it has stock it doesn’t.

A successful hold writes a reservation and hands the shopper to the store’s checkout with the item reserved — a real cart, held under their name. The reservation row records who holds it, for which restock, and when it expires. That expiry is the important half of the design: a hold is not a sale. Someone can click the link, glance at the price, and wander off, and if a held unit could sit reserved forever, the queue behind them would be stuck — a chair reserved for a person who’s never coming back is exactly as useless as a chair sold to a bot. So every hold has a clock on it, and something has to watch that clock.



A hold can never take stock below zero, and an unclaimed hold returns to the next person in the queue — never to a latecomer.

Fig 5. Holding and expiring a reservation. A reserve-link click holds one unit with a conditional decrement that can't go below zero, then sends the shopper to checkout. A scheduled sweep returns any hold not converted in time, tops the units back up, and offers the freed unit to the next person in line — and clears stale requests as it goes.

The sweep that keeps the line moving

Watching a clock across thousands of holds is a job for something that runs on a schedule, not a trigger — the thing that needs acting on is an event that *didn't* happen: a checkout that never completed. That's the expiry sweep, driven by EventBridge Scheduler on a short cadence (every few minutes). Each run queries the reservations table for holds that are past their expiry and were never converted to a sale. For each one, it does the mirror image of the hold: it returns the unit by incrementing the available count back up on the restock record, and marks the reservation expired so it's never processed twice. The increment is safe to run repeatedly because the expired marker makes it idempotent — a hold is returned exactly once, however many times the sweep runs.

Returning the unit is only half of it; the point is to give it to the *next person in line*, not to whoever happens to click a stale link. So when the sweep frees a unit and finds people still waiting on that SKU, it enqueues a small top-up notify batch — the same job Part 4 processes — for the number of units just freed. Marcus, fifth in the queue, is told only now, because a unit has genuinely become available for him and the restock's cap has room again. The queue always advances by exactly the number of holds that lapsed, in order, and a latecomer who was never on the list is never slipped in ahead of the people who waited. The same sweep also does a quieter housekeeping pass: requests that have sat on a queue past their time-to-live — someone who asked months ago and clearly moved on — are expired off

the list, so a restock isn't spent messaging people who no longer care while the genuinely keen wait behind them.

DESIGN RULES THAT SHAPED THE RESERVATION

- A notice buys a hold, not a race. The reserve link holds a single unit for the recipient for a short, fixed window.
- The decrement can't go below zero. A conditional write makes overselling impossible even when two links are clicked at once.
- One link, one use. The token is signed, single-use, and tied to one request and one restock — it can't be shared or replayed.
- A hold is not a sale. Every hold has an expiry, because a unit reserved forever blocks the whole queue behind it.
- Freed units go to the next in line. The sweep returns a lapsed hold and tops up a notice to the next person, never a latecomer.
- Stale requests fall away. Time-to-live clears people who've moved on, so a restock isn't wasted on a queue full of ghosts.

PART 6 OF 7

JULY 2, 2026 PART 6 OF 7 · BACK-IN-STOCK NOTIFIER SERIES ~6 MIN READ

What the back-in-stock notifier costs

A system that costs more than the sales it recovers is a toy. This post is the cost breakdown: every AWS service this design touches, what each adds up to at around 120 notify-me requests a month, and why the total lands near $2.10 — plus what happens to the bill when a busier shop runs it at ten times the volume.

KEY TAKEAWAYS

- About $2.10/month at roughly 120 notify-me requests, and the fixed cost is almost nothing — nothing runs between restocks.
- The two biggest lines are the outbound SMS and one small Bedrock call per restock. Everything else is cents.
- The only real fixed cost is Secrets Manager: two keys at $0.40 each, billed whether or not an item ever restocks.
- Because the model is called once per restock — not once per person — Bedrock stays cheap even when a queue is long.
- SMS carrier fees vary by country and provider; the numbers here are a UK-leaning estimate, not a fixed AWS price list.

Where the money goes

The system is serverless end to end, so there's no instance ticking over between restocks and no idle bill. You pay for a notice only when an item actually comes back. At a typical independent-shop volume — call it 120 notify-me requests a month across a handful of restocks, with the notices, reserve clicks, and expiry sweeps that follow — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two keys — inventory-webhook signing key, SMS/reserve-token key ($0.40 each)	$0.80
SNS (SMS)	One text per notified shopper (~90), plus a few top-up texts from the sweep	$0.55
Bedrock (Claude Haiku 4.5)	One notice-phrasing call per restock (a handful a month)	$0.30
DynamoDB (on-demand)	Requests, restocks, reservations, opt-out, audit — small reads and writes	$0.15
CloudWatch Logs	Function logs, 7-day retention	$0.10
SES	Email notices for shoppers who chose email, plus exception handovers	$0.08
Lambda (Python 3.14, arm64)	Capture, detect, notifier, reserve, sweep, catalogue-sync	$0.05

AWS service	What it does here	Monthly
EventBridge Scheduler	The expiry sweep and the catalogue sync	$0.05
SQS + DLQ	Buffering restock batches to the slower model and messaging calls	$0.02
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~120 requests/month**	**$2.10**

The shape of that bill is the point. The only line that costs money while the system sleeps is Secrets Manager — two keys at $0.40 each, $0.80 a month no matter what, which is well over a third of the total at this volume. Everything else is genuinely usage-priced and rounds to zero between restocks. The two lines that move with volume are the ones that actually reach the customer: the SMS itself and the Haiku call that phrases the batch. And because the model runs once per *restock* rather than once per person, a queue of forty people costs the same Bedrock spend as a queue of four — the notice is phrased once and templated the rest of the way. The capture, detect, reserve, and sweep machinery — all the real work of keeping the queue fair and never overselling — together costs less than the SMS bill alone.

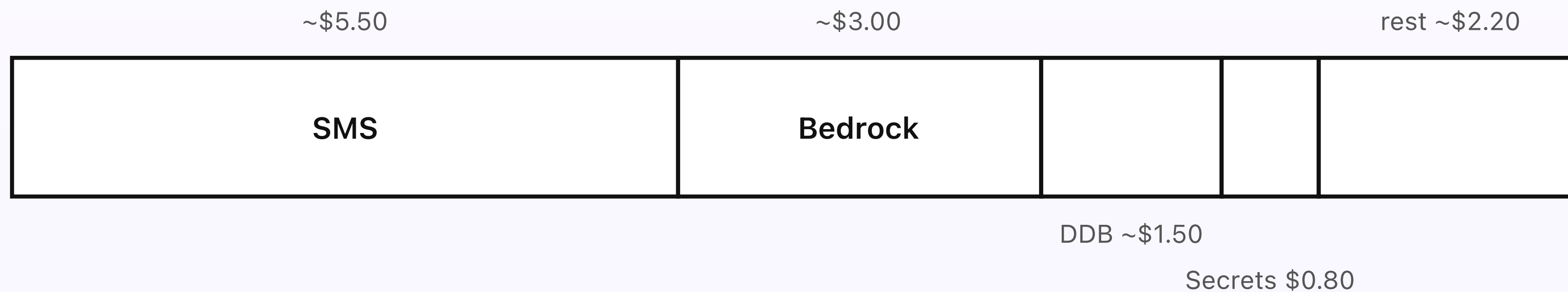
The line that isn't purely AWS

The SMS line deserves a caveat. AWS prices outbound SMS per message, and the exact rate depends on the destination country and the mobile carrier — a UK mobile is a few pence, other countries differ, and some routes add carrier surcharges. The $0.55 here is a UK-leaning estimate for around 90 outbound texts (many shoppers choose email, which is far cheaper through SES); your real number will track your country and your provider. If a shopper picks email over SMS, that notice moves onto the near-free SES line instead — so the channel mix, not just the volume, shapes this row. Either way, SMS is the one line worth watching as volume grows, which is exactly why the AWS Budgets alarm sits on top of the whole thing.

What ten times the volume costs

Push this to a busier shop — 1,200 notify-me requests a month, ten times the volume, with proportionally more restocks — and the bill lands near $13, not $21. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules stay at a few cents, and AWS Budgets stays free. What scales is the genuinely usage-priced work — roughly $5.50 of SMS for ten times the texts, about $3 of Bedrock for ten times the restocks, and a few dollars more spread across DynamoDB, logs, SES, and Lambda. Even then, the two things that reach the customer dominate, and all the machinery in between stays close to free.

Monthly cost — ~1,200 requests — total ~$13



Fixed lines don’t move with volume; only the SMS and the once-per-restock model call scale, so the bill grows sub-linea

Fig 6. The monthly bill at ten times the base volume, about 1,200 requests. SMS and Bedrock are the bulk of it; the fixed lines stay put, so the total grows sub-linearly — near $13, not ten times $2.10.

The honest way to read this: the AWS bill is rounding error against what a recovered sale is worth. A single walnut dining chair or a pair of trainers is worth far more than $2.10, and this design turns demand you’d otherwise lose into orderly sales — without ever overselling and having to apologise. Even at $13 a month for a busy shop, the system pays for itself the first time it turns a restock into a sale that would have vanished into a scramble, and the few notices that go

unclaimed simply pass down the queue to the next person who still wants the thing.

DESIGN RULES THAT SHAPED THE COST

- Pay per restock, not per hour. No always-on compute means no idle bill between the times an item actually returns.
- Spend the model once per restock. One Haiku call phrases the whole batch, so a long queue costs no more model spend than a short one.
- Cheap work stays cheap. Capturing, detecting, reserving, and sweeping are plain Lambda and DynamoDB, cents at this scale.
- Know your one fixed cost. Secrets Manager is the only line that bills while the system sleeps.
- Watch the SMS line. It’s the part that scales and whose price varies, so the Budgets alarm sits right on top of it.

PART 7 OF 7

JULY 2, 2026 PART 7 OF 7 · BACK-IN-STOCK NOTIFIER SERIES ~10 MIN READ

Engineering reference: the back-in-stock notifier architecture

This is the back-in-stock notifier with the friendly labels removed: the real resource names, the runtime, the table key schemas, the three public Function URLs, the reservation-expiry schedule, and the IAM scope. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Six Lambda functions, all Python 3.14 on arm64, with the notifier fed through one SQS queue and a dead-letter queue.
- Three public surfaces: Function URLs on `bisn-capture`, `bisn-detect`, and `bisn-reserve` — no API Gateway.
- Five DynamoDB tables, all on-demand: requests (the queue), restocks (the cap), reservations (the holds), opt-out, and audit.
- Exactly-once is enforced by three conditional writes: the dedup key, the restock id, and the atomic units decrement.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by the notifier. Single region, `eu-west-2`.

The architecture, for engineers

This is the same system as Part 1 with the friendly labels removed and the real resources named. Everything is in one region, `eu-west-2` (London), in one account. There is no API Gateway, no NAT Gateway, and nothing always-on; the only inbound surfaces are three Lambda Function URLs, outbound messaging goes through SNS and SES, and the one asynchronous hop — a detected restock to the notifier — is buffered on a single SQS queue.

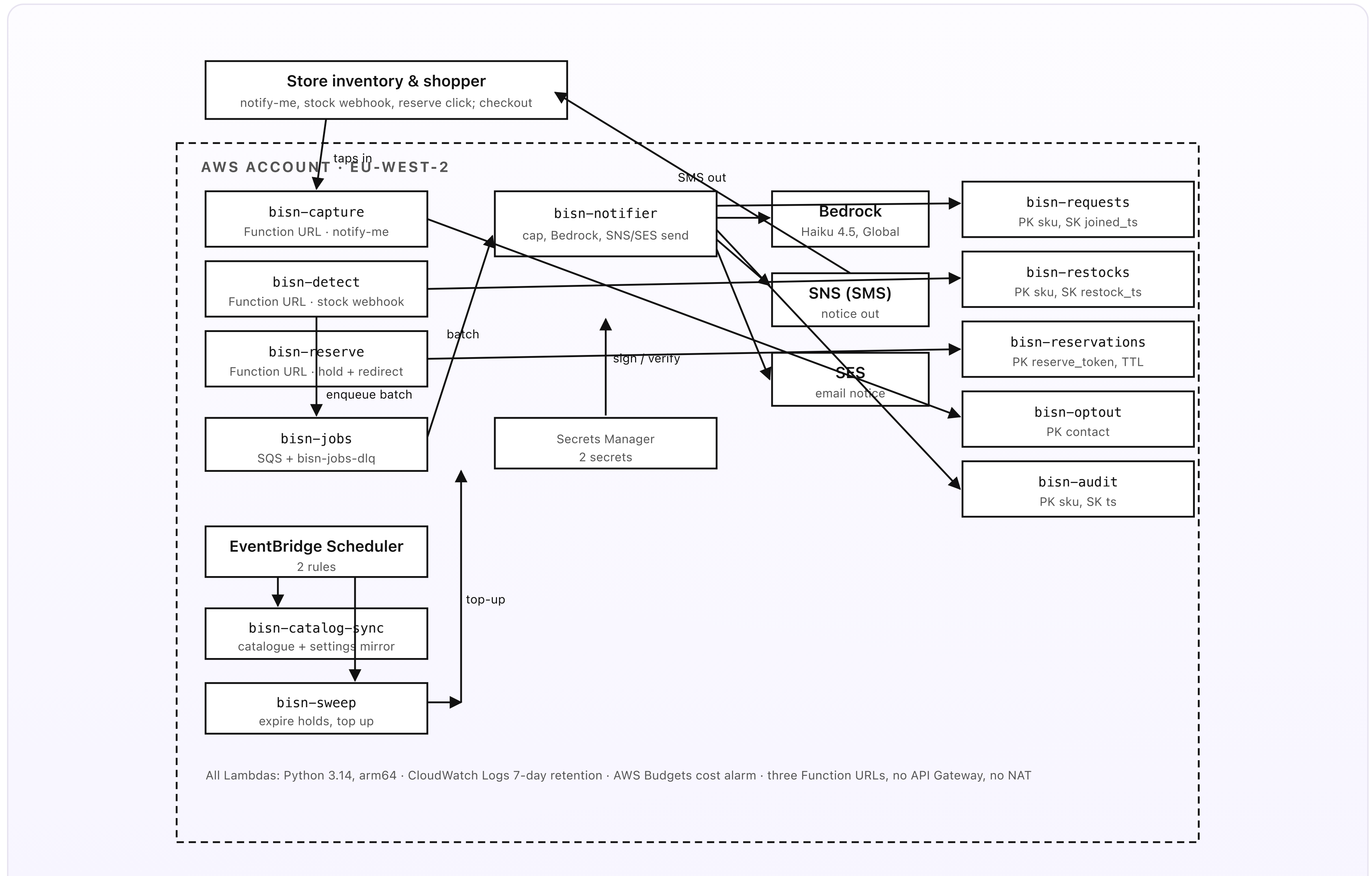

Store inventory & shopper
notify-me, stock webhook, reserve click; checkout
taps in
AWS ACCOUNT · EU-WEST-2
SMS out
bisn-capture
Function URL · notify-me
bisn-notifier
cap, Bedrock, SNS/SES send
Bedrock
Haiku 4.5, Global
bisn-requests
PK sku, SK joined_ts
bisn-detect
Function URL · stock webhook
SNS (SMS)
notice out
bisn-restocks
PK sku, SK restock_ts
batch
bisn-reserve
Function URL · hold + redirect
sign / verify
bisn-reservations
PK reserve_token, TTL
SES
email notice
enqueue batch
bisn-optout
PK contact
bisn-jobs
SQS + bisn-jobs-dlq
Secrets Manager
2 secrets
bisn-audit
PK sku, SK ts
EventBridge Scheduler
2 rules
top-up
bisn-catalog-sync
catalogue + settings mirror
bisn-sweep
expire holds, top up
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm · three Function URLs, no API Gateway, no NAT

Fig 7. The back-in-stock notifier drawn for engineers: three Function URLs (`bisn-capture`, `bisn-detect`, `bisn-reserve`), an SQS-buffered notifier, five DynamoDB tables, Bedrock called only by the notifier, SNS for SMS and SES for email, and two scheduled jobs. One region, one account, no API Gateway.

The DynamoDB data model

Five tables, all on-demand, all in `eu-west-2`. The schema is deliberately shaped so that the two hard guarantees — queue order and never-oversell — fall out of the keys rather than out of application logic.

- `bisn-requests` — the waiting list. PK `sku`, SK `joined_ts` (an ISO-8601 timestamp with a short random suffix for uniqueness). Because the sort key is the join time, a single `Query` on a SKU with `ScanIndexForward=true` returns the queue oldest-first — the ordering is the data. A global secondary index, `bisn-requests-by-contact` (PK `sku#contact`), backs the one-active-request-per-person check with a conditional write. Each item holds the channel, the request state (`waiting` / `notified` / `reserved` / `converted`), the `last_notified_restock` id, and a `ttl` for staleness expiry.
- `bisn-restocks` — one item per detected restock. PK `sku`, SK `restock_ts`, plus a minted `restock_id`. The item carries `units_available` — the decrementable counter that is the whole never-oversell mechanism — and the restock `state`. Writing this item conditionally on the `restock_id` not already existing is what makes detection idempotent.
- `bisn-reservations` — the holds. PK `reserve_token` (the signed, single-use token from the notice). Each row records the `sku`, the `request_id`, the

`restock_id`, an `expiry_ts`, and a `state` (`held` / `converted` / `expired`). A DynamoDB TTL on `expiry_ts` tidies old rows, but the sweep, not TTL, drives the functional expiry so timing is precise.

- `bisn-optout` — PK `contact` (E.164 number or email). The suppression list, checked at capture and again at send.
- `bisn-audit` — PK `sku`, SK `ts`, append-only. One row per notice sent and per restock detected, holding the facts each was built from — the record you reach for when a customer asks "why was I (not) told?".

Lambda functions

Six functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. Each does one job; only the notifier runs off the queue, so the public webhooks always answer fast.

- `bisn-capture` — a Function URL. Validates the notify-me tap, normalises the contact, checks the SKU against the catalogue and `bisn-optout`, and writes the request to `bisn-requests` via a conditional put on `bisn-requests-by-contact` so a repeat tap refreshes rather than duplicates. Nothing slow happens here.
- `bisn-detect` — a Function URL. Verifies the store signature, compares the new level to the last known one, and on a genuine below-to-above crossing mints a `restock_id`, writes `bisn-restocks` conditionally, records `units_available`, and enqueues one batch on `bisn-jobs`.
- `bisn-notifier` — SQS-triggered on batch jobs. Reads `bisn-requests` oldest-first, walks the front skipping opt-outs and the already-notified, caps at

`units_available`, makes the single Bedrock call, mints per-recipient reserve tokens, sends via SNS or SES, and writes `bisn-audit` — marking each recipient `notified` for the restock id with a conditional write before sending.

- `bisn-reserve` — a Function URL. Validates the signed token, holds a unit with a conditional decrement of `units_available` on `bisn-restocks`, writes the `bisn-reservations` row, and redirects to the store checkout. A failed condition (units at zero) returns an honest "just gone".
- `bisn-catalog-sync` — scheduled. Mirrors the product catalogue and per-SKU settings (name, threshold, reserve window, voice) from the store into the catalogue store the other functions read.
- `bisn-sweep` — scheduled. Queries `bisn-reservations` for holds past `expiry_ts` that never converted, increments `units_available` back on `bisn-restocks`, marks each reservation `expired`, expires stale `bisn-requests`, and enqueues top-up batches on `bisn-jobs` for any freed units.

Exactly-once, and never-oversell

Three conditional writes carry the whole correctness story, because SQS is at-least-once and webhooks retry. First, the **dedup key**: `bisn-capture`'s conditional put on `sku#contact` guarantees one place in the queue per person per item, however many times they tap. Second, the **restock id**: `bisn-detect`'s conditional write on a fresh `restock_id` means a re-sent webhook enqueues no second batch, and `bisn-notifier`'s conditional "notified for this restock" marker — written before the send — means a redelivered SQS message re-sends nothing. Third, the **units decrement**: `bisn-reserve`'s conditional `units_available >= 1` before subtracting is what makes overselling impossible even under simultaneous

clicks; DynamoDB serialises the writes, and the loser sees zero. The sweep's `expired` flag closes the loop, so a returned unit is returned exactly once.

The queue between `bisn-detect` and `bisn-notifier` is `bisn-jobs`, with `bisn-jobs-dlq` as its dead-letter queue after five failed attempts. Because the notifier is idempotent per recipient per restock, a redelivery after a partial failure is safe — it resumes rather than re-sends. A batch that keeps failing (a malformed job, a downstream outage) lands in the DLQ with the restock id intact, alarms, and can be replayed once the cause is fixed; no notices are silently lost, and none are duplicated on replay.

IAM scope, observability, and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `bisn-capture` can read the catalogue and `bisn-optout` and conditionally write `bisn-requests` — it cannot call Bedrock, SNS, or SES. `bisn-detect` can read the signing secret, write `bisn-restocks`, and send to `bisn-jobs` — nothing more. `bisn-notifier` is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile; it can publish to SNS and SES, read `bisn-requests` and `bisn-optout`, and write `bisn-restocks` markers and `bisn-audit`, but cannot delete from any table. `bisn-reserve` can conditionally update `bisn-restocks` and write `bisn-reservations`, and holds the token-signing secret — it has no messaging permissions at all. The scheduled functions hold only the narrow catalogue and table permissions they need and no inbound surface.

Observability is CloudWatch throughout: structured logs at 7-day retention, metrics on batch size versus units available (a notifier that ever sends more notices than units is a bug worth paging on), on DLQ depth, and on reserve-conversion rate. An AWS Budgets alarm watches monthly spend — with SMS the line most likely to move, it's the cheapest early warning that volume, or a loop, is running hot. Everything runs in `eu-west-2` from one infrastructure-as-code definition; the only cross-Region path is Bedrock's Global inference profile, which routes the single model call for capacity and holds no data. The model id is `anthropic.claude-haiku-4-5` via that Global profile, invoked only by `bisn-notifier`.

That's the whole system: a notify-me tap becomes an ordered place in a queue, a signed inventory webhook becomes a counted restock, and the two meet in a notifier that tells people oldest-first, never past the number of units in hand, each with a link that holds one unit for a short while and passes it on if they don't. Five tables, six small functions, three conditional writes doing the real work, and one region — a fair queue and a firm never-oversell promise for a couple of dollars a month.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Six small Lambdas beat one that does everything; only the notifier runs off the queue.
- Order and cap live in the schema. The join-time sort key is the queue; the units counter is the cap — not application guesswork.
- Three conditional writes hold the line. The dedup key, the restock id, and the units decrement give exactly-once and never-oversell.
- Least privilege, per role. Only the notifier can call Bedrock, SNS, and SES; only reserve can decrement the units count.
- Fail into the DLQ, not into silence. Idempotent retries resume safely; a stuck batch alarms and replays without duplicating notices.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called once per restock.