7-PART SERIES · FREE COMPANION

Upsell recommender

Most add-on sales are lost in the same quiet way: the customer buys the bike and never hears that the shop also stocks the lights and the lock that go with it. This is the design of a small serverless system that turns a placed order into one tasteful, well-judged nudge: it reads what was bought and the customer's history, asks a model to propose a shortlist of add-ons, and then lets deterministic catalogue rules decide — in stock, genuinely complementary, sensibly priced, not already owned. It sends one follow-up with a one-tap add link, tracks whether the add-on was taken up so the offer can be measured, and caps how often anyone is nudged so it never feels like spam. One suggestion per order, opt-out honoured, and no offer at all when nothing fits. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Upsell recommender

PART 1 OF 7

JULY 7, 2026 PART 1 OF 7 · UPSELL RECOMMENDER SERIES ~10 MIN READ

An upsell recommender on AWS for a few dollars a month

An add-on sale is the quietest kind a shop loses: the customer buys the bike, the checkout thanks them, and nobody ever mentions the lights and the lock that go with it. This post walks through the design of a small serverless system that turns a placed order into one well-judged nudge — the single add-on that actually complements what was bought — with a one-tap link to add it, and nothing at all when nothing fits.

KEY TAKEAWAYS

- A placed order fires a webhook from your store; a short while later the customer gets one nudge for the add-on that actually goes with it.
- A model proposes a shortlist of add-ons; deterministic catalogue rules dispose of it — in stock, complementary, sensibly priced, not already owned.
- The offer is one tap to accept: a signed add link that drops the item straight into the store's basket.
- It suggests exactly one add-on per order, caps how often anyone is nudged, honours opt-out, and sends nothing when nothing fits.
- Designed on AWS for about $2.60/month at roughly 150 orders a month. It only ever suggests — a human never has to.

The whole system on one page

Before any code, here's the shape of what we're designing. Every shop that sells more than one thing has natural pairings — a bike and its lights, a bag of beans and a grinder, a dog bed and the blanket that matches it — and almost none of them get mentioned at the moment they'd land. The checkout thanks the customer and goes quiet, and the add-on sale that was one sentence away is simply never made. The system below catches that moment: an order comes in, and a single, well-judged nudge goes back out a short while later, with a one-tap link to add the thing that goes with it.

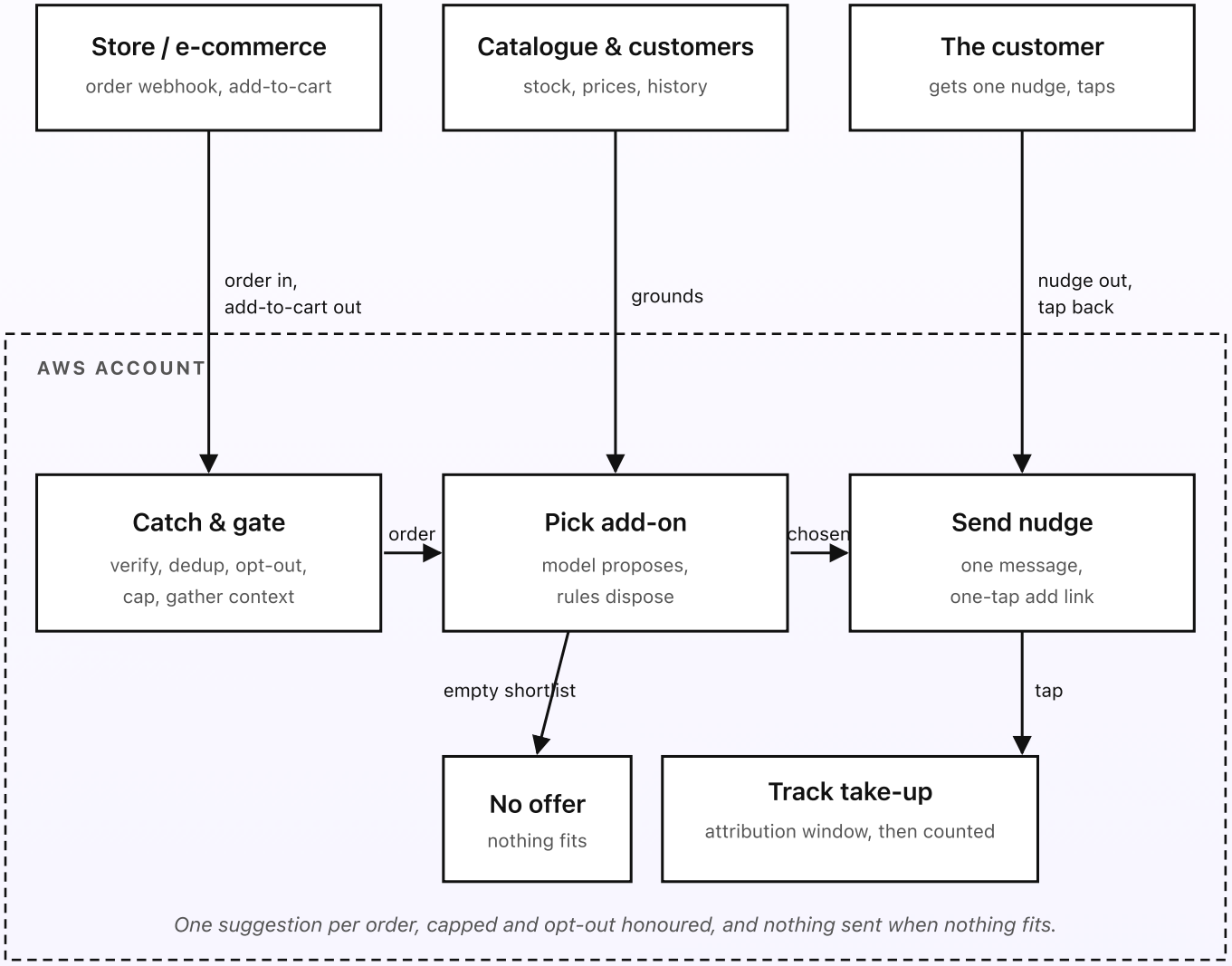

Store / e-commerce
order webhook, add-to-cart
Catalogue & customers
stock, prices, history
The customer
gets one nudge, taps
order in,
add-to-cart out
grounds
nudge out,
tap back
AWS ACCOUNT
Catch & gate
verify, dedup, opt-out,
cap, gather context
order
Pick add-on
model proposes,
rules dispose
chosen
Send nudge
one message,
one-tap add link
empty shortlist
tap
No offer
nothing fits
Track take-up
attribution window, then counted
One suggestion per order, capped and opt-out honoured, and nothing sent when nothing fits.

Fig 1. Three things outside, four pieces inside AWS. An order comes in from the store; Catch & gate decides whether a suggestion is even warranted, Pick add-on lets a model propose and the catalogue rules dispose, Send nudge delivers one one-tap offer, and Track take-up counts whether it landed. When the shortlist empties, the flow ends at "No offer".

What you set up once (the outside)

- **Store and add-to-cart.** Whatever platform already takes your orders — a hosted shop or e-commerce backend that can do two things: fire a webhook the moment an order is placed, and accept a link that adds a known product to a basket or starts a pre-filled checkout. The order webhook is the trigger for everything; the add-to-cart URL is where the one-tap link lands. You point the store's webhook at one AWS URL and store its keys in Secrets Manager. This is the only moving part you don't own, and it's covered in Part 2.
- **Catalogue and customer records.** A product list — one row per item with its price, current stock, category, and a tag or two for what it complements — alongside each customer's order history. You already keep both; this just mirrors them where the system can read them, so a pick can know that the track pump is in stock, costs a sensible fraction of the bike, and isn't already in this customer's past orders. A small settings doc holds the voice, the send delay, the frequency cap, and the attribution window.
- **The customer.** The person who placed the order. They receive, at most, one nudge for one add-on — and only if a good one exists — with a single tap to accept it. If they've opted out, or been nudged too recently, or nothing genuinely fits, they hear nothing at all. The system never chases, never sends a second offer for the same order, and never fills a quiet moment with a weak suggestion.

What runs on every order (the inside)

- **Catch and gate.** The store posts an order event to one Lambda Function URL. The function verifies the signature, dedups so one order can only ever make one offer, checks the customer against the opt-out list and the per-customer frequency cap, and gathers the basket and history. Only then does it decide a suggestion is worth attempting. This is Part 2.
- **Pick add-on.** One Bedrock Haiku 4.5 call proposes a ranked shortlist of add-ons with a short line for each. Then deterministic catalogue rules filter that shortlist — in stock, complementary, sensibly priced, not already owned — and take the top survivor. If the shortlist empties, no offer is made. The model proposes; Python disposes. This is Part 3.
- **Send nudge.** After a short, deliberate delay, one message goes out with a one-tap add link — a signed token on a second Function URL that records the tap and redirects straight into the store's add-to-cart. One nudge, one add-on, one tap to say yes. This is Part 4.
- **Track take-up.** The tap is recorded inside an attribution window and tied back to its offer, the frequency cap is updated so the next nudge stays honest, and a scheduled sweep closes the window so every offer ends up counted as taken, ignored, or never sent. This is Part 5.

In plain words

It's Tuesday evening and Tom buys a commuter bike from Ridgeback Cycles for £520. The store fires an order webhook. The system records the order, sees Tom has no lights or lock in his history, and asks the model for add-ons that go with a commuter bike; it comes back with a shortlist — a lights-and-lock bundle, a set of

mudguards, a track pump. The catalogue rules take over: the mudguards for his frame are out of stock, so they're dropped; the track pump he already bought last year, so it's dropped; the lights-and-lock bundle is in stock, complements the bike, and at £45 is a sensible fraction of the order. It wins. An hour later Tom gets one message: "Your new bike will want lights and a lock — here's our commuter bundle at £45, one tap to add: ridgeback.uk/a/…". He taps, it lands in his basket, and he checks out. Nobody at the shop lifted a finger.

The next morning a different order comes in from a customer who bought the same bundle a fortnight ago and has already had a nudge this month. The model would happily propose something — a helmet, perhaps — but it never gets asked: the frequency cap gate stops the pick before the model runs, because a second nudge inside the window would read as pestering. And a third order, for a gift card, produces an empty shortlist — nothing in the catalogue genuinely complements a gift card — so the system does the most disciplined thing it can and sends nothing. One good offer when there is one; silence when there isn't.

DESIGN RULES THAT SHAPED EVERY DECISION

- One order, one offer. Each order can produce at most a single suggestion — repeats and retries collapse to one, and it never sends twice.
- The model proposes, Python disposes. Bedrock suggests a shortlist; deterministic catalogue rules make the actual choice.
- No offer beats a weak offer. If nothing is in stock, complementary, priced right and not already owned, the system stays silent.
- Never nag. A per-customer frequency cap limits how often anyone is nudged, and an opt-out is permanent.
- Saying yes is one tap. The offer carries a one-tap add link straight into the basket; there is nothing to type.
- Every offer is counted. Taken, ignored, or never sent — each one is measured, so the shop can tell what actually works.

Why this shape

Most shops handle add-ons one of three ways: they don't suggest them at all, they bolt a "you might also like" strip onto the product page and hope, or they blast every customer a generic promo a few days later. The first leaves obvious sales on the table — the person buying a bike is never more receptive to lights than in the minute they bought it. The second is untargeted and easy to ignore. And the third trains customers to filter the shop into spam, because it fires

whether or not there's anything worth saying. The gap is a single, well-judged, well-timed suggestion — and nothing off-the-shelf sends one and only one.

The shape above fills exactly that gap and nothing more. It leans on the store you already run as the trigger, keeps the catalogue and order history you already maintain as the source of truth, and adds a small system that makes one good suggestion per order — when a good one exists. The common case — "oh, I do need lights" — becomes a single tap. The cases where nothing fits, or the customer has heard enough, resolve to silence rather than a weak offer, which is the thing that keeps the whole channel worth trusting.

The next four posts walk through each piece in turn: how an order triggers a suggestion, how the add-on gets picked, how the offer gets sent, and how a take-up gets tracked. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JULY 7, 2026 PART 2 OF 7 · UPSELL RECOMMENDER SERIES ~7 MIN READ

How an order triggers a suggestion

Before the system suggests a single thing, it has to earn the right to: prove the order is real and new, check the customer hasn't opted out or been nudged too recently, and gather what they actually bought. This post is about that gate — how a raw order webhook becomes a safe, context-rich job that a suggestion can be built from, or is quietly dropped before it ever costs a model call.

KEY TAKEAWAYS

- The trigger is an order webhook from your store, posted to one Lambda Function URL — no API Gateway.
- The webhook is verified by its signature before anything else runs, so a stranger can't make the system suggest things to people.
- A conditional write keyed on the order id means one order can only ever create one offer, however many times the webhook fires.
- Opt-out and a per-customer frequency cap are checked before any model call — the cheapest offer is the one you correctly decide not to build.
- Only once an order is real, new, allowed and gathered does it become a suggest job; everything slow happens later.

From an order to a job

Everything starts with an event the shop already produces but rarely acts on in the moment: an order was placed. E-commerce platforms expose this as a webhook — when a customer checks out, the store sends a small HTTP POST with the order id, the customer id, and the line items. That POST lands on 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 webhook needs and the cheapest way to receive one.

The order function's job is not to suggest anything. Its job is to decide whether a suggestion is worth attempting at all, and to set it up so it can only ever happen once. Most of this post is about the checks that sit between "an order arrived" and "a suggest job is queued", because that gap is where an upsell system either stays welcome or tips over into nagging.

Prove it's real, then prove it's new

The first check is authenticity. A Function URL is public, so the first thing the function does is verify the store's signature — a hash of the request body signed with a shared secret held in Secrets Manager. If the signature doesn't match, the request is dropped with a `401` and nothing else happens. Without this, anyone who found the URL could fabricate orders and make your system message strangers; with it, only your store can trigger a pick.

The second check is duplication, and it's what enforces the one-order-one-offer promise. Stores retry webhooks they think failed, and some fire more than once for a single checkout — an order created event, then a paid event, then a fulfilled event, all for the same purchase. The system must build at most one offer per order. So the function writes a record keyed on the order id using a conditional write to DynamoDB: the first event for that order wins and creates the job; any later events or retries see the existing record and stop. One order, one offer, however many times the store shouts about it.

Should we suggest at all?

Two more gates decide whether a suggestion is even allowed, regardless of what was bought. **Opt-out** is the suppression list: anyone who has ever asked not to be marketed to is recorded, and the system will never nudge that customer again. **The frequency cap** is the anti-nagging rule: no customer is nudged more than once inside a set window — thirty days by default — so a regular who orders every week still only hears a suggestion occasionally. The function reads the customer's last-nudged timestamp and count, and if a nudge would fall inside the window, it stops here. Both checks run before a single token is spent on the model, because the cheapest suggestion to handle is the one you correctly decide not to make.

Only once an order is verified, de-duplicated, not opted out, and inside the frequency cap does the function gather the context a pick will need — and enqueue the job.

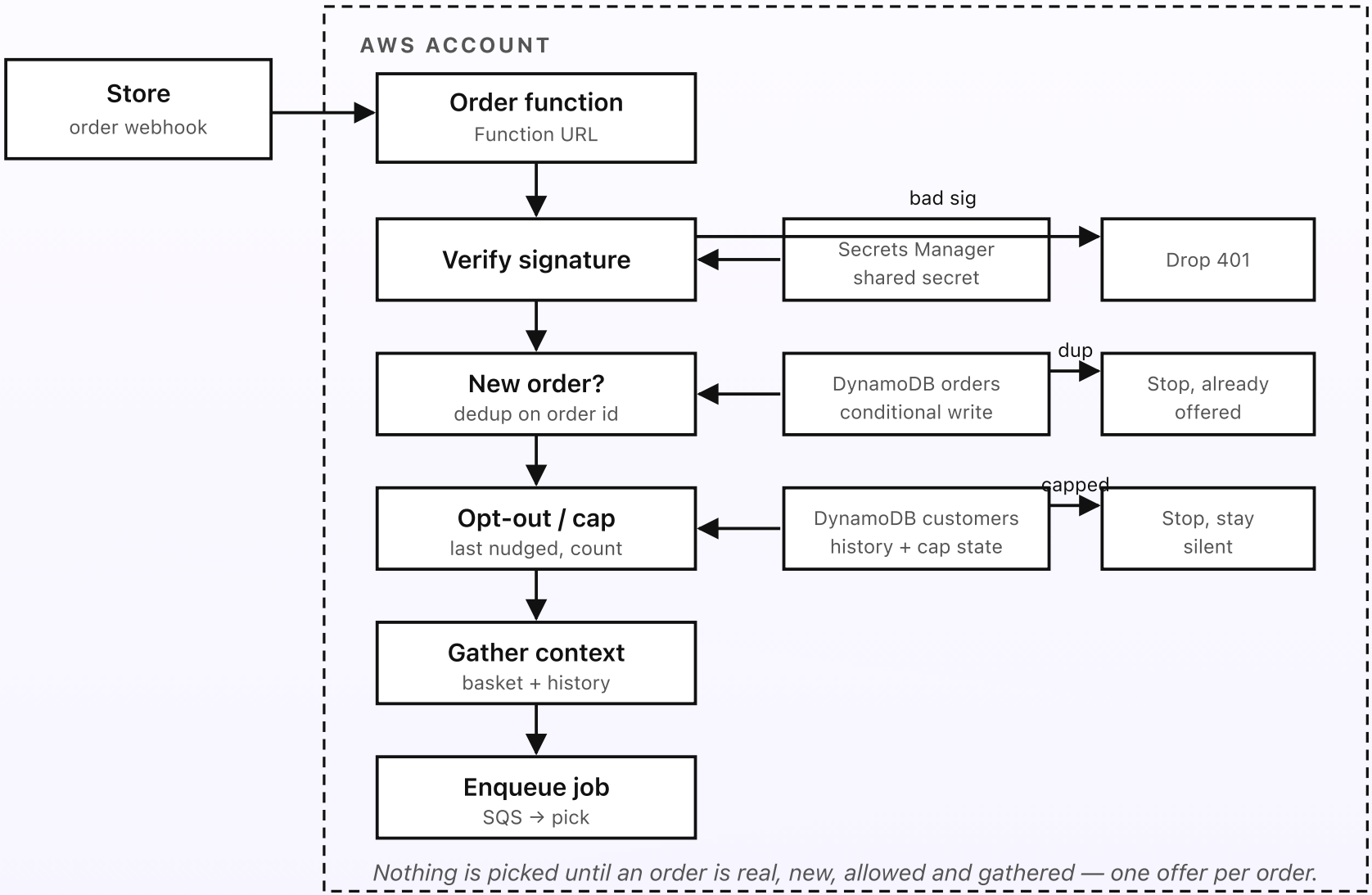

AWS ACCOUNT
Store
order webhook
Order function
Function URL
Verify signature
bad sig
Secrets Manager
shared secret
Drop 401
New order?
dedup on order id
DynamoDB orders
conditional write
dup
Stop, already
offered
Opt-out / cap
last nudged, count
DynamoDB customers
history + cap state
capped
Stop, stay
silent
Gather context
basket + history
Enqueue job
SQS → pick
Nothing is picked until an order is real, new, allowed and gathered — one offer per order.

Fig 2. The order gate. An order webhook is verified by signature, de-duplicated with a conditional write on the order id so retries make one offer, checked against the opt-out list and the frequency cap, and enriched with the basket and history — only then is a suggest job enqueued.

Gathering the context

Gathering is where the order stops being a bare event and becomes something a good suggestion can be built from. The function reads the line items from the webhook — what was actually bought, in what quantities, at what value — and joins them to the customer's order history mirrored from the store: what they've bought before, so the pick can avoid suggesting something already owned, and how much they typically spend, so the price rules have a sense of proportion. It also notes the order value, because a £45 add-on makes sense against a £520 bike and looks absurd against a £6 pack of inner tubes.

None of this is a judgement yet; it's just assembling the facts. The one thing the gate deliberately does *not* do is call the catalogue or the model — those belong to the pick, which is slower and costs money, and should only run for orders that have already earned it. So the output of this step is a small, clean job on the SQS queue: the order id, the customer id, the basket, the relevant history, and the order value, ready for Part 3 to turn into a single suggestion. Everything expensive happens on the far side of that queue, which keeps the public webhook fast and cheap no matter how busy the shop gets.

DESIGN RULES THAT SHAPED THE GATE

- One public surface. A single Lambda Function URL receives the order webhook; there is no API Gateway and no other way in.
- Verify before you trust. A bad signature is dropped with a 401 before any work — the URL being public is fine because the secret isn't.
- One order, one offer. A conditional write on the order id collapses store retries and duplicate events into a single job.
- Decide not to suggest, cheaply. Opt-out and the frequency cap are checked before the model runs — a suppressed order costs nothing.
- Gather, don't judge. The gate assembles the basket and history; it never calls the catalogue or the model itself.
- Keep the webhook fast. Everything slow — the catalogue join, the model, the send — happens behind the SQS queue.

PART 3 OF 7

JULY 7, 2026 PART 3 OF 7 · UPSELL RECOMMENDER SERIES ~8 MIN READ

How the add-ons get picked

This is the heart of the system, and the place its guardrails matter most. A model is good at spotting that a road bike and a track pump go together; it is hopeless at knowing whether the pump is in stock, priced right, or already in the customer's garage. So the model proposes a shortlist and Python disposes of it through hard catalogue rules — and a suggestion the model loves but the rules reject never reaches anyone.

KEY TAKEAWAYS

- One Bedrock Haiku 4.5 call per order proposes a ranked shortlist of add-ons with a short line for each — the only place a model runs.
- The model proposes; Python disposes. Deterministic catalogue rules are the hard filter that makes the actual choice.
- The rules drop anything out of stock, not genuinely complementary, wrongly priced, or already owned — run in that order, over the model's list.
- The top surviving candidate wins. If the whole shortlist is filtered out, no offer is made — silence is a valid, common outcome.
- A suggestion the model loves but the rules reject can never reach a customer, because the rules run after the model, not before.

The one place judgement is shared

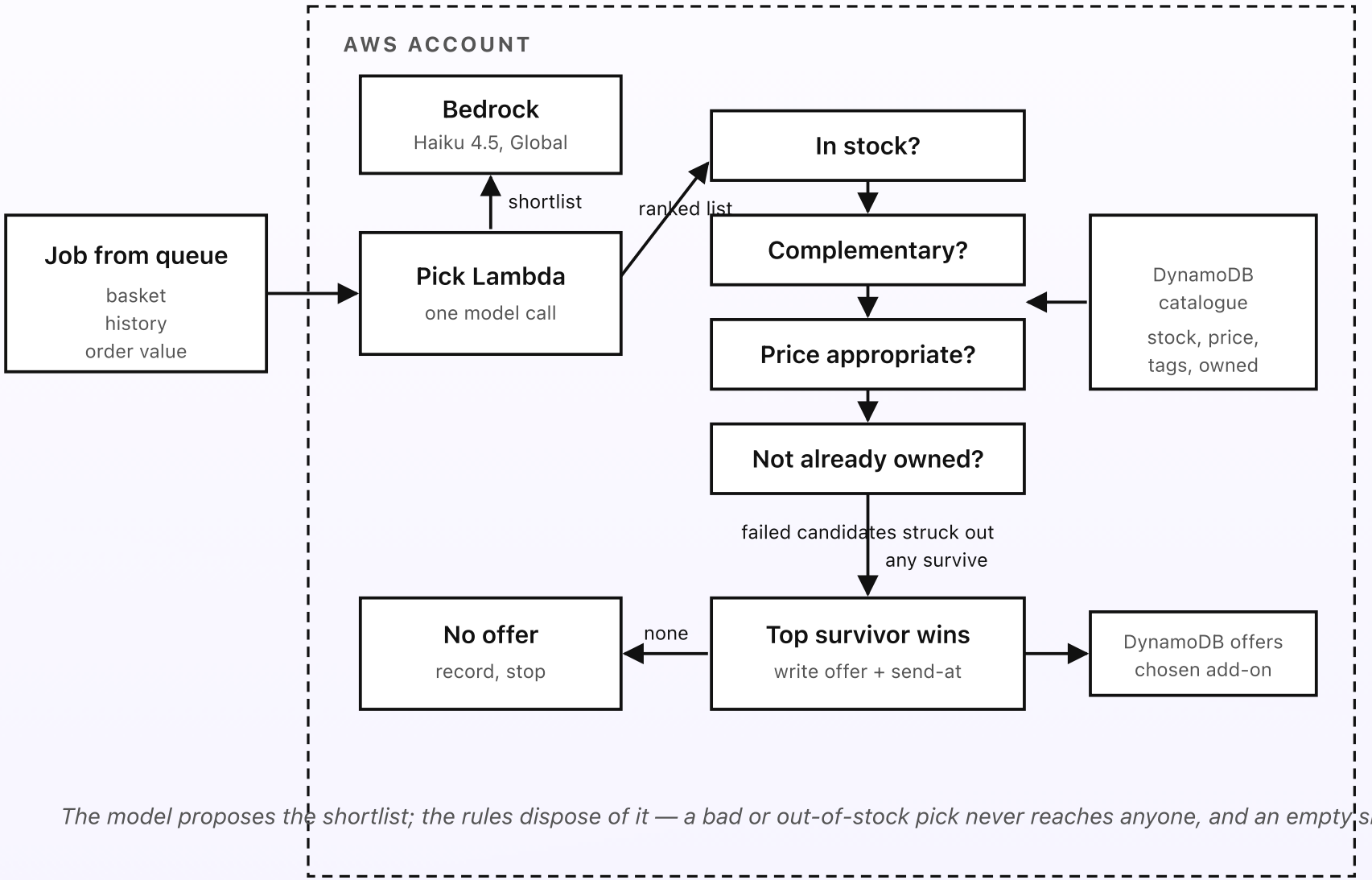
By the time this step runs, the gate in Part 2 has proved the order is real and new, cleared the customer against opt-out and the frequency cap, and gathered the basket and history. The job on the queue carries what was bought, what the customer already owns, and how much the order was worth. What's left is the actual recommendation — and this is the one place in the system where a model and deterministic code share the work, so it's worth being precise about who does which half.

A model is genuinely good at one thing here: knowing that a road bike goes with a track pump and mudguards, that a bag of single-origin beans goes with a burr grinder and a descaler, that a dog bed goes with a matching blanket. That's associative knowledge, and it's exactly what an off-the-shelf catalogue lacks. But a model is hopeless at the things that decide whether a suggestion is any good in practice: it doesn't know what's in stock this morning, what the margins are, what this customer already bought last year, or what price looks sensible against this order. So the split is clean. The model *proposes*; the catalogue rules *dispose*.

The model proposes a shortlist

One Bedrock call, using Claude Haiku 4.5, is handed the basket, a short summary of the customer's history, and a slice of the catalogue — the candidate add-ons in the relevant categories, with their names and tags but not the final say. It is asked to return a *ranked shortlist* of the add-ons that best complement what was bought, each with a one-line pitch in the shop's voice. Haiku is the right tool because the task is small and bounded: rank a handful of candidates and write a sentence each, no reasoning chain, no tools. It's fast and it costs a fraction of a penny — which matters, because unlike the rest of the system this call runs on *every* eligible order, not just the ones that convert.

Crucially, the model's output is a *proposal*, not a decision. It returns something like "1. commuter lights-and-lock bundle — 'your new bike will want lights and a lock'; 2. mudguards — 'keep the road spray off'; 3. track pump — 'top up before every ride'". Every one of those is a candidate SKU with a pre-written line attached. Nothing is sent yet. Nothing is even chosen yet. All the model has done is put its shortlist on the table, best first, for the rules to work through.



The model proposes the shortlist; the rules dispose of it — a bad or out-of-stock pick never reaches anyone, and an empty shortli

Fig 3. Picking the add-on. One Bedrock call proposes a ranked shortlist; deterministic catalogue rules filter it in order — in stock, complementary, sensibly priced, not already owned — and the top survivor wins. If every candidate is struck out, the flow ends at "No offer" and nothing is sent.

The rules dispose of it

Now the deterministic half runs, and it is deliberately unglamorous: a short list of hard filters applied to the model's shortlist in order, each reading the catalogue mirror. **In stock** — the SKU must have positive stock right now, or it's dropped; there is no faster way to lose a customer's trust than to offer them something you can't sell. **Complementary** — the candidate must carry a catalogue tag that genuinely pairs it with something in the basket; the model might free-associate its way to a suggestion, but the tag is the shop's own considered opinion, and it wins. **Price appropriate** — the add-on must sit inside a sensible band relative to the order value, so a £300 wheelset is never offered against a £40 order, and a 50p valve cap is never the headline suggestion on a £520 bike. **Not already owned** — anything in the customer's history is removed, because suggesting the pump they bought last spring makes the whole system look like it isn't paying attention.

The candidates are walked in the model's ranked order, and the first one that passes every filter is the winner — the shop's preferences (the tags, the stock, the price bands) acting as a hard gate on the model's preferences (the ranking). The chosen SKU, its price, and the model's pre-written line are written to the offers table with a send-at time, ready for Part 4. And here is the guardrail that matters most: because the rules run *after* the model and not before, there is no path by which a suggestion the model liked but the rules reject can reach a customer. The

model never touches stock, price, or ownership; it only ever ranks. The filter is the last word.

When nothing fits

Sometimes every candidate is struck out — the complementary items are all out of stock, or the customer already owns them, or the model's shortlist was thin because the order genuinely has no natural add-on (a gift card, a lone spare part, a one-off clearance item). When that happens, the system does the disciplined thing and makes *no offer*. The order is recorded as "no fit" so it can be counted later, and nothing is sent. This is not a failure mode; it's a feature. A system that always finds something to suggest is a system that will cheerfully suggest rubbish, and a customer who gets one weak nudge trusts the next one less. By making silence a first-class outcome — and a common one — every nudge that *does* go out has earned its place. The frequency cap stops the system nudging too often; the empty-shortlist rule stops it nudging when it has nothing worth saying.

DESIGN RULES THAT SHAPED THE PICK

- One call, one shortlist. A single Haiku call proposes ranked candidates — no chains, no second opinions, no tools.
- The model proposes, Python disposes. The model ranks; the deterministic catalogue rules make the actual choice.
- The rules run last. Filters apply after the model, so an out-of-stock or already-owned pick can never slip through.
- In stock, complementary, priced right, not owned. Four hard filters, in that order; the first survivor wins.
- No fit, no offer. An empty shortlist sends nothing and is recorded — silence is a valid, frequent outcome.
- The catalogue is the authority. Stock, tags and price bands are the shop’s opinion, and they outrank the model every time.

PART 4 OF 7

JULY 7, 2026 PART 4 OF 7 · UPSELL RECOMMENDER SERIES ~8 MIN READ

How the offer gets sent

By the time this step runs, the add-on is chosen and the wording is written. All that's left is to deliver it well: to wait a beat so it doesn't crowd the checkout, send exactly one nudge, and make saying yes a single tap. This post is about the send and the one-tap add link — a Function URL redirect that turns "go on then" into an item in the basket.

KEY TAKEAWAYS

- The nudge goes out after a short, deliberate delay — long enough not to crowd the checkout, soon enough to still be relevant.
- A scheduled sender releases due offers; it re-checks opt-out and the cap at send time, so a late opt-out still stops the message.
- The offer is one message carrying the chosen add-on, its price, and a single one-tap add link — nothing to type.
- The add link is a signed token on a second Lambda Function URL that records the tap and 302-redirects into the store's add-to-cart.
- The link URL is built by code from the chosen SKU, never written by the model, so it can never point at the wrong product.

A beat, then one message

By the time this step runs, the add-on is chosen and its line is written. What's left is delivery, and delivery has two jobs: send at the right moment, and make saying yes effortless. The right moment is not the instant of checkout — a nudge that lands while the customer is still reading the order confirmation feels grabby, like a shop assistant following you to the door. So the offer carries a *send-at* time set a short, deliberate delay after the order: an hour or so by default, tunable per shop. Long enough that the checkout has settled; soon enough that a commuter bike still feels like a fresh purchase and lights are an obvious afterthought rather than a cold re-approach a week later.

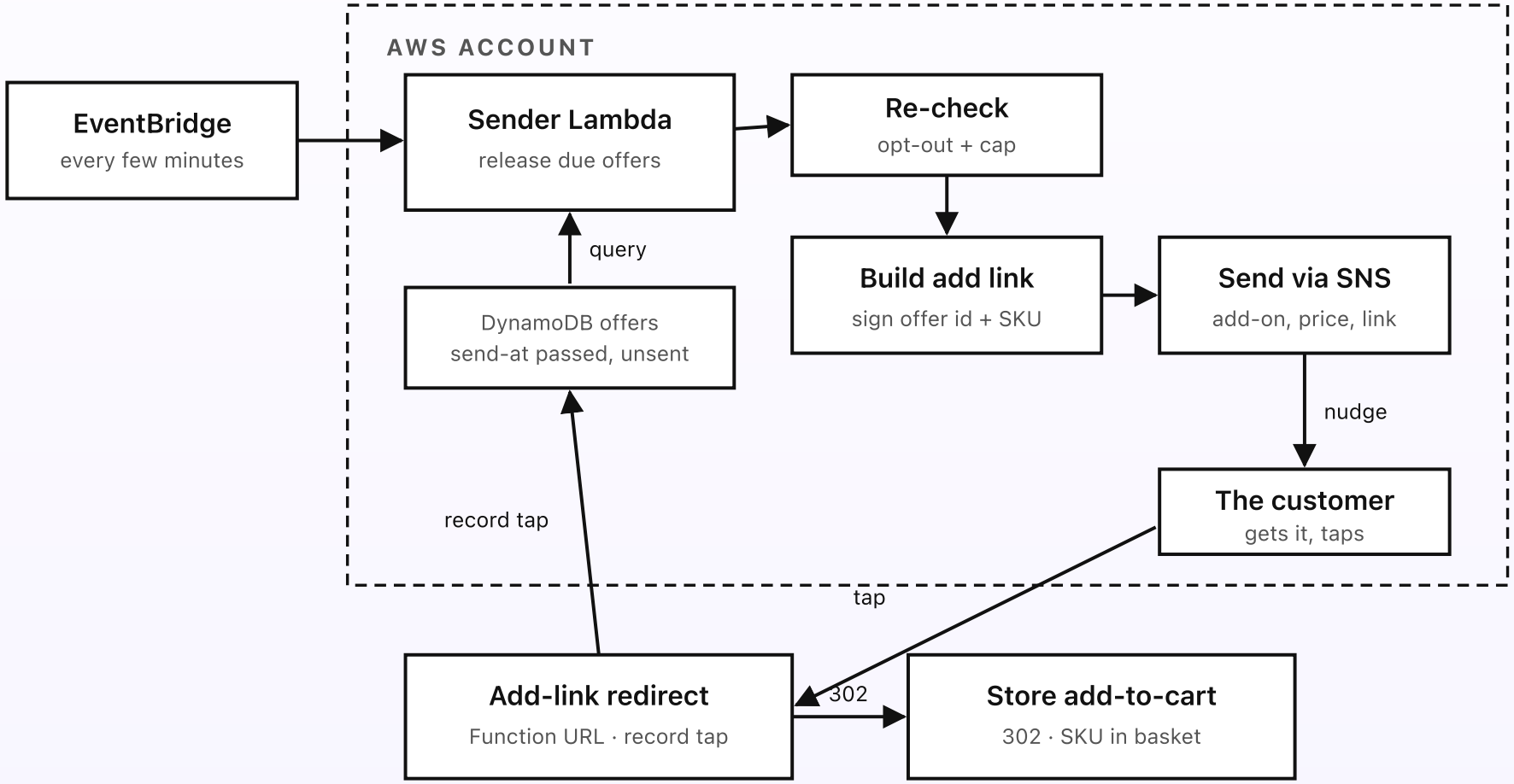
The send itself is driven by a scheduled sender, not by a per-order timer. On a fixed cadence it queries the offers table for anything whose send-at has passed and which hasn't yet been sent, and releases those. This is cheaper and simpler than scheduling one timer per order, and it gives the system a natural place to re-check the guardrails: right before sending, it looks again at the opt-out list and the frequency cap. A customer who opted out in the hour between order and send is caught here, and the message is dropped. The decision to send is never older than the send itself.

The one-tap add link

The message is short: the chosen add-on, its price, the model's one line, and a single link. Everything hangs on that link being one tap. It is not a coupon code to copy, not a "visit our shop" homepage, not a search the customer has to redo — it is a signed token on a second Lambda Function URL that, when tapped, does two

things and redirects. First it records the tap against the offer (the attribution signal Part 5 is built on). Then it issues an HTTP 302 to the store's own add-to-cart or pre-filled-checkout URL for exactly the SKU that was chosen, so the item is sitting in the basket before the customer has finished looking at their phone. The whole distance between "go on then" and "it's in the basket" is one tap and one redirect.

The token is signed so it can't be forged or edited — a customer can't swap the SKU for a pricier item at a discount, and a stranger can't mint links — and it carries the offer id and the SKU, nothing sensitive. Critically, the destination URL is assembled by code from the chosen SKU, not written by the model. The model wrote the sentence; the code owns the link. A language model is perfectly capable of "improving" a URL or mistyping a product id, and a link that adds the wrong thing (or nothing) is the one error that quietly wastes the whole nudge. By keeping the link out of the model's hands, the tap always adds exactly the add-on the rules chose.



One message, one link; the link is built by code from the chosen SKU, records the tap, and redirects straight into the basket.

Fig 4. Sending the offer. A scheduled sender releases offers whose delay has elapsed, re-checks opt-out and the cap, builds a signed one-tap link and sends one nudge via SNS. When the customer taps, the add-link Function URL records the tap and 302-redirects into the store's add-to-cart for the chosen SKU.

One channel, one message

The nudge goes out on one channel — an SMS via SNS in the walk-through here, though the same offer can just as well be an email through SES where that suits the shop better. Whichever it is, it's one message, and the message is checked by code before it leaves: it must be within the length limit, contain exactly one link (the add link we built, and no others), carry the small opt-out line that keeps the shop compliant, and name a price that matches the chosen SKU's catalogue price rather than anything the model might have typed. If a draft fails any check, or the model's line is missing, the sender falls back to a plain fixed template — "{name}, to go with your order: {add-on} for {price}. Add it in one tap: {link}. Reply STOP to opt out." — which is duller but always correct.

What the sender never does is send twice. The moment an offer is sent, its state flips to *sent* with a timestamp, so the next run of the scheduled sweep passes over it. One order produced one offer in Part 3; that offer produces exactly one message here. There is no "did you see our suggestion?" follow-up, no second reminder, no escalation of the ask. If the customer taps, Part 5 records it; if they don't, the offer simply expires when its attribution window closes. The whole design rests on a single, unrepeated, easy-to-accept nudge, and the send step is where that promise is kept.

DESIGN RULES THAT SHAPED THE SEND

- Wait a beat. A short send-at delay keeps the nudge off the checkout screen and out of the way of the order confirmation.
- A sweep, not a timer per order. A scheduled sender releases due offers, which is cheaper and gives a place to re-check the guardrails.
- Re-check at send time. Opt-out and the frequency cap are read again just before sending, so a late opt-out still stops the message.
- Saying yes is one tap. A signed token on a Function URL records the tap and 302-redirects straight into add-to-cart.
- The link is the code's job. The destination is built from the chosen SKU, never written by the model, so it can't point at the wrong thing.
- One message, never two. The offer flips to sent on the first message; there is no reminder and no second ask.

PART 5 OF 7

JULY 7, 2026 PART 5 OF 7 · UPSELL RECOMMENDER SERIES ~8 MIN READ

How a take-up gets tracked

An offer you can't measure is a guess you keep repeating. The point of the one-tap link isn't just convenience — it's the clean signal that tells the shop whether the suggestion landed. This post is about attribution: how a tap is tied back to its offer inside a set window, how the frequency cap is updated, and how a scheduled sweep closes the book on every nudge so the whole thing can be judged.

KEY TAKEAWAYS

- The one-tap link is also the measurement: the tap is the clean signal that an offer landed, recorded against that exact offer.
- A tap only counts as a take-up if it falls inside the attribution window — a set span after the nudge, a few days by default.
- Sending a nudge stamps the customer's frequency cap, so the next order is gated on when they were last nudged, not just whether.
- A scheduled sweep closes the window on offers with no tap, so every offer ends up counted as taken, ignored, or never sent.
- Because every outcome is recorded, the shop can see the real take-up rate — and prune add-ons that never land.

The tap is the signal

An offer you can't measure is a guess you keep repeating, so the point of the one-tap link isn't only convenience — it's the cleanest possible attribution signal. When the customer taps, the add-link redirect from Part 4 does its two jobs in order: it records the tap against the specific offer before it redirects, and only then sends them on to the store's add-to-cart. Because the token carries the offer id, there is no guessing about which nudge earned the click; the tap is tied to the exact offer, the exact SKU, and the exact customer. That is a far stronger signal than "this customer bought lights sometime this week", which could have happened for a dozen reasons.

The system deliberately measures the *tap*, not the eventual purchase. Whether the customer completes checkout after the item lands in their basket is up to them and the store, and chasing an abandoned basket is a different system with different manners. What the recommender is responsible for is whether its suggestion was good enough to act on, and the tap answers exactly that. Keeping the measured event narrow — "did the one-tap add link get tapped" — keeps the attribution honest and the system's scope clean.

The attribution window

A tap only counts as a take-up if it happens inside the **attribution window** — a set span after the nudge goes out, three days by default. A tap the same evening is plainly a response to the offer; a tap three weeks later, on a link the customer stumbled back onto, is not really the recommender's doing and shouldn't flatter its numbers. The window draws that line. Each offer carries its own window as a TTL-style expiry: taps before it counts as a take-up and stamps the offer *taken*; taps after it are still honoured — the item still lands in the basket, because it would be rude to break the link — but they're recorded as out-of-window and don't count toward the take-up rate.

Sending the nudge does one more thing: it stamps the customer's frequency cap. The moment an offer is sent, the customer's last-nudged timestamp and count are updated in the customers table — the very fields Part 2's gate reads. This is what makes the cap real rather than notional: the next order that customer places is checked against *when* they were last nudged, and if that's inside the window, the whole pick is skipped before the model ever runs. The take-up tracking and

the anti-nagging cap are two views of the same small piece of per-customer state, which is why they live together.

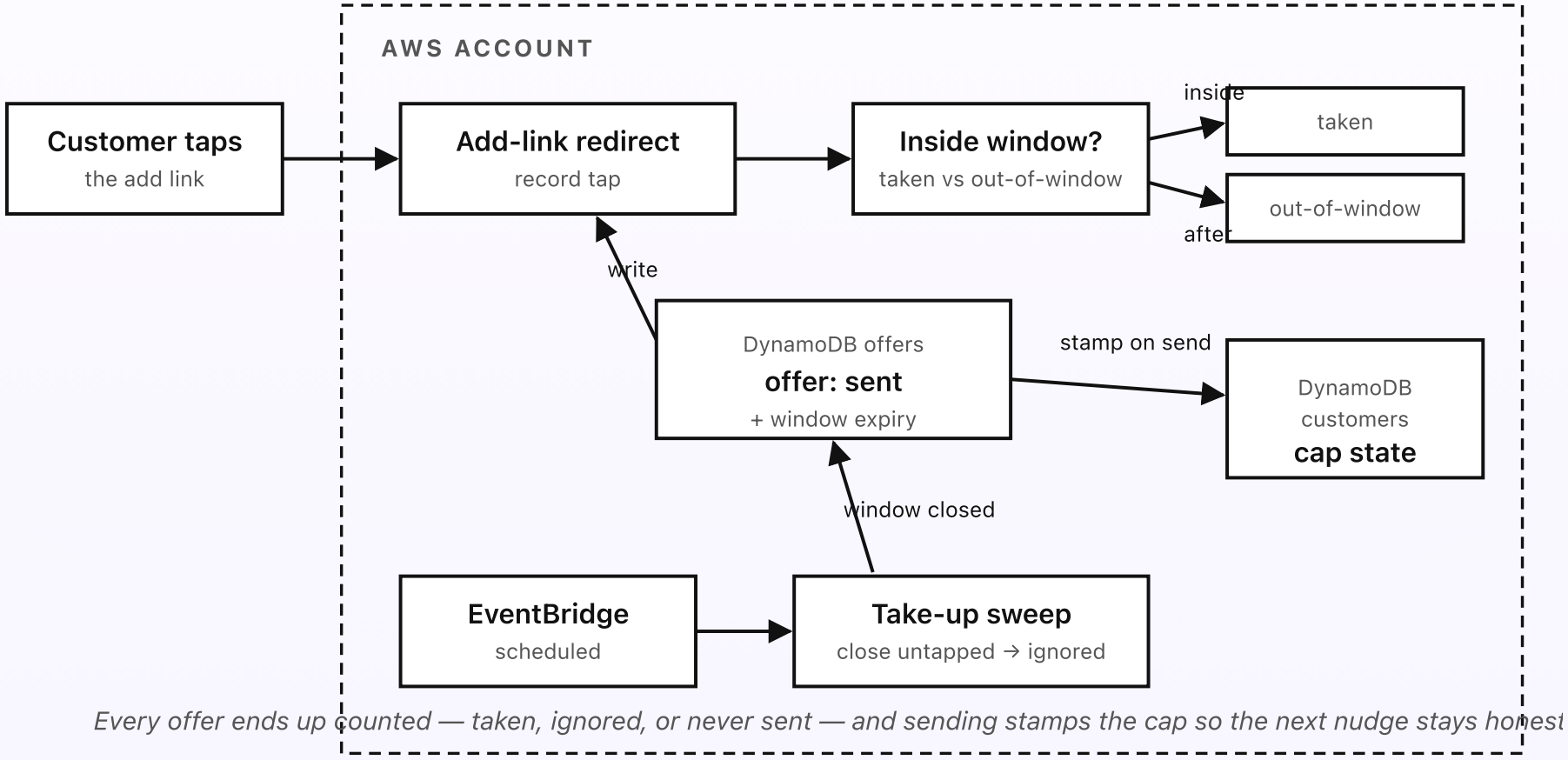

AWS ACCOUNT
Customer taps
the add link
Add-link redirect
record tap
Inside window?
taken vs out-of-window
inside
taken
after
out-of-window
write
DynamoDB offers
offer: sent
+ window expiry
stamp on send
DynamoDB
customers
cap state
window closed
EventBridge
scheduled
Take-up sweep
close untapped → ignored
Every offer ends up counted — taken, ignored, or never sent — and sending stamps the cap so the next nudge stays honest

Fig 5. Tracking take-up. A tap is recorded against its offer and counts as a take-up only inside the attribution window; sending the nudge stamps the customer's frequency cap; and a scheduled sweep marks untapped offers ignored once their window closes — so every offer ends up counted.

Closing the book on every offer

Most offers are never tapped — that's normal, and it's a number the shop needs to know. But an untapped offer produces no event, and a system that only reacts to taps would leave those offers hanging in the *sent* state forever, uncounted. So a scheduled sweep, driven by EventBridge Scheduler, runs on a fixed cadence and does the one thing the tap can't: it notices absence. It queries the offers table for anything still in *sent* whose attribution window has closed with no tap, and marks it *ignored*. Each offer is closed exactly once; the sweep is idempotent because an offer already marked taken, ignored, or out-of-window is skipped. Between the tap path and the sweep, every single offer reaches a terminal state — taken, ignored, or, for orders that never got one, no-fit.

That completeness is the whole payoff. Because no offer is left dangling, the shop can read a true take-up rate: of the orders that produced an offer, what fraction were tapped, broken down by add-on and by the item that triggered them. That turns the recommender from a black box into something a shopkeeper can actually manage — the descaler that goes with the coffee subscription lands four times out of ten, so keep it; the matching blanket the model keeps proposing for the dog bed lands almost never, so tag it out of the complementary set and let the rules stop offering it. The system suggests; the numbers teach; and because every outcome including "no offer" is recorded, the teaching is honest.

Why a window and a cap, not a memory

It would be tempting to reach for something cleverer — a model that "remembers" each customer and decides case by case how often to nudge them. That's the wrong tool for a job that's really just arithmetic. The attribution window is a fixed span; the frequency cap is a count and a timestamp. Both are trivially auditable, trivially explainable to a shop owner ("we won't message the same person more than once a month, and a tap only counts if it's within three days"), and impossible to drift. A model can't accidentally decide a good customer is fair game for a nudge a day, because the model is never asked; the cap is deterministic Python reading two fields. Keeping measurement and pacing as plain state — not judgement — is what lets the shop trust the recommender to run unattended without ever turning into the thing everyone dreads, a store that won't stop messaging you.

DESIGN RULES THAT SHAPED TRACKING

- The link is the measurement. The tap is recorded against its exact offer before the redirect — clean, unambiguous attribution.
- Measure the tap, not the purchase. Whether checkout completes is the store's business; the recommender owns whether the offer landed.
- A window draws the line. A tap only counts inside the attribution span; later taps are honoured but not counted.
- Sending stamps the cap. The last-nudged time and count are written on send, so the next order is paced, not just permitted.
- Close every offer. A sweep marks untapped offers ignored once their window shuts, so nothing is left uncounted.
- Pacing is arithmetic, not judgement. A fixed window and a count keep the whole thing auditable and impossible to drift.

PART 6 OF 7

JULY 7, 2026 PART 6 OF 7 · UPSELL RECOMMENDER SERIES ~6 MIN READ

What the upsell recommender costs

A recommender that costs more than the add-ons it sells is a toy. This post is the cost breakdown: every AWS service this design touches, what each adds up to at around 150 orders a month, and why the total lands near $2.60 — with the one model call per order the standout line — plus what happens to the bill when the volume goes up tenfold.

KEY TAKEAWAYS

- About $2.60/month at roughly 150 orders, and the fixed cost is almost nothing — nothing runs when no orders are coming in.
- The biggest usage line here is Bedrock: one propose-and-phrase call per order, which reads the basket and a candidate list every time.
- The next line is the outbound nudge; everything else — Lambda, queue, tables, schedules — is cents at this scale.
- The only real fixed cost is Secrets Manager: two secrets at $0.40 each, billed whether or not an order comes in.
- At ten times the volume (around 1,500 orders) the bill lands near $17 — it scales with use, not with idle time.

Where the money goes

The system is serverless end to end, so there's no instance ticking over between orders and no idle bill. You pay for an order only when one happens — and, for the two lines that reach the customer, only when an order actually earns an offer. At a typical small-shop volume — call it 150 orders a month, most of which produce an offer, with a handful filtered out as no-fit — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two secrets — webhook signing key, store API key ($0.40 each)	$0.80
Bedrock (Claude Haiku 4.5)	One propose-and-phrase call per order (~150), reading basket + candidates	$0.70
SNS (SMS)	One nudge per order that gets an offer (~120)	$0.50
DynamoDB (on-demand)	Orders, offers, catalogue mirror, customers, opt-out — small reads and writes	$0.25
CloudWatch Logs	Function logs, 7-day retention	$0.15
Lambda (Python 3.14, arm64)	Order, picker, sender, add-link, sweep, catalogue-sync	$0.10
SES	Internal take-up report to the shop owner	$0.05

AWS service	What it does here	Monthly
SQS + DLQ	Buffering between the webhook and the slower model and send calls	$0.05
EventBridge Scheduler	The send, the take-up sweep, and the catalogue sync	$0.00
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~150 orders/month**	**$2.60**

The shape of that bill is the point, and it's a little different from the other systems in this family. The only line that costs money while the system sleeps is Secrets Manager — two secrets at $0.40 each, $0.80 a month no matter what. But the biggest line that *moves* is Bedrock, and that's deliberate: this is the most model-heavy design of the lot, because a full recommendation runs on every eligible order, and each call reads the basket and a slice of the catalogue rather than just writing one short sentence. The nudge itself is the next line down. Everything else — the six Lambdas, the queue, the five tables, the three schedules — together costs less than the Bedrock line alone.

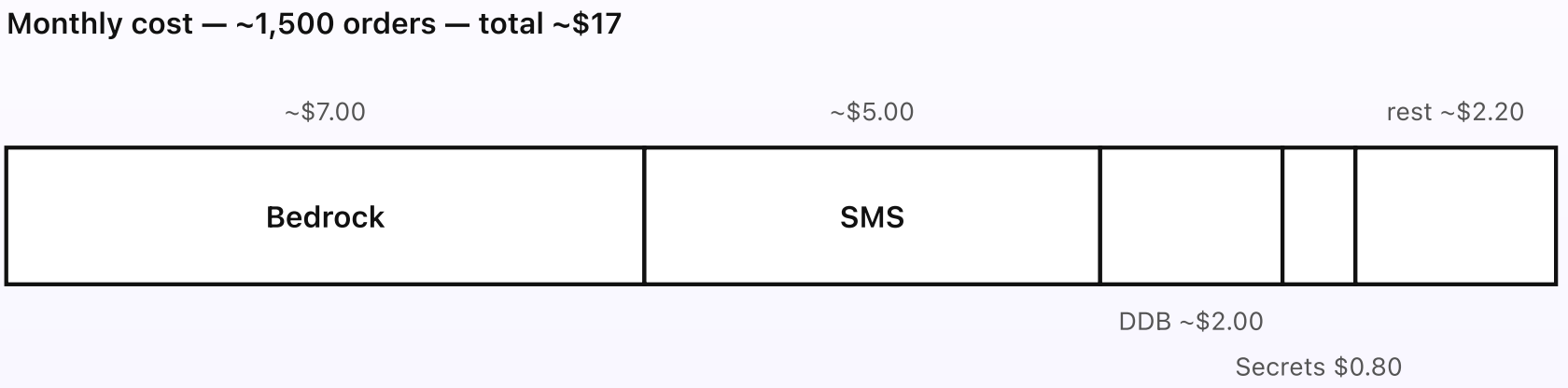
The line that isn't purely AWS

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 $0.50 here is a UK-leaning estimate for around 120 outbound nudges; your real number will track your country and your provider. If you send the offer by email through SES instead of SMS — a perfectly good choice for many shops — this line all but vanishes and the total drops below $2.20. Either way, it's one of only two lines that scale with volume, which is why the AWS Budgets alarm sits on top of the whole thing.

What ten times the volume costs

Push this to a busy shop — 1,500 orders a month, ten times the volume — and the bill lands near $17, not $26. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules stay at a cent, and AWS Budgets stays free. What scales is the genuinely usage-priced work, and here the model call leads it: roughly $7 of Bedrock for ten times the picks, about $5 of SMS, around $2 of DynamoDB, and a couple of dollars more across logs, SES, Lambda and the queue. Even then, the propose-and-phrase call and the nudge dominate, and all the machinery in between stays close to free.



Fixed lines don’t move with volume; here the model call is the largest line that scales, so the bill still grows sub-linearly.

Fig 6. The monthly bill at ten times the base volume, about 1,500 orders. Bedrock and SMS are the bulk of it; the fixed lines stay put, so the total grows sub-linearly — near $17, not ten times $2.60.

The honest way to read this: the AWS bill is rounding error against what an add-on sale is worth. A single lights-and-lock bundle, a burr grinder, a matching blanket — each is worth far more than $2.60, and this design catches them by the dozen without ever pestering the customer who doesn’t want one. Even at $17 a month for a busy shop, the system pays for itself the first few times it turns a bare order

into a one-tap add — and the customers it has nothing good to say to simply hear nothing at all.

DESIGN RULES THAT SHAPED THE COST

- Pay per order, not per hour. No always-on compute means no idle bill.
- The model is the big line — spend it once. One propose-and-phrase call per order, and only to rank and word — never to decide.
- Only offer when it fits. No-fit orders and capped customers skip the model and the nudge entirely, so the two big lines stay lean.
- Cheap work stays cheap. Catching, filtering, sending and sweeping are plain Lambda and DynamoDB, cents at this scale.
- Watch the two lines that scale. Bedrock and SMS are the parts that grow, so the Budgets alarm sits right on top of them.

PART 7 OF 7

JULY 7, 2026 PART 7 OF 7 · UPSELL RECOMMENDER SERIES ~10 MIN READ

Engineering reference: the upsell recommender architecture

This is the upsell recommender with the friendly labels removed: the real resource names, the runtime, the table key schemas, the two public Function URLs, the scheduled send and sweep, and the IAM scope that keeps the one model call in its box. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Six Lambda functions, all Python 3.14 on arm64, wired through one SQS queue with a dead-letter queue.
- Two public surfaces: a Function URL on `upsr-order` for the order webhook, and one on `upsr-addlink` for the one-tap redirect — no API Gateway.
- Five DynamoDB tables, all on-demand: orders, offers, a catalogue mirror, customers, and an opt-out list.
- One EventBridge Scheduler with three rules: the delayed send, the take-up sweep, and the catalogue sync.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by `upsr-picker`. 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 two Lambda Function URLs, the outbound nudge goes through SNS, and work is buffered on a single SQS queue.

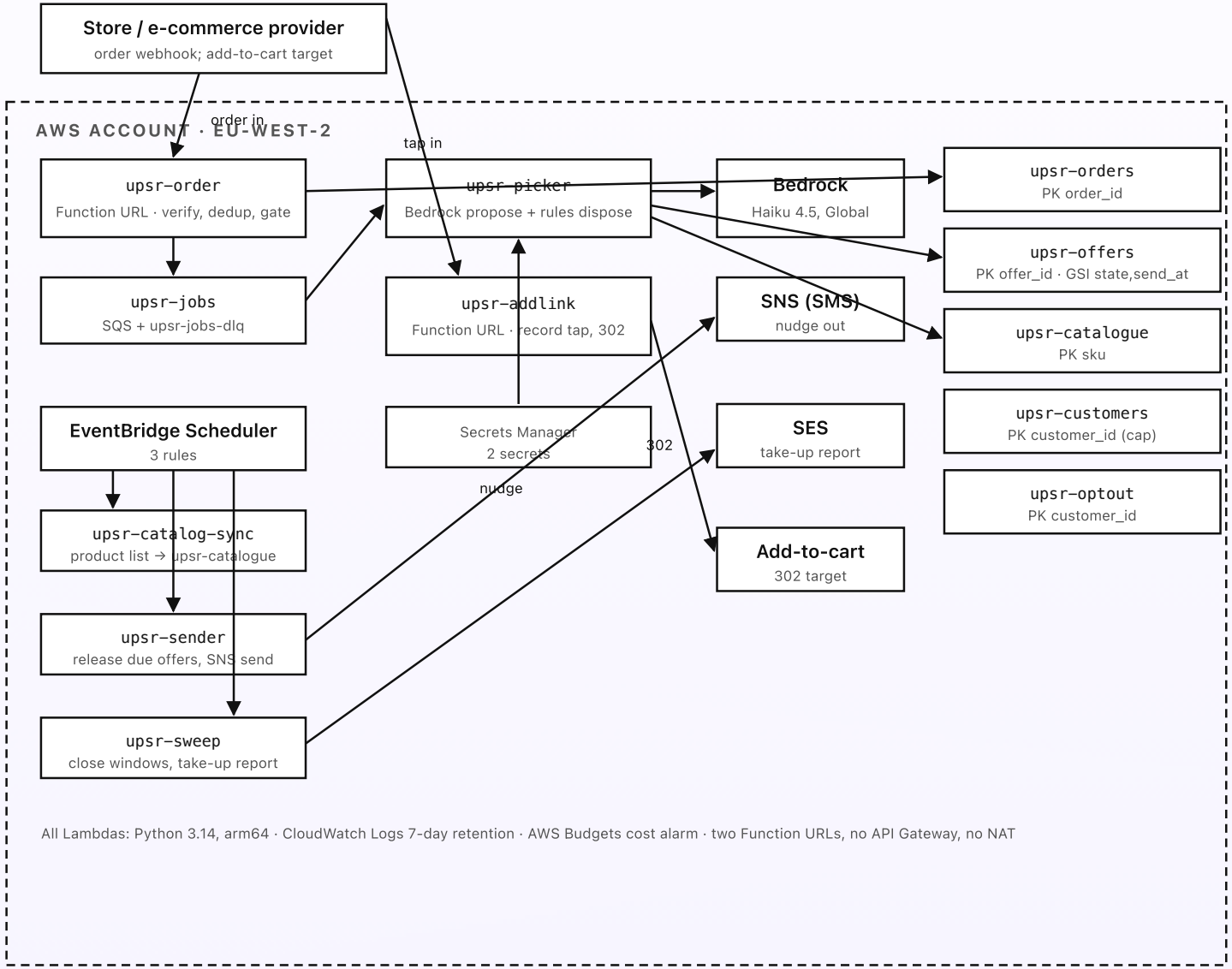

Store / e-commerce provider
order webhook; add-to-cart target
order in
AWS ACCOUNT · EU-WEST-2
tap in
upsr-order
Function URL · verify, dedup, gate
upsr-picker
Bedrock propose + rules dispose
Bedrock
Haiku 4.5, Global
upsr-orders
PK order_id
upsr-offers
PK offer_id · GSI state,send_at
upsr-catalogue
PK sku
upsr-customers
PK customer_id (cap)
upsr-optout
PK customer_id
upsr-jobs
SQS + upsr-jobs-dlq
upsr-addlink
Function URL · record tap, 302
SNS (SMS)
nudge out
EventBridge Scheduler
3 rules
Secrets Manager
2 secrets
302
nudge
SES
take-up report
upsr-catalog-sync
product list → upsr-catalogue
Add-to-cart
302 target
upsr-sender
release due offers, SNS send
upsr-sweep
close windows, take-up report
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm · two Function URLs, no API Gateway, no NAT

Fig 7. The upsell recommender drawn for engineers: two Function URLs (`upsr-order` and `upsr-addlink`), an SQS-buffered set of six Lambdas, five DynamoDB tables, Bedrock called only by the picker, SNS for the nudge and SES for the take-up report, and three scheduled jobs. One region, one account, no API Gateway.

Lambda functions

Six functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. Each does one job and hands off; the SQS queue (`upsr-jobs`, with `upsr-jobs-dlq` as its dead-letter queue after five attempts) decouples the public webhook from the slower model call.

- `upsr-order` — a public surface. Backs the order Function URL: verifies the store's signature, de-duplicates against `upsr-orders` with a conditional write on the order id, checks `upsr-optout` and the frequency cap on `upsr-customers`, gathers the basket and history, and enqueues a suggest job. Nothing slow happens here — no catalogue join, no model call.
- `upsr-picker` — SQS-triggered on suggest jobs. Makes the single Bedrock call to propose a ranked shortlist, applies the deterministic catalogue filters (in stock, complementary, price-appropriate, not already owned) against `upsr-catalogue`, and writes the top survivor with a `send_at` to `upsr-offers` — or writes a `no-fit` record and stops. The only function with `bedrock:InvokeModel`.
- `upsr-sender` — scheduled. Queries `upsr-offers` by the state-and-send-at index for offers whose delay has elapsed and are still `chosen`, re-checks opt-out and the cap, builds the signed one-tap link, sends one nudge via SNS,

stamps the frequency cap on `upsr-customers`, and flips the offer to `sent` with a conditional update so it is never sent twice.

- `upsr-addlink` — a public surface. Backs the add-link Function URL: validates the signed token, records the tap against the offer in `upsr-offers` (marking it `taken` if inside the attribution window, or `out-of-window` if not), and issues an HTTP 302 to the store's add-to-cart URL for the chosen SKU.
- `upsr-sweep` — scheduled. Queries `upsr-offers` for `sent` offers past their attribution window with no tap, marks each `ignored`, and emails the shop owner a rolled-up take-up report via SES.
- `upsr-catalog-sync` — scheduled. Pulls the product list — stock, prices, categories and complementary tags — and upserts rows into `upsr-catalogue`, so the picker's filters always read current stock.

Idempotency and exactly-once

Three points in the flow must not double-act, and each is pinned by a conditional write rather than by hope. First, **one offer per order**: `upsr-order` writes `upsr-orders` with a condition that the order id doesn't already exist, so a store that fires the webhook three times for one checkout creates exactly one job. Second, **one nudge per offer**: `upsr-sender` flips the offer from `chosen` to `sent` with a condition on the current state, so two overlapping scheduler runs can never both send — the loser's conditional update simply fails. Third, **one terminal state per offer**: both `upsr-addlink` and `upsr-sweep` only transition an offer that is still `sent`, so a tap that races the sweep resolves to a single outcome, and a re-run of the sweep skips anything already closed. The offers table's state field is the single source of truth for where each offer is, and every transition is a guarded write.

Data stores, schedules, and messaging

- **DynamoDB (all on-demand).** `upsr-orders` — PK `order_id`; one item per order with its dedup marker, basket snapshot, and gate outcome. `upsr-offers` — PK `offer_id`, holding the order, customer, chosen SKU, line, `send_at`, and state (`chosen` / `sent` / `taken` / `out-of-window` / `ignored` / `no-fit`); a GSI on `state` + `send_at` drives the sender and sweep queries, and the attribution window is a TTL attribute. `upsr-catalogue` — PK `sku`, the product mirror with stock, price, category and complementary tags. `upsr-customers` — PK `customer_id`, the history plus the frequency-cap fields (`last_nudged_at`, `nudge_count`). `upsr-optout` — PK `customer_id`, the suppression list checked before every send.
- **Function URLs.** Two — `upsr-order` for the inbound webhook and `upsr-addlink` for the one-tap redirect, both `AuthType NONE` at the edge because authenticity is enforced in-function: the webhook by the store's shared-secret signature, the add link by the signed token it carries. No API Gateway.
- **SNS and SES.** SNS sends the outbound nudge (or SES does, if you send the offer by email); SES sends the internal take-up report to the shop owner from a verified domain with DKIM.
- **EventBridge Scheduler.** Three rules — `upsr-sender` at `rate(5 minutes)` to release due offers, `upsr-sweep` at `rate(1 hour)` to close windows, and `upsr-catalog-sync` at `rate(15 minutes)` to refresh stock and prices.
- **Secrets Manager.** Two secrets — the webhook signing secret and the store API key — fetched at call time, never in env vars or the catalogue sheet. The add-link token signing key lives here too, alongside the webhook secret.

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

IAM scope, observability, and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `upsr-order` can read `upsr-customers` and `upsr-optout`, conditionally write `upsr-orders`, read the signing secret, and send to `upsr-jobs` — it cannot call Bedrock, SNS, or the catalogue. `upsr-picker` is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile; it reads `upsr-catalogue` and writes `upsr-offers`, and cannot send anything. `upsr-sender` can read `upsr-offers`, `upsr-optout` and `upsr-customers`, publish to SNS, and update the offer and cap — but cannot call Bedrock. `upsr-addlink` can validate the token, update `upsr-offers`, and redirect, nothing more. The scheduled functions hold the narrow catalogue, offers and SES permissions they need and, apart from `upsr-addlink`, no inbound surface at all. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call for capacity and is not a data store. CloudWatch Logs at 7-day retention carry every function's trace, and an AWS Budgets alarm watches the monthly spend — with Bedrock and SMS the lines most likely to move, it's the cheapest early warning that volume (or a runaway loop) is running hot.

That's the whole system: an order comes in, a model proposes and the catalogue rules dispose, one tasteful nudge goes out with a one-tap link, and every offer — taken, ignored, or never sent — ends up counted. No always-on compute, one small model call per order, and a hard filter that means a bad suggestion can

never reach a customer. It sells the add-on that actually fits, and stays quiet when none does.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Six small Lambdas beat one that does everything; the queue decouples the model call from the webhook.
- Two public surfaces, both self-authenticating. `upsr-order` by signature, `upsr-addlink` by signed token — no API Gateway.
- Least privilege, per role. Only the picker can call Bedrock; only the order and sender functions touch the opt-out list.
- State in DynamoDB, transitions guarded. The offers table's state field drives the send and sweep, and every change is a conditional write.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called once per order.
- A budget alarm is a smoke detector. With Bedrock and SMS the variable lines, a Budgets alert is the cheapest way to catch a runaway.