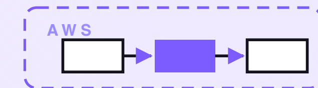
7-PART SERIES · FREE COMPANION



Referral tracker

A happy customer who'd recommend you is the cheapest growth a small business has — and most shops have no way to turn that goodwill into anything they can actually track. This is the design of a small serverless system that gives every customer their own referral link on request: it mints a unique code, writes a shareable invite in the business's voice, logs the click behind a redirect, attributes a sign-up to the referrer on a last-touch window, and — when the referred person actually converts — credits both sides exactly once and sends a thank-you. It blocks the obvious abuse: self-referrals, the same person converting twice, and codes sprayed into the void. 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/referral-tracker

CONTENTS

Referral tracker

01 A referral tracker on AWS for a few dollars a month

02 How a referral link gets minted

03 How a referral gets attributed

04 How a reward gets issued

05 How referral fraud gets caught

06 What the referral tracker costs

07 Engineering reference: the referral tracker architecture

PART 1 OF 7

JULY 4, 2026 PART 1 OF 7 · REFERRAL TRACKER SERIES ~10 MIN READ

A referral tracker on AWS for a few dollars a month

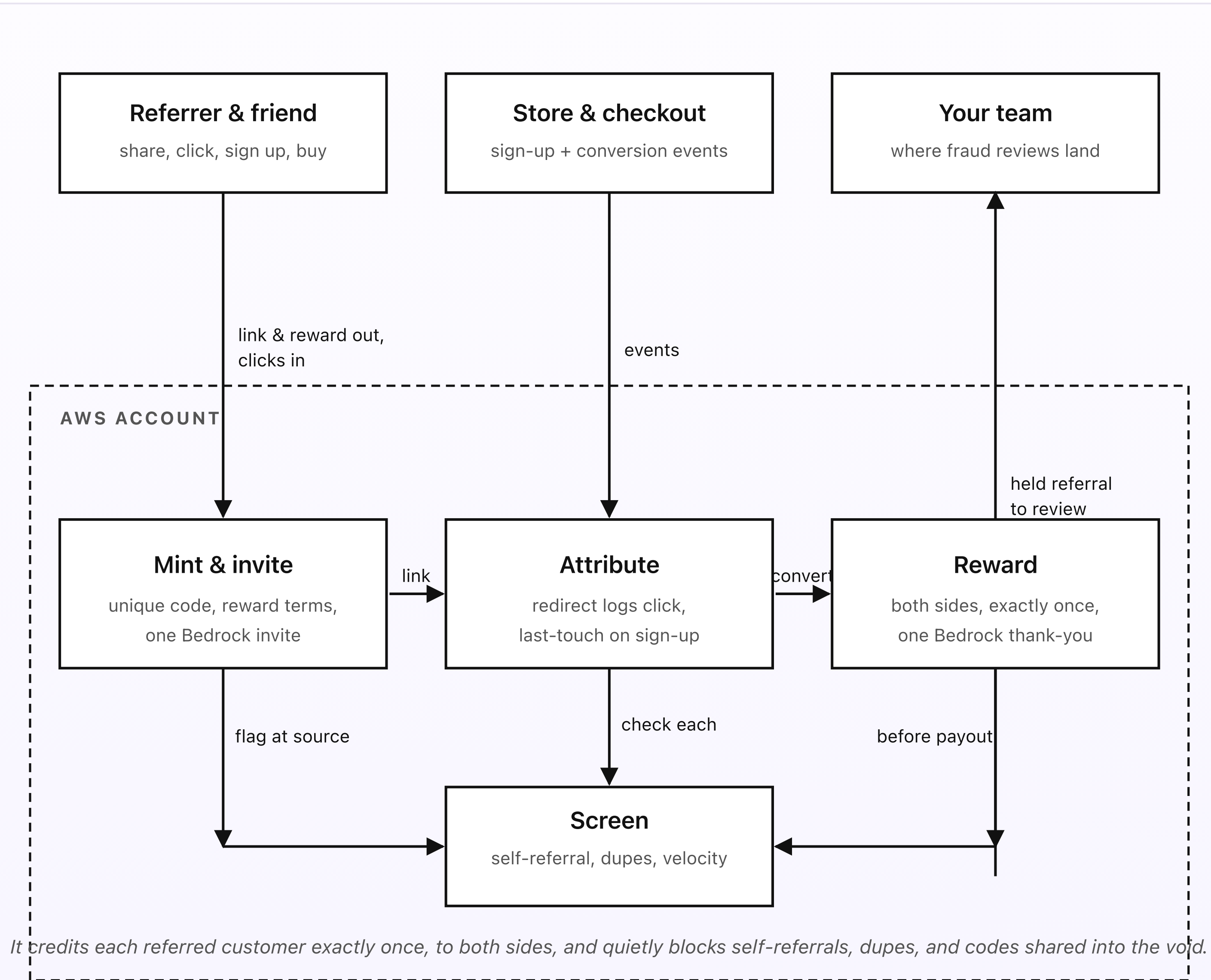
Word of mouth is the cheapest customer a small business ever gets, and almost nobody tracks it. This post walks through the design of a small serverless system that gives every customer their own referral link, follows the friend they send from click to sign-up to first purchase, and — only then — credits both sides exactly once, with the self-referrals and dupes quietly filtered out.

KEY TAKEAWAYS

- A customer asks for a referral link; the system mints a unique code and one Bedrock call writes a shareable invite in your voice.
- A tap on the link is logged behind a fast redirect, so word of mouth becomes something you can actually see and credit.
- A sign-up is attributed to the referrer on a last-touch window; the reward stays pending until the friend genuinely converts.
- On conversion, both sides are credited exactly once and thanked — self-referrals, dupes, and empty codes are filtered out first.
- Designed on AWS for about $1.70/month at roughly 200 clicks and 20 conversions. Every decision is plain Python; the model only writes words.

The whole system on one page

Before any code, here's the shape of what we're designing. Nearly every small business has customers who'd happily recommend them — the meal-prep subscriber who won't shut up about it at work, the regular at the barber's, the neighbour who watched the window cleaner do next door. The trouble is that goodwill vanishes the moment it's spoken: there's no link, no record, no way to say thank you to the person who actually sent someone. The system below catches that moment and makes it trackable. A customer asks for their link, a friend they send taps it, and if that friend becomes a real customer, both of them are rewarded — automatically, and only once.



It credits each referred customer exactly once, to both sides, and quietly blocks self-referrals, dupes, and codes shared into the void.

Fig 1. Three things outside, four pieces inside AWS. A customer requests a link; Mint & invite issues it, Attribute logs the click and credits the last referrer on sign-up, and Reward credits both sides once the friend converts. Everything a payout depends on passes through Screen first.

What you set up once (the outside)

- **Your store and checkout.** Wherever a new customer signs up and makes their first purchase — a subscription app, a booking page, a shop. It needs to do three things: send a sign-up event when someone creates an account (carrying whatever referral token they arrived with), fire a conversion webhook when that account makes its first real purchase, and accept a reward back — a credit, a discount code, a free box — so the system can fulfil what it promises. You point these at one AWS URL and store the signing key in Secrets Manager. This is the trigger for attribution and reward, and it's covered in Parts 3 and 4.
- **The reward terms and voice.** A small settings doc: what each side gets (say £10 credit for the referrer and £10 off the friend's first box), how long the last-touch window runs, the daily velocity limits, and the business's voice for the invite and thank-you copy. This is where you decide the shape of the offer; the system just enforces it. The reward catalogue — the actual codes or credits it hands out — lives with the store.
- **Your team.** The person who looks at anything the system deliberately won't pay on its own — a referral held by the fraud screen, usually because it looks like a self-referral or a code behaving like a bot. They get an email with the referrer, the referred customer, and exactly why it was held, and they approve or reject it. The system never pays a held referral automatically; a human makes that call.

What runs on every referral (the inside)

- **Mint & invite.** A customer asks for their link — a tap in the app, a click on the account page. One Lambda mints a short, unique, collision-proof code, records who it belongs to and the reward terms in force, and makes a single Bedrock Haiku 4.5 call to write a shareable invite in the business's voice. The link goes back to the customer to share. This is Part 2.
- **Attribute.** When a friend taps the link, the request hits a Lambda Function URL that logs the click and redirects them straight to the sign-up page in a few milliseconds. If that friend signs up, the system credits the most recent referrer inside the last-touch window and opens a *pending* referral. A click and a sign-up on their own pay nothing. This is Part 3.
- **Reward.** When the referred person makes a genuine first purchase, the checkout fires a conversion webhook. The system confirms the referral is pending and eligible, credits both sides exactly once with a conditional write, fulfils the reward through the store, and makes one Bedrock call to write two short thank-you notes. This is Part 4.
- **Screen.** The fraud lane that everything a payout depends on runs through: is the referred person really the referrer wearing a different hat? Has this friend already been rewarded once? Is one code suddenly spraying clicks and sign-ups like a script? Self-referrals and dupes are blocked outright; anything merely suspicious is held for a person. This is Part 5.

In plain words

Dan has had a FreshBox meal-prep subscription for a few months and keeps recommending it. He taps "refer a friend" in the app; a second later he has a link

— `refer.freshbox.uk/r/7QK2A` — and a ready-written message: “I’ve been getting FreshBox for a while and it’s genuinely saved my weeknights. Here’s £10 off your first box if you fancy trying it.” He sends it to his friend Mara. On Tuesday she taps the link out of curiosity; the redirect logs the click and drops her on the sign-up page. On Thursday she comes back and signs up — the system looks over the last-touch window, sees Dan’s was the most recent link she touched, and opens a pending referral crediting him. Nothing is paid yet. The following week her first box ships and the checkout fires a conversion. Now both sides are credited exactly once: Dan gets £10 off next month, Mara’s £10 welcome discount is confirmed, and each gets a warm thank-you. Nobody at FreshBox touched a thing.

A week later a different account tries the same code from Dan’s own phone, on the same card he pays with, under a new email. That isn’t a referral — it’s Dan trying to refer himself for a second £10. The system matches the new sign-up’s payment fingerprint and device to Dan’s existing account, recognises a self-referral, and blocks it before a penny moves. It doesn’t email Dan an accusation; it simply declines to pay and records why, and if anything about the case is genuinely ambiguous it lands in front of a person rather than being guessed at. The honest referral pays instantly; the self-referral quietly doesn’t.

DESIGN RULES THAT SHAPED EVERY DECISION

- Pay on conversion, not on hope. A click or a sign-up earns nothing; only a genuine first purchase triggers a reward.
- Both sides, exactly once. Each referred customer credits the referrer and themselves a single time — a repeat webhook never double-pays.
- Last touch wins, inside a window. The most recent referrer within the window gets the credit; stale clicks expire and pay nobody.
- The model only writes words. The invite and the thank-you are phrased by Bedrock; every attribution and fraud decision is deterministic.
- Assume it'll be gamed. Self-referrals and dupes are blocked, and anything spraying like a bot is held for a human, not paid.
- The redirect stays instant. Logging a click never slows the link down; the visitor is on their way in milliseconds.

Why this shape

Most small businesses handle referrals one of three ways: they don't — goodwill evaporates unrecorded; they run an honesty-box "mention my name" scheme that nobody remembers to apply; or they buy a full referral SaaS with a monthly seat price that only makes sense at real scale. The first leaves money on the table every week. The second is untrackable and unfair — the customer who actually sent three people gets nothing. The third is a fixed bill for something a barber or a window cleaner uses a dozen times a month. The gap is a cheap, honest way to mint a link, follow it, and pay out fairly — and nothing in a typical small shop fills it.

The shape above fills exactly that gap and nothing more. It leans on the store you already run for the sign-up and conversion events, keeps the reward terms as a setting you control, and adds a small system that turns each customer's recommendation into a trackable link and a fair payout. The common case — a real friend who really subscribes — runs end to end with no human involved. The few that are dubious are pulled out before any money moves and put in front of a person, with the whole story attached.

The next four posts walk through each piece in turn: how a referral link gets minted, how a referral gets attributed, how a reward gets issued, and how referral fraud gets caught. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JULY 4, 2026 PART 2 OF 7 · REFERRAL TRACKER SERIES ~8 MIN READ

How a referral link gets minted

A referral programme is only as good as the link a customer can actually share. This post is about minting that link: how a request becomes a unique, collision-proof code tied to the right referrer, how one small model call turns it into an invite the customer is happy to send, and why every fact about the reward is pinned at mint time, not later.

KEY TAKEAWAYS

- Minting starts when an existing customer asks for a link — a tap in the app or on the account page — never on its own.
- The code is short, random, and collision-proof: a conditional write claims it, so two customers can never be handed the same one.
- The referrer, the reward terms, and the window are pinned onto the code at mint time, so later changes never move an in-flight referral's goalposts.
- One Bedrock call writes a shareable invite in the business's voice; the link itself is injected by code, never typed by the model.
- A customer who already has an active code is handed the same one back, so a nervous double-tap doesn't litter the table with duplicates.

A link worth sharing

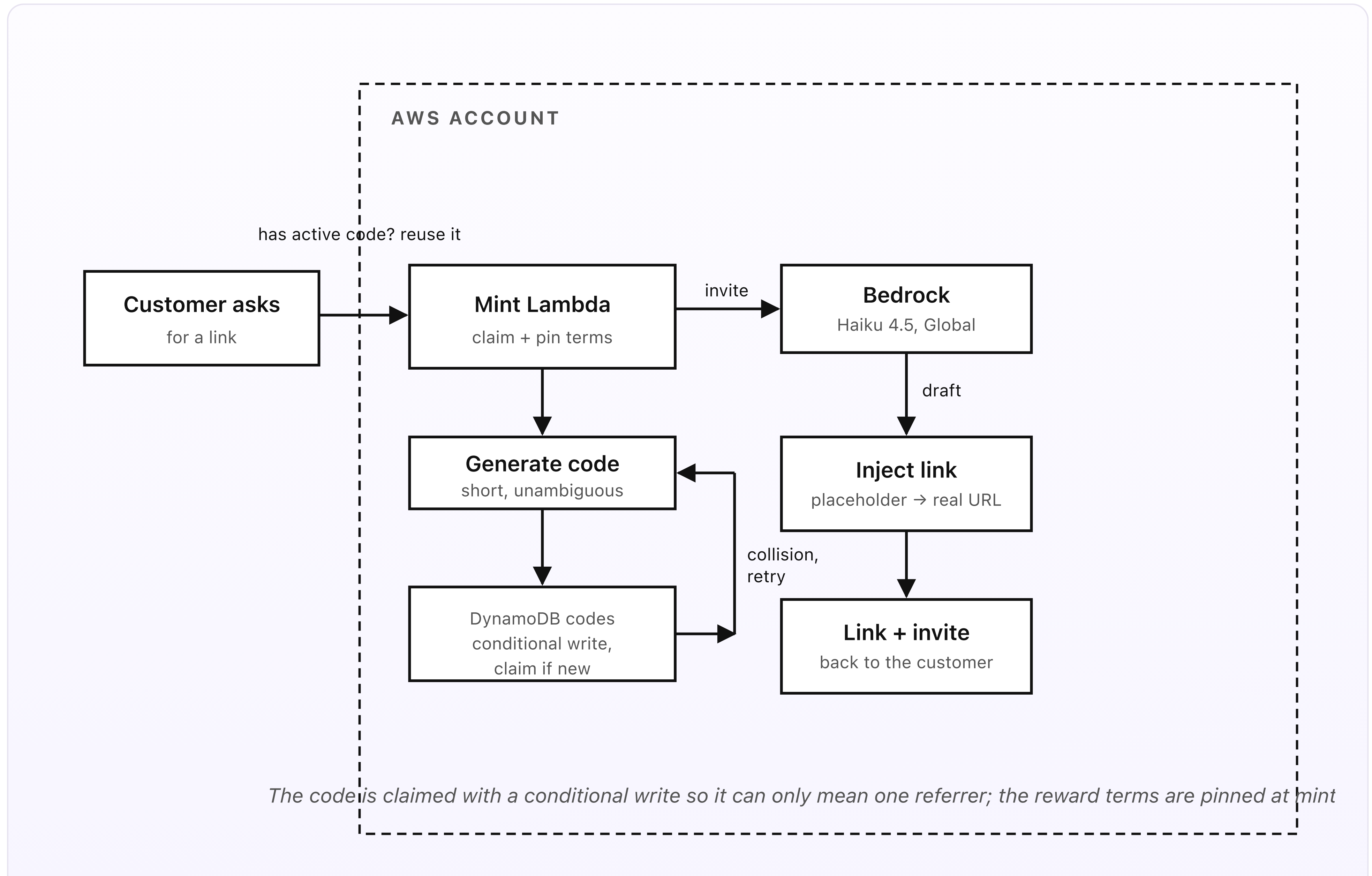
Everything downstream depends on one small artefact: a link a customer is actually willing to send. Get it wrong and the whole programme stalls before it starts — an ugly code nobody trusts, a message so corporate it makes the sender cringe, or a link that collides with someone else's and credits the wrong person. So minting gets its own step, and it does three things carefully: it makes a code that can only ever mean one referrer, it pins down exactly what reward that code promises, and it writes an invite the customer is happy to paste into a message to a friend.

Minting is always triggered by a real customer asking. A subscriber taps “refer a friend” in the FreshBox app; a regular scans the QR on the mirror at the barber’s; the window cleaner’s customer clicks a link in their receipt email. That request carries the customer’s identity from the store, which is how the system knows whose code this is. The system never mints a code nobody asked for, and it never invents a referrer — a code with no owner could credit anyone, which is exactly the hole a fraud programme can’t have.

A code that means one person

The code is short on purpose — five or six characters from an unambiguous alphabet, no easily-confused `0`/`O` or `1`/`l` — because it’s going to be read aloud, typed on a phone, and printed on a card in the window cleaner’s van. Short codes mean a small chance that two random draws collide, so the system doesn’t trust to luck: it generates a candidate and claims it with a conditional write to the codes table — write this item only if this code doesn’t already exist. If the claim fails, it draws another and tries again. The first writer wins; nobody is ever handed a code that already belongs to someone else.

At the same moment the code is claimed, the system pins the facts that make it mean something: which customer owns it, and the reward terms in force right now — what the referrer gets, what the friend gets, and how long the last-touch window runs. Pinning these at mint time is deliberate. If FreshBox later cuts the referrer reward from £10 to £5, a code Dan minted last week still pays the £10 it promised when he shared it; the offer a customer sent in good faith is the offer that’s honoured. The code carries its own terms so nothing that changes later can quietly move the goalposts on a referral already in flight.



The code is claimed with a conditional write so it can only mean one referrer; the reward terms are pinned at mint

Fig 2. Minting a link. An existing active code is handed straight back; otherwise a short candidate is drawn and claimed with a conditional write, retried on collision, then stamped with the referrer and the reward terms. One Bedrock call writes the invite, code injects the real URL, and the link goes back to the customer.

One code per customer, not per tap

Before it draws a fresh code, the mint function checks whether this customer already has an active one. Most do, after the first time — and a referral link is more useful the longer it stays stable, because a customer might paste it into a WhatsApp group today and a work Slack next month, and both should credit the same person. So a customer's code is durable: ask twice and you get the same link back, not a second one. This also quietly defends the table against the nervous double-tap and the impatient retry, which would otherwise scatter half a dozen live codes for one person and make attribution ambiguous later. One customer, one active code, handed back as many times as they ask.

There's one deliberate exception: if a customer's code has been retired — because it was implicated in abuse, say, or the customer left and came back — a fresh mint issues a new one and the old code simply stops attributing. Retirement is a state on the code, not a deletion, so the click and referral history tied to the old code stays intact for the record even though it can no longer earn.

The invite, in your voice

The last thing minting does is write the message the customer will actually send, and this is the one place a model runs at mint time. A single Bedrock Haiku 4.5 call

is handed a narrow brief: the business name and voice notes from the settings doc, the friend-side reward ("£10 off your first box"), and a placeholder where the link will go. It writes one short, warm invite that sounds like the business and doesn't oversell — the sort of thing a real person would be happy to paste under "you should try this". Haiku is right for the job precisely because it's small: a sentence or two, in a set tone, with nothing to reason about.

The link itself is never written by the model. Bedrock emits a token like `{{link}}`, and the code substitutes the real short URL — `refer.freshbox.uk/r/7QK2A` — afterwards. A language model is perfectly capable of subtly mistyping a URL or "tidying" it, and a broken referral link is the one mistake that quietly wastes the whole invite: the friend can't click, the click can't be logged, nobody gets attributed. By keeping the link out of the model's hands and injecting it deterministically, the link is always exactly right, and the draft is checked for length and a single link before it goes out. If the model is slow or the draft fails a check, a plain fixed template is sent instead — less charming, always correct. The model gets the warmth; the code keeps the guarantee.

DESIGN RULES THAT SHAPED MINTING

- Only on request, always owned. A code is minted when a real customer asks, and it always names exactly one referrer.
- Claim, don't hope. A conditional write claims each code so two customers can never be handed the same one.
- Pin the terms at mint. The reward and window travel on the code, so a later change never moves an in-flight referral's goalposts.
- One durable code per customer. Ask twice, get the same link — stable to share, and no duplicate codes to confuse attribution.
- The link is the code's job. The model emits a token; the real URL is injected afterwards so it can never be mangled.
- A template is always ready. If the model drifts or fails, a fixed correct invite goes out — the link is never held up by a bad draft.

PART 3 OF 7

JULY 4, 2026 PART 3 OF 7 · REFERRAL TRACKER SERIES ~8 MIN READ

How a referral gets attributed

Between a shared link and a paid reward sit three separate events: a click, a sign-up, and a conversion. This post is about tying them together honestly — how the redirect logs a click, how a sign-up is attributed to the last referrer inside the window, and why the reward stays *pending* until the friend actually buys.

KEY TAKEAWAYS

- A tap on the link hits a Lambda Function URL that logs the click and 302-redirects to the sign-up page in a few milliseconds.
- The click drops a short-lived attribution token whose lifetime is the last-touch window, so old clicks expire and stop counting.
- On sign-up, the most recent valid click wins — last touch — and the referrer named on that code is credited.
- Attribution opens a *pending* referral; a click and a sign-up on their own pay nothing until a real conversion follows.
- A sign-up with no click in the window, or a code that's been retired, is simply unattributed — nobody is credited by accident.

Three events, not one

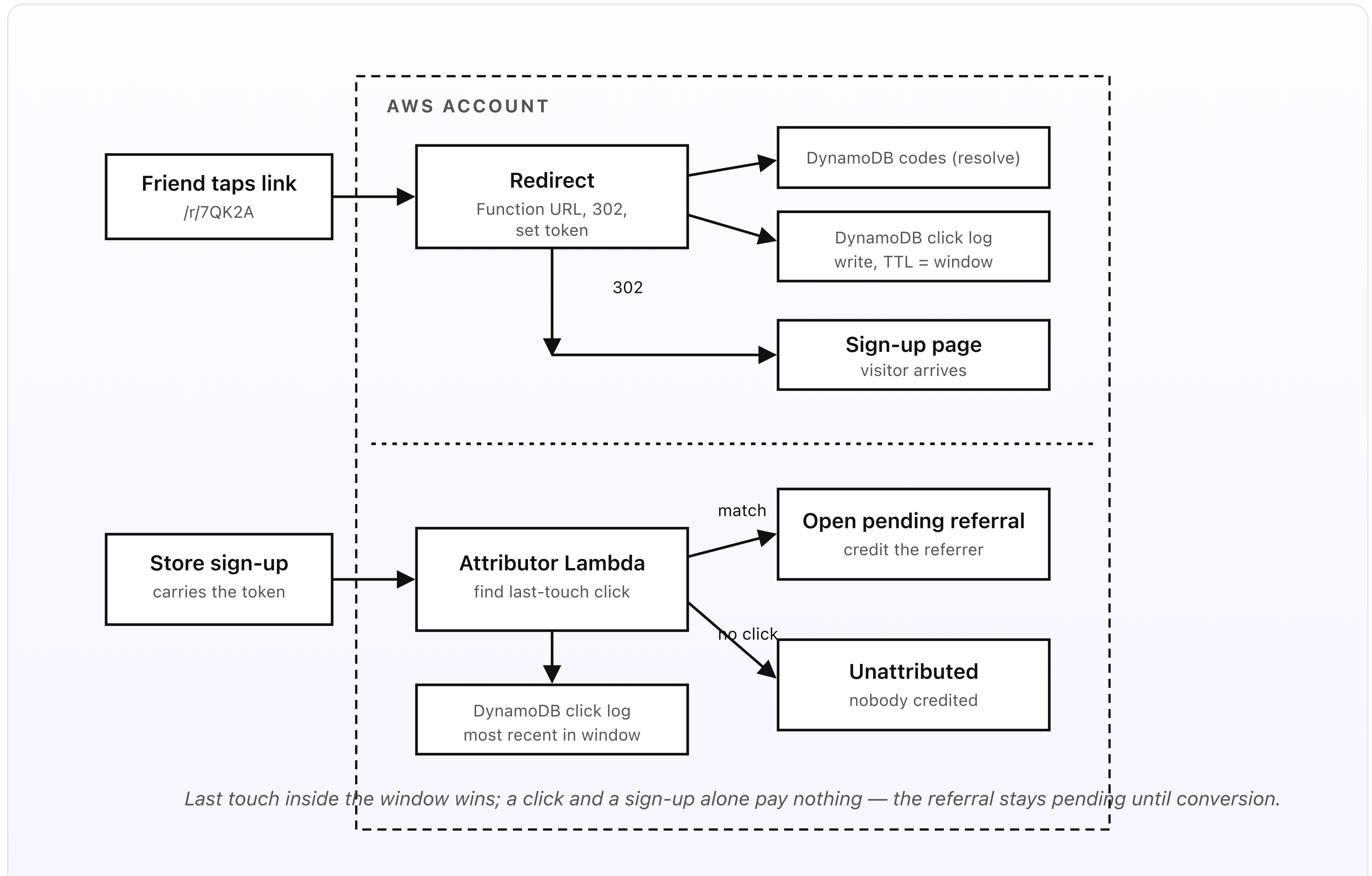
It's tempting to think of a referral as a single thing, but honestly attributing one means keeping three separate events apart and only joining them when each is real. First a **click**: the friend taps the shared link. Then a **sign-up**: they create an account. Then, later and by no means always, a **conversion**: they actually buy. Plenty of clicks never sign up, and plenty of sign-ups never buy. If you pay on the click you reward curiosity; if you pay on the sign-up you reward an email address. The reward belongs to the conversion — but to know *whom* to reward when it finally happens, you have to have quietly followed the trail from the very first tap. This post is about laying that trail down honestly.

The design keeps a firm line: attribution decides *who would be credited if this converts*, and nothing more. It opens a pending referral and waits. The money doesn't move here — that's Part 4 — but everything Part 4 needs to pay the right person, exactly once, is assembled here.

The redirect that logs a click

The shared link points at one Lambda Function URL — `refer.freshbox.uk/r/7QK2A` resolves to a function, not a web server. When a friend taps it, the function does the smallest possible amount of work and gets out of the way: it looks up the code, records a click, sets a short-lived attribution token, and returns a `302` redirect to the real sign-up page. All of that takes a few milliseconds, so the link behaves exactly like any other short link — tap, arrive. Nothing about logging the click is allowed to make the visitor wait; a slow referral link is one nobody clicks twice.

The attribution token is the quiet mechanism that ties a future sign-up back to this click. When the redirect fires, it sets a token — a first-party cookie on the store's domain, and the same value appended to the redirect URL as a fallback for when cookies are blocked — that says "this visitor last touched code `7QK2A`". Crucially, that token is given a lifetime equal to the last-touch window: 30 days, say. After that it expires on its own. The click itself is written to a click log with a matching TTL, so the record and the token die together, and a tap from two months ago can't reach forward and claim a sign-up today.



Last touch inside the window wins; a click and a sign-up alone pay nothing — the referral stays pending until conversion.

Fig 3. Attribution in two lanes. The redirect logs a click with a window-length TTL and sets an attribution token, then sends the visitor on. When the store reports a sign-up carrying that token, the attributor finds the most recent valid click and opens a pending referral for that referrer — or leaves it unattributed if nothing valid is in the window.

Last touch, inside the window

When the store reports a sign-up, it carries whatever attribution token the visitor arrived with, and the attributor uses it to answer one question: which referrer, if any, gets the credit? The rule is **last touch** — the most recent valid click within the window wins. If Mara tapped a link from Dan a fortnight ago and another from her colleague Sam yesterday, Sam gets the credit, because his was the touch that actually brought her back to sign up. Last-touch is the honest default for word of mouth: it rewards the nudge that landed, not the one that happened to be first, and it's simple enough to explain to a customer who asks why they were or weren't credited.

"Valid" is doing real work in that sentence. A click only counts if it's inside the window — the TTL guarantees an expired one isn't even in the table — and if the code it points at is still live. A sign-up that arrives with a token for a retired code, or with no token at all, or whose only clicks fell outside the window, is simply *unattributed*: the account is created as normal and nobody is credited. That's the right outcome, not a failure. Crediting nobody is always safer than crediting the wrong person, and a referral programme that occasionally misses a genuine referral is far less damaging than one that pays out on ghosts.

Pending, and waiting

A successful attribution doesn’t pay anyone; it opens a *pending* referral. That record ties three things together — the referrer from the code, the newly signed-up friend, and the reward terms pinned back at mint time — and it sits in the pending state until one of two things happens. Either the friend converts, and Part 4 credits both sides exactly once; or they never do, and the referral quietly expires when its own window elapses (the expiry sweep in the engineering reference does the tidying). Keeping the reward pending is what makes the whole programme honest: the click showed interest and the sign-up showed intent, but only the conversion shows a real customer, and only a real customer is worth a reward. Everything up to here has just made sure that when the conversion arrives, the system already knows exactly whose reward it is.

DESIGN RULES THAT SHAPED ATTRIBUTION

- The redirect is sacred. Log the click, set the token, and 302 in milliseconds — the link must never feel slow.
- The token lives for the window. Click records and tokens share a TTL, so an expired click can't reach forward and claim a sign-up.
- Last touch wins. The most recent valid click in the window is credited — the nudge that actually landed, not the first one.
- Credit nobody before the wrong body. No click, no token, or a retired code means unattributed — never a guessed referrer.
- Attribution opens, it doesn't pay. A sign-up only ever creates a pending referral; the money waits for a real conversion.

PART 4 OF 7

JULY 4, 2026 PART 4 OF 7 · REFERRAL TRACKER SERIES ~8 MIN READ

How a reward gets issued

The reward is the whole point, and the one place a mistake costs real money — a double payout, or a reward for a purchase that never happened. This post is about issuing it safely: how a conversion is confirmed, how both sides are credited exactly once even if the webhook fires twice, and how the thank-you gets written without letting a model near the ledger.

KEY TAKEAWAYS

- The reward is triggered by a conversion webhook — a genuine first purchase — not by a click or a sign-up.
- The pending referral is flipped to *rewarded* with a conditional write, so a webhook that fires twice can only pay once.
- Both sides are credited in the same step: the referrer's reward and the friend's, each written to the ledger exactly once.
- The actual crediting is done deterministically through the store's API; one Bedrock call only writes the two thank-you notes.
- Anything the fraud screen holds, or a conversion for an already-rewarded referral, is never paid — it stops or goes to a person.

The one step that spends money

Everything until now has been careful bookkeeping: a code claimed, a click logged, a pending referral opened. None of it cost anything. This step is different, because this is where real value leaves the business — £10 of credit to the referrer, £10 off the friend's box — and it's the one place a mistake is expensive in both directions. Pay twice and you've handed out money you can't claw back; pay for a purchase that never really happened and you've rewarded a phantom. So the reward step is built around a single obsession: credit both sides exactly once, and only for a conversion that's genuinely real.

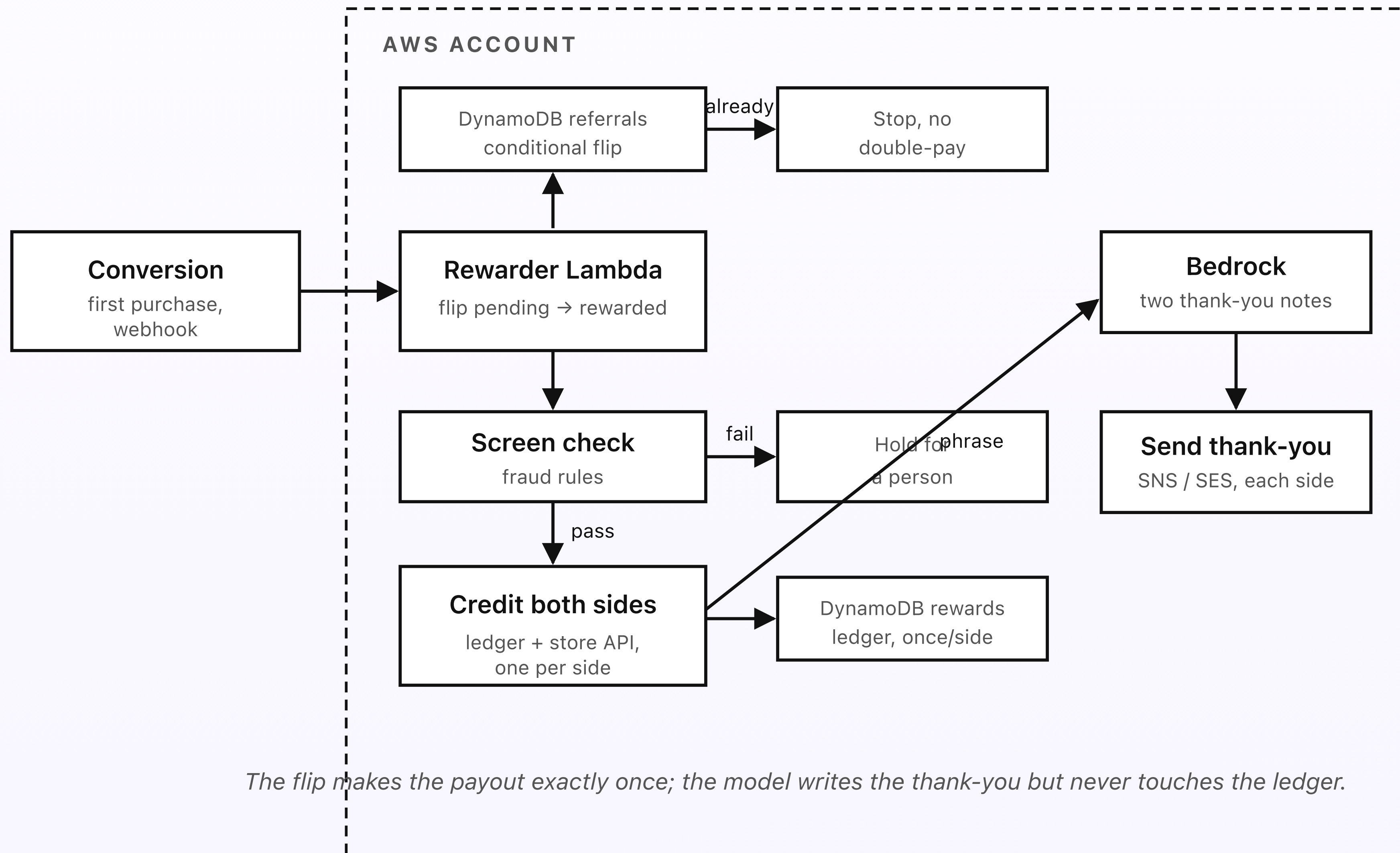
The trigger is the conversion webhook. When the referred friend makes their first real purchase — Mara's first FreshBox actually ships and is paid for, the barber's new customer pays for their first cut, the window cleaner's new sign-up settles their first clean — the store fires a webhook to the same Function URL that handles everything else. "First real purchase" is the store's call, not the model's: a paid, non-refunded order, past any cooling-off the business sets, so a sign-up that buys and immediately cancels never trips it.

Exactly once, both sides

Webhooks are famously unreliable in one specific way: they arrive more than once. A network hiccup, a retry the store thinks failed, a checkout that double-fires — any of these can deliver the same conversion two or three times. If the reward step simply paid out each time it was called, a flaky network would become free money. So the very first thing it does is claim the payout with a conditional write: flip this referral from *pending* to *rewarded*, but only if it's currently pending. The

first webhook wins the flip and proceeds to pay; any later copy finds the referral already *rewarded*, the conditional write fails, and it stops without paying a second time. Exactly-once isn't a hope here, it's a property of that one atomic write.

Only after winning the flip does the step credit the two sides, and it treats them as one unit of work. The referrer's reward and the friend's reward are each written to a rewards ledger keyed so that a given referral can only ever record one credit per side. The ledger is the source of truth for "has this been paid?" — separate from the store's own view — so even if fulfilment through the store has to be retried, the ledger keeps the count honest. Both sides, together, once.



The flip makes the payout exactly once; the model writes the thank-you but never touches the ledger.

Fig 4. Issuing a reward. A conversion webhook flips the referral from pending to rewarded with a conditional write — a duplicate can't pay twice. On a clean screen, both sides are credited once through the ledger and the store, and one Bedrock call writes the two thank-you notes that go out by SMS or email.

The model writes the thanks, not the cheque

Once both sides are credited, the last thing the step does is say thank you, and this is the only place a model runs at conversion time. A single Bedrock Haiku 4.5 call is handed the facts it may use — the two first names, the business voice, and the reward each side actually received — and writes two short notes: one to the referrer ("Mara just made her first order — your £10 is on your next box, thanks for the recommendation") and one to the friend ("welcome to FreshBox, your £10 has been applied"). Each goes out by whatever channel suits — SMS through SNS where a mobile is the contact, email through SES otherwise. The reward amounts in those notes come from the ledger the code already wrote, not from the model; Bedrock is told them so the wording is accurate, but it doesn't get to decide them.

That line matters more than it looks. The model never touches the ledger, never calls the store's API, never sees a total it could round up. It writes words about a payout that has already, deterministically, happened. If the Bedrock call is slow or the draft is off, a fixed thank-you template goes out instead — the credit is already applied, so a plain note is no loss. The warmth is nice to have; the accuracy is non-negotiable, and keeping them on opposite sides of that fence is what makes it safe to let a model near a system that moves money at all.

The payouts that don’t happen

Plenty of conversions arrive that shouldn’t pay, and the reward step is where they’re quietly turned away. A duplicate webhook loses the conditional flip and stops. A conversion for a referral the fraud screen has already *held* — a suspected self-referral, a code caught spraying — never gets past the screen check, and lands in front of a person instead of the ledger (that’s Part 5). A conversion for a friend who was never attributed has no pending referral to flip at all, so there’s simply nothing to pay. And a refund that arrives after a reward doesn’t silently reverse the credit; it’s flagged for a person, because clawing money back from a customer’s account is exactly the sort of decision that should never be automatic. In every one of these cases the default is the same: when in doubt, don’t pay, and tell a human why.

DESIGN RULES THAT SHAPED THE REWARD

- Only a conversion pays. A genuine, paid, non-refunded first purchase is the trigger — never a click or a sign-up.
- The flip is the guarantee. One conditional write from pending to rewarded makes the payout exactly once, however many times the webhook fires.
- Both sides, one unit. The referrer’s and the friend’s credits are written together, one entry per side, to a ledger that is the source of truth.
- Code credits, the model thanks. The store API and ledger do the crediting; Bedrock only writes the thank-you notes.
- The model never sees the ledger. Amounts come from the record; the model is told them for wording and decides nothing.
- When in doubt, don’t pay. Held, unattributed, duplicate, or post-refund conversions stop or go to a person — never an automatic payout.

PART 5 OF 7

JULY 4, 2026 PART 5 OF 7 · REFERRAL TRACKER SERIES ~8 MIN READ

How referral fraud gets caught

Any programme that pays people to bring friends will be tested by people bringing themselves. This post is about the screen that sits in front of every payout: how a self-referral is spotted, how repeat conversions are deduped, how a code behaving like a bot is held rather than paid, and why the honest majority never feel it.

KEY TAKEAWAYS

- Every referral that could pay out passes a fraud screen first — deterministic checks, run before any money moves.
- Self-referrals are caught by matching the referred person's identity — account, payment fingerprint, device — against the referrer.
- The same person converting twice is collapsed to one reward by keying the referral on the referred identity, not the sign-up.
- A code that suddenly sprays clicks or sign-ups trips a velocity check and is held for a human rather than paid.
- Clear abuse is blocked outright; anything merely suspicious is held and escalated — the honest majority never feel it.

Anyone you pay, you'll be tested by

The moment a business offers £10 for bringing a friend, some people will try to bring themselves. It's not always malicious — plenty is just a customer who figures a second account and a spare email is a harmless way to get their own discount — but it's money going out for customers who were never really referred, and left unchecked it's the thing that quietly kills a referral programme's economics. So a fraud screen sits in front of every payout, and its whole job is to answer one question before a reward is issued: is this a real friend, brought by a real recommendation, being credited for the first time? Only a clean yes gets paid automatically.

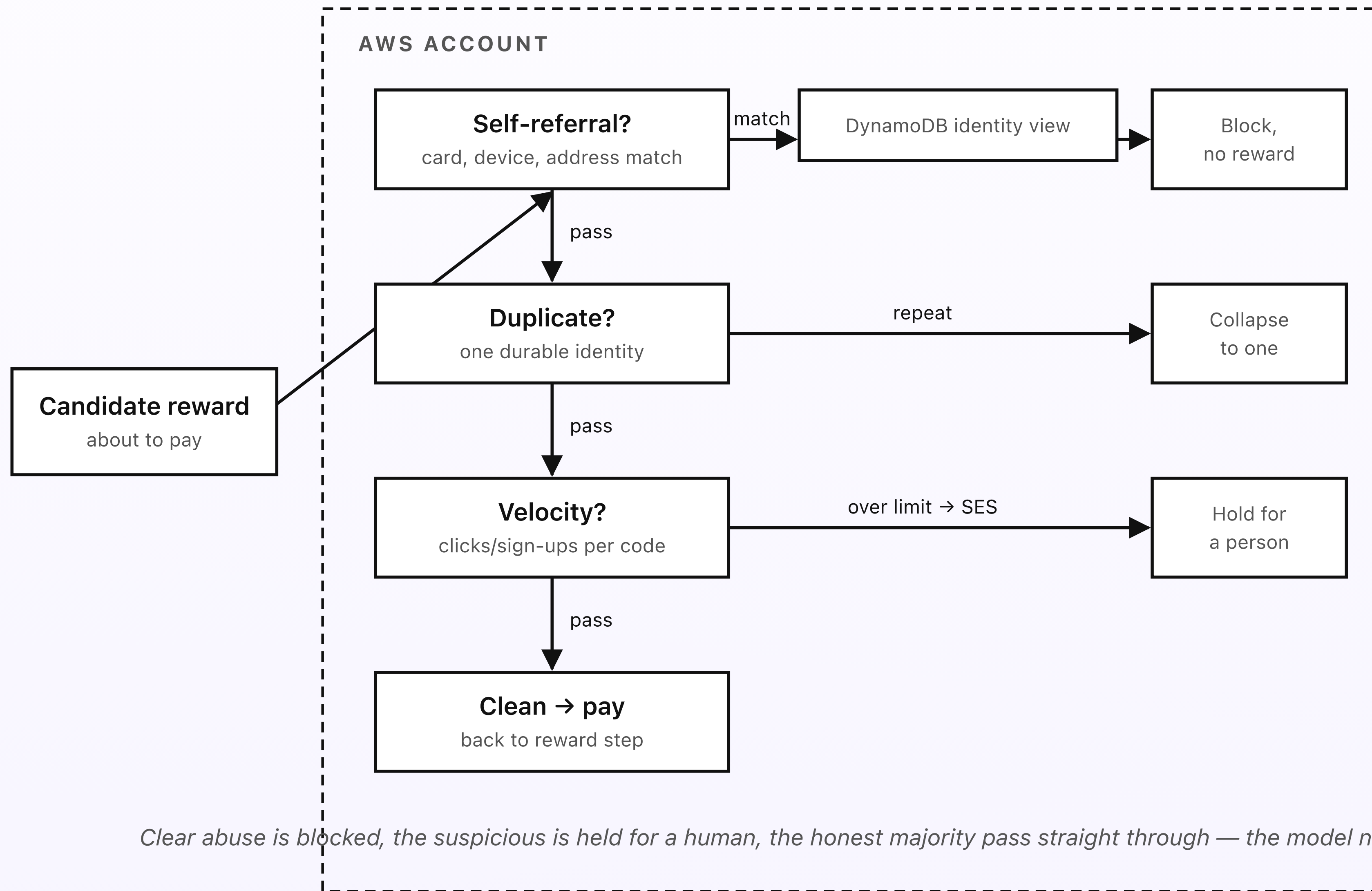
The screen is deliberately deterministic. It would be easy to imagine handing a model the case and asking "does this look fraudulent?" — and that's exactly what the design avoids, because a fraud decision has to be explainable, consistent, and defensible if a customer asks why they weren't paid. Every check here is a plain rule over facts the system already holds. The model writes invites and thank-yous; it gets nowhere near the question of who is or isn't cheating.

Three things it looks for

The screen runs three families of check. The first is **self-referral**: is the referred person actually the referrer wearing a different hat? At sign-up and again at conversion, the system compares the friend's identity against the referrer's across several axes — the account and email, the payment fingerprint (the same card or PayPal behind two "different" people is the strongest tell), the device and the

delivery address. No single match is proof, but the combination is: same card and same device as the referrer is a self-referral, full stop, and it's blocked outright.

The second is **duplicates**: has this friend already earned someone a reward? This is defended at the data model, not by a check that might be skipped — the referral is keyed on the referred person's durable identity, so the same human signing up three times under three emails still resolves to one referral record, and only the first conversion can ever pay. Farming the same friend through repeated sign-ups earns exactly one reward, or none. The third is **velocity**: is a code behaving like a script rather than a person? A real customer's link gets a handful of clicks and the odd sign-up; a code that suddenly logs two hundred clicks in an hour, or a burst of sign-ups all from one IP range or one device, is almost certainly being sprayed by a bot or gamed in a loop. That code is held — not deleted — and its pending referrals wait for a person, rather than paying out into the noise.



Clear abuse is blocked, the suspicious is held for a human, the honest majority pass straight through — the model never juc

Fig 5. The fraud screen. A candidate reward runs three deterministic checks in turn — self-referral by identity match, duplicates collapsed to one durable identity, and velocity per code. Clear abuse is blocked, suspicious cases are held and emailed to a person, and clean referrals pass through to be paid.

Block, hold, or pass

Each check resolves to one of three outcomes, and the difference between them is the whole design. A **block** is for the unambiguous: same payment fingerprint and device as the referrer is a self-referral, and it's declined without ceremony — no reward, a note on the record of exactly why, and no accusatory message to the customer. A **hold** is for the suspicious-but-not-certain: a velocity spike, a partial identity match, an address that's close but not identical. The referral is parked, and an email goes to your team — the referrer, the referred customer, the specific rule that fired, and the numbers behind it — so a person can approve it (and release the payout) or reject it in one click. A **pass** is the overwhelming majority: a real friend, a clean identity, ordinary volume, paid automatically within seconds.

Holding rather than blocking the grey cases is what keeps the screen fair. Fraud checks always catch some innocents — a couple who share a card and both genuinely love the meal-prep service will trip the payment-fingerprint rule — and the cost of wrongly blocking a real advocate is high: you've insulted your best kind of customer. Routing the ambiguous to a person, with the evidence laid out, means the system can be strict without being unjust. The rules are tuned to block only what's certain and hold everything else, and the thresholds live in the settings doc so a business can loosen them if it's turning away honest referrals or tighten them if it's being farmed.

The codes shared into the void

Not every disappointing referral is fraud. A great many codes are simply shared into the void: minted, pasted into a group chat, and never clicked — or clicked once and never signed up. These cost nothing and mean nothing, and the system treats them as exactly that: no reward, no alarm, no action beyond letting the click log expire on its TTL and the pending referral lapse when its window closes. It's worth naming the difference, because it's the difference between the screen's two failure modes. A code that does nothing is a non-event and is ignored. A code that does *too much, too fast* is a red flag and is held. The screen spends its attention on the second and wastes none on the first, which is exactly why it stays cheap to run and quiet to live with.

DESIGN RULES THAT SHAPED THE FRAUD SCREEN

- Screen before you pay. Every candidate reward passes the checks first; nothing is credited on trust.
- Deterministic, explainable rules. Fraud is decided by plain checks over known facts — never handed to a model to judge.
- Identity beats email. Self-referral is caught on payment fingerprint, device, and address, not on a throwaway address.
- Dedup at the data model. The referral is keyed on the referred person's durable identity, so repeat sign-ups can only pay once.
- Block the certain, hold the rest. Clear abuse is declined outright; the ambiguous is parked for a person with the evidence attached.
- A quiet code is not a crime. Codes shared into the void just expire; only codes behaving like scripts raise a flag.

PART 6 OF 7

JULY 4, 2026 PART 6 OF 7 · REFERRAL TRACKER SERIES ~7 MIN READ

What the referral tracker costs

This is the least expensive system in the collection, and the cost breakdown shows why: the busy path is just redirects and tiny writes. This post is every AWS line it touches at around 200 clicks and 20 conversions a month, why the total lands near $1.70, and what happens to the bill when the volume goes up tenfold.

KEY TAKEAWAYS

- About $1.70/month at roughly 200 clicks and 20 conversions — the cheapest system in this collection, because the busy path is just redirects and tiny writes.
- Nearly half the bill is one fixed line: Secrets Manager, two provider secrets at $0.40 each, billed whether or not anyone refers a soul.
- The variable lines are small — a couple of Bedrock calls, some SMS, and cents of DynamoDB, Lambda, and logs.
- At ten times the volume the bill lands near $9, because the fixed lines don't move and the variable work is genuinely cheap.
- 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 referrals and no idle bill. More than that, the thing it does most often — redirect a click — is about the cheapest operation on AWS: a few milliseconds of Lambda and one tiny DynamoDB write. The expensive-sounding parts, the model calls, only happen twice per referral at most: once when a link is minted, once when it converts. At a typical small-business volume — call it 200 link clicks, 40 sign-ups, and 20 conversions in a month — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two provider secrets — webhook signing key, store/reward API key ($0.40 each)	$0.80
Bedrock (Claude Haiku 4.5)	One invite per minted link, two thank-you notes per conversion	$0.30
DynamoDB (on-demand)	Codes, click log, referrals, rewards ledger, audit — many tiny writes	$0.20
SNS (SMS)	A few reward-confirmation texts where a mobile is the contact	$0.15
Lambda (Python 3.14, arm64)	Redirect, minter, attributor, rewarder, screen, sweep	$0.10
CloudWatch Logs	Function logs, 7-day retention	$0.08
SES	Invite, thank-you, and fraud-review email	$0.05
SQS + DLQ	Buffering sign-up and conversion jobs behind the redirect path	$0.01
EventBridge Scheduler	The attribution-window expiry sweep	$0.01
AWS Budgets	Cost alarm (first two budgets are free)	$0.00

AWS service	What it does here	Monthly
Total	**~200 clicks / ~20 conversions per month**	**$1.70**

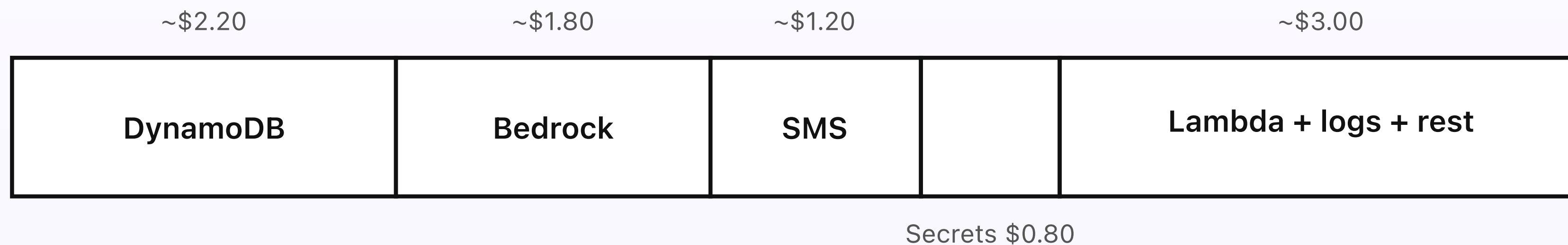
The shape of that bill is the point. The single biggest line — nearly half the total — is Secrets Manager, and it's fixed: two secrets at $0.40 each, $0.80 a month whether the programme sends one referral or a thousand. Everything else is genuinely usage-priced and rounds towards zero when the phone's quiet. The model calls are the next line down, and they're modest because they're rare — you pay for a bit of Haiku only when a link is born or a friend converts, never on the hundreds of clicks in between. The redirects, the writes, the queue, the schedule — all the machinery doing the real work of tracking and crediting — together cost less than the SMS line. This is what a system built almost entirely from cheap operations looks like on a bill.

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 $0.15 here is a UK-leaning estimate for the handful of reward texts a small programme sends; most notifications in this system are email through SES, which is cheaper still, so SMS is a minor line to begin with. If you send thank-yous entirely by email it drops off the table almost completely. Either way it's the one line that tracks a real-world price list rather than an AWS one, which is why the Budgets alarm watches the total.

What ten times the volume costs

Push this to a busy business — 2,000 clicks, 400 sign-ups, and 200 conversions a month, ten times the volume — and the bill lands near $9, not $17. It's sub-linear because the fixed line doesn't move: Secrets Manager stays at $0.80, the schedule stays at a cent, and AWS Budgets stays free. What scales is the genuinely usage-priced work — the click log and the small writes grow into the largest variable line at a couple of dollars, the model calls rise to under $2 as more links are minted and more convert, and SMS, Lambda, logs, and SES add a few dollars more between them. Even at ten times the traffic, the redirects that dominate the volume barely register, because each one is worth a fraction of a cent.



Fixed lines don't move with volume; the redirects that dominate the traffic barely register, so the bill grows sub-linearly.

Fig 6. The monthly bill at ten times the base volume, about 2,000 clicks and 200 conversions. The click log and small writes become the largest line, Bedrock and SMS follow, and the fixed Secrets Manager line stays put — so the total grows sub-linearly, near $9 rather than ten times $1.70.

The honest way to read this: the AWS bill is rounding error against what a referral is worth. One converted friend at a meal-prep service, a barber, or a window cleaner is worth far more than $1.70 — a subscription that runs for months, a customer who comes back every three weeks — and this design catches them by the handful and pays their advocates fairly. Even at $9 a month for a busy

programme, it pays for itself the first time it turns one shared link into one real customer, and it does the counting, the crediting, and the fraud-checking that a person would never keep up with by hand.

DESIGN RULES THAT SHAPED THE COST

- Build from cheap operations. The busy path is redirects and tiny writes, the two cheapest things AWS does — so volume is nearly free.
- Spend the model twice, not two hundred times. Bedrock runs at mint and at conversion, never on a click — the copy stays a minor line.
- Know your one fixed cost. Secrets Manager is nearly half the base bill and the only line that bills while the system sleeps.
- Prefer email to SMS. Most notifications go by SES; SMS is a small, country-varying line kept for where a mobile is the only contact.
- Watch the total. A Budgets alarm sits over the whole bill, the cheapest early warning that volume — or a loop — is running hot.

PART 7 OF 7

JULY 4, 2026 PART 7 OF 7 · REFERRAL TRACKER SERIES ~11 MIN READ

Engineering reference: the referral tracker architecture

This is the referral tracker with the friendly labels removed: the real resource names, the runtime, the table key schemas, the single public Function URL that both redirects and receives webhooks, the exactly-once ledger, 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 sign-up and conversion webhooks buffered through one SQS queue with a dead-letter queue.
- One public surface: a single Lambda Function URL on `rft-edge` that serves the link redirect and takes the sign-up and conversion webhooks — no API Gateway.
- Five DynamoDB tables, all on-demand: codes, a TTL'd click log, referrals keyed on the referred identity, a rewards ledger, and an append-only audit.
- Exactly-once payout is a property of two writes: a code claim, and a conditional flip of the referral from pending to rewarded.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by the minter and the rewarder. 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 surface is one Lambda Function URL that both redirects link clicks and receives the store's webhooks, outbound messaging goes through SNS and SES, and slow work is buffered on a single SQS queue.

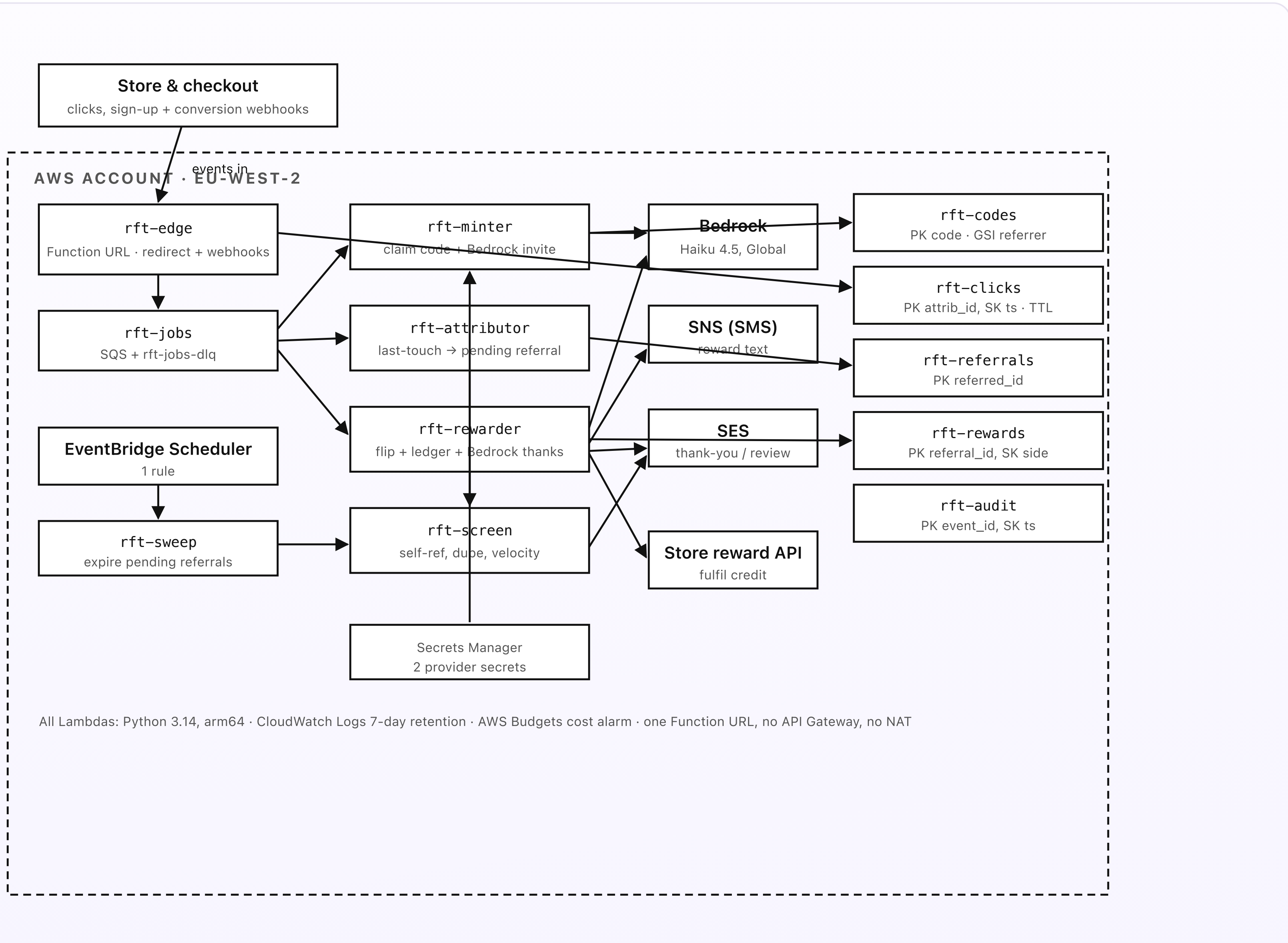

Store & checkout
clicks, sign-up + conversion webhooks
events in
AWS ACCOUNT · EU-WEST-2
rft-edge
Function URL · redirect + webhooks
rft-jobs
SQS + rft-jobs-dlq
EventBridge Scheduler
1 rule
rft-sweep
expire pending referrals
rft-minter
claim code + Bedrock invite
rft-attributor
last-touch → pending referral
rft-rewarder
flip + ledger + Bedrock thanks
rft-screen
self-ref, dupe, velocity
Secrets Manager
2 provider secrets
Bedrock
Haiku 4.5, Global
SNS (SMS)
reward text
SES
thank-you / review
Store reward API
fulfil credit
rft-codes
PK code · GSI referrer
rft-clicks
PK attrib_id, SK ts · TTL
rft-referrals
PK referred_id
rft-rewards
PK referral_id, SK side
rft-audit
PK event_id, SK ts
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm · one Function URL, no API Gateway, no NAT

Fig 7. The referral tracker drawn for engineers: one Function URL on `rft-edge` that redirects and receives webhooks, an SQS-buffered set of six Lambdas, five DynamoDB tables, Bedrock called only by the minter and rewarder, SNS and SES for messaging, and one scheduled expiry sweep. 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 (`rft-jobs`, with `rft-jobs-dlq` as its dead-letter queue after five attempts) decouples the store's webhooks from the slower model, ledger, and store-API calls. The link redirect is the one path that is *not* queued — it runs synchronously inside `rft-edge` so the click is logged and the 302 returned in milliseconds.

- `rft-edge` — the only public surface. Backs the single Lambda Function URL and handles three event kinds. For a link click (`GET /r/{code}`): resolves the code in `rft-codes`, writes a row to `rft-clicks`, sets the attribution token, and returns a `302`. For a sign-up or conversion webhook: verifies the provider signature against the secret, and enqueues a typed job on `rft-jobs`. For a mint request: enqueues a mint job. Nothing slow happens here.
- `rft-minter` — SQS-triggered on mint jobs. Reuses the customer's active code via the `rft-codes` referrer GSI or claims a new one with a conditional write, pins the reward terms and window, makes the single Bedrock call for the invite, injects the real URL, and returns the link.
- `rft-attributor` — SQS-triggered on sign-up jobs. Reads `rft-clicks` for the most recent valid click within the window, resolves the referred person to a

durable identity, calls `rft-screen`, and opens a *pending* item in `rft-referrals` (or leaves the sign-up unattributed).

- `rft-rewarder` — SQS-triggered on conversion jobs. Flips the referral `pending → rewarded` with a conditional write, re-checks `rft-screen`, writes one entry per side to `rft-rewards`, calls the store reward API to fulfil, makes the single Bedrock call for the two thank-you notes, and sends via SNS or SES.
- `rft-screen` — invoked synchronously by the attributor and rewarder. Runs the deterministic self-referral, duplicate, and velocity checks over `rft-referrals`, `rft-clicks`, and the identity fields; returns block / hold / pass, and on a hold emails your team via SES with the rule that fired and the evidence.
- `rft-sweep` — scheduled. Queries `rft-referrals` for *pending* items whose window has elapsed with no conversion and marks them `expired`, so a code shared into the void is tidied away rather than lingering as an open liability.

Data stores, schedules, and messaging

- **DynamoDB (all on-demand).** `rft-codes` — PK `code`, with a GSI on `referrer_id` to find a customer's active code; holds the owner, the pinned reward terms and window, and a `status` (`active` / `retired`). `rft-clicks` — PK `attribution_id` (the token), SK `click_ts`, with a TTL equal to the window so expired clicks vanish and can't attribute; queried newest-first for last touch. `rft-referrals` — PK `referred_id` (the referred person's durable identity), one item per referred customer so repeat sign-ups collapse to one; carries `referrer_id`, `code`, `state` (`pending` / `rewarded` / `held` / `rejected` / `expired`), the pinned terms, and the identity fingerprints. `rft-rewards` — PK `referral_id`, SK `side` (`referrer` / `friend`), the exactly-

once ledger of what was actually paid. `rft-audit` — PK `event_id`, SK `ts`, append-only, holding each event and the facts a decision was made on.

- **Function URL.** One, on `rft-edge`, with provider signature verification in-function for the webhooks; `AuthType NONE` at the edge because authenticity is enforced by the shared secret, not by IAM, and the redirect route is public by nature. No API Gateway.
- **SNS and SES.** SNS sends the reward-confirmation SMS where a mobile is the contact; SES sends the invite (if delivered by the business rather than pasted by the customer), the thank-you emails, and the fraud-review email to the team from a verified domain with DKIM.
- **EventBridge Scheduler.** One rule — `rft-sweep` at `rate(1 hour)`, expiring pending referrals past their window; the granularity is loose because expiry is not time-critical.
- **Secrets Manager.** Two secrets — the webhook signing secret and the store/reward API key — fetched at call time, never in env vars or the settings doc.
- **Bedrock.** Model id `anthropic.claude-haiku-4-5` via the Global cross-Region inference profile, invoked only by `rft-minter` and `rft-rewarder`.

Exactly-once and failure handling

Two properties carry the whole design, and both are enforced by DynamoDB conditions rather than by hope. **Codes are unique** because minting claims each with `attribute_not_exists(code)`; a collision fails the write and the minter draws again. **Payouts happen once** because the rewarder flips `rft-referrals`

from `pending` to `rewarded` under `ConditionExpression state = pending`; the first conversion webhook wins the flip, and any duplicate finds the condition false and stops before touching the ledger. Keying `rft-referrals` on the referred person's durable identity does the third piece of work — the same human signing up under many emails resolves to one item, so they can only ever be rewarded once. The `rft-rewards` ledger adds belt-and-braces with a conditional put per `(referral_id, side)`.

Failure handling is ordinary and boring, which is the point. The webhook-driven work runs from `rft-jobs`; a handler that throws is retried by SQS and, after five attempts, parked in `rft-jobs-dlq` for inspection rather than lost or looped. The store reward API is called after the ledger write, so a fulfilment that fails can be retried idempotently against a ledger that already knows the payout is owed — the credit is never double-issued because the ledger, not the API call, is the source of truth. Bedrock is best-effort on both paths: if the invite or thank-you call is slow or errors, a fixed template goes out and the referral is unaffected, because the model only ever wrote words.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `rft-edge` can read `rft-codes`, write `rft-clicks`, read the signing secret, and send to `rft-jobs` — it cannot call Bedrock, SNS, or the store API, and it cannot touch `rft-referrals` or the ledger. `rft-minter` can conditionally write `rft-codes` and invoke Bedrock, nothing more. `rft-attributor` can read `rft-clicks`, write `rft-referrals`, and invoke `rft-screen`. `rft-rewarder` is the only role that can flip a referral, write `rft-rewards`, call the store reward API,

publish to SNS, and invoke Bedrock — and it cannot delete from any table. `rft-screen` reads its inputs and sends via SES only. `rft-sweep` can query and update `rft-referrals` and has no inbound surface at all. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call for capacity and is not a data store. An AWS Budgets alarm watches the monthly spend — with a runaway loop (a webhook storm, a code being sprayed) the thing most likely to move it, a Budgets alert is the cheapest early warning that something is running hot, and it pairs with a CloudWatch alarm on the `rft-jobs-dlq` depth.

That's the whole system: a redirect that logs a click, a queue that carries the rest, six small functions, five tables, and two conditional writes that make the money move exactly once. It turns a customer's offhand recommendation into a link you can follow and a reward you can trust — and it does it for less than the price of the coffee the referrer and their friend might have had while they talked you up.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Six small Lambdas beat one that does everything; the queue decouples the slow calls from the public edge.
- One public surface. Only `rft-edge` is reachable from outside, on a single Function URL that both redirects and takes webhooks.
- Exactly-once by condition. A code claim and a pending-to-rewarded flip, both conditional writes, are what make the payout happen once.
- Least privilege, per role. Only the rewarder can move money; only the minter and rewarder can call Bedrock; the edge can't reach either.
- The ledger is the truth. Fulfilment is retried against `rft-rewards`, so a failed store call never double-credits.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called only at mint and conversion.