7-PART SERIES · FREE COMPANION

Chargeback responder

A customer files a payment dispute — a chargeback — and the clock starts the moment it lands: the bank wants your evidence within a tight window, and miss it and you lose by default. This is a small serverless system that catches the dispute the instant the processor reports it, gathers the evidence a bank actually looks at — the order, proof of delivery and tracking, the customer's own messages, and the refund and shipping policy they agreed to — and uses Bedrock to assemble a structured, on-point rebuttal packet. It files the response through the processor's API well before the deadline and tracks the outcome, and it never fabricates evidence: if a case is unwinnable it says so and recommends taking the loss rather than burning the fee. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

Free lite starter + this PDF · paid tiers at

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

shop.allanninal.dev/w/chargeback-responder

CONTENTS

Chargeback responder

PART 1 OF 7

JUNE 30, 2026 PART 1 OF 7 · CHARGEBACK RESPONDER SERIES ~10 MIN READ

A chargeback responder on AWS for a few dollars a month

A chargeback is a small emergency on a deadline. A customer disputes a charge, the bank pulls the money straight back, and you have a tight, unforgiving window to prove the sale was real — or you lose the amount and the fee, every time. Doing that by hand means dropping what you're doing to dig out an order, chase a tracking number, find the support thread, and write a coherent rebuttal before a date you'll probably forget. This post walks through the design of a small system that catches every dispute the moment it lands, gathers the evidence a bank actually looks at, assembles a structured rebuttal, and files it on time — without ever inventing a thing.

KEY TAKEAWAYS

- One dispute webhook in, one filed response out — the system runs the whole window in between, on a clock.
- It gathers the four things a bank actually weighs: the order, proof of delivery, the customer's messages, and the agreed policy.
- Bedrock assembles a structured rebuttal tied to the dispute's reason code; the deadline and the decision to file are plain code.
- A deadline watch guarantees the response is filed before the bank's cutoff — never late, with owner approval or on its own.
- Designed on AWS for about $2.40/month at typical volume. It never fabricates evidence, and on an unwinnable case it says so.

The whole system on one page

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

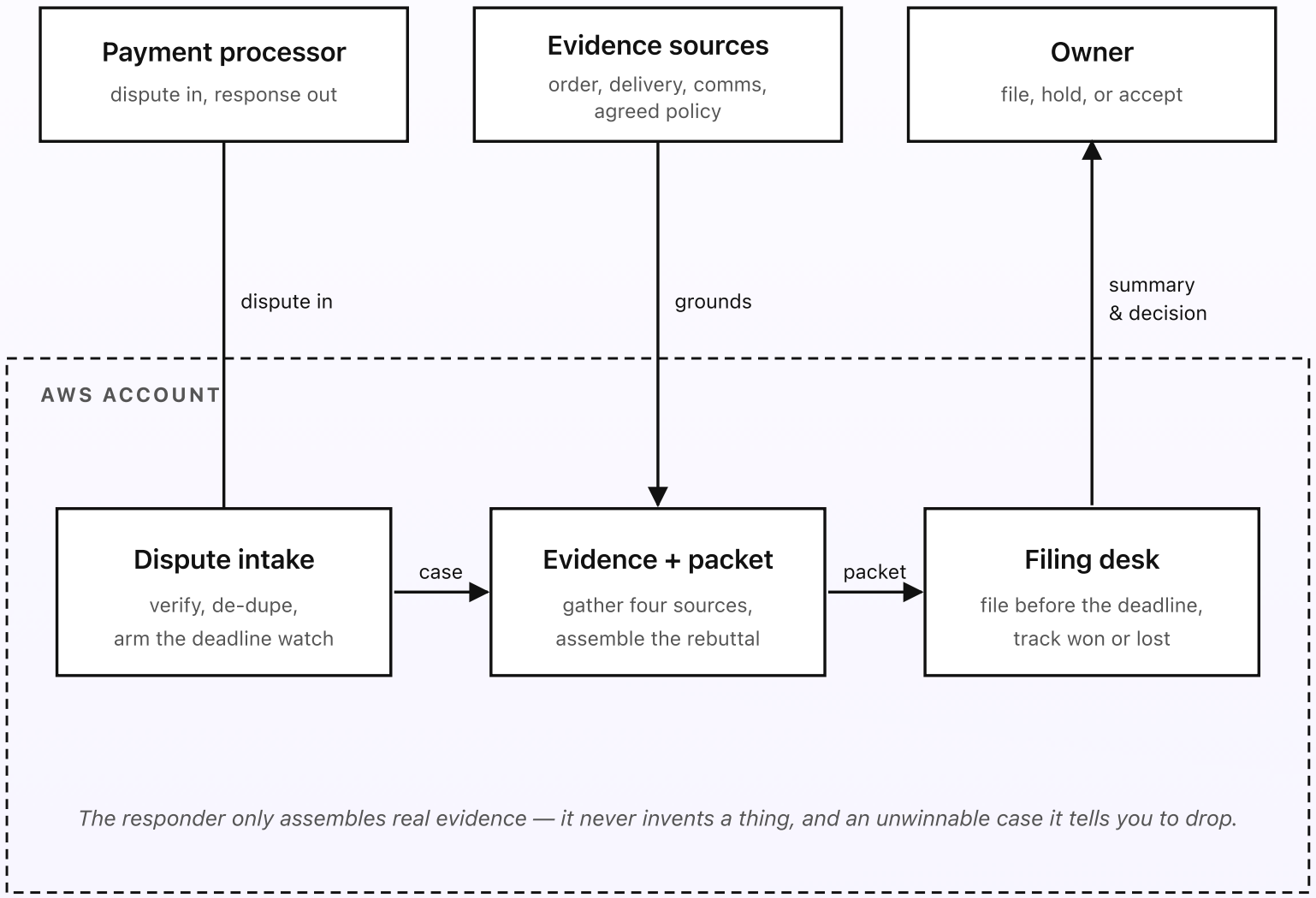

Payment processor
dispute in, response out
Evidence sources
order, delivery, comms, agreed policy
Owner
file, hold, or accept
dispute in
grounds
summary & decision
AWS ACCOUNT
Dispute intake
verify, de-dupe, arm the deadline watch
case
Evidence + packet
gather four sources, assemble the rebuttal
packet
Filing desk
file before the deadline, track won or lost
The responder only assembles real evidence — it never invents a thing, and an unwinnable case it tells you to drop.

Fig 1. One processor outside, the evidence sources in the middle, three pieces inside AWS. A dispute flows in; the intake arms a deadline watch; the engine gathers evidence and assembles a rebuttal; the filing desk files before the bank's cutoff and tracks the result.

What you set up once (the outside)

- **The processor connection.** One processor (Stripe, or a Stripe-style gateway) is wired two ways. Outbound from them: a webhook endpoint that fires on `charge.dispute.created` and `charge.dispute.closed`, pointed at a Lambda Function URL. Inbound to them: an API key, kept in Secrets Manager, that the system uses to read the full dispute and to *submit* the evidence before the bank's deadline. The deadline itself comes from the processor on the dispute object — the `evidence_details.due_by` timestamp — so the system never has to guess how long it has.
- **The evidence sources.** The four systems that already hold your proof. The *order record* in your commerce platform (what was bought, when, for how much, the billing and shipping addresses, the AVS and CVC checks at the time of sale). The *carrier* that delivered it, read through its tracking API for the delivery confirmation, the timestamp, and a signature or photo where there is one. The *support inbox or chat* where the customer's own messages live. And the *refund and shipping policy* the customer agreed to at checkout, kept with the version and the timestamp of their acceptance. You don't build any of these — the system just reads them.
- **The owner.** The person who decides whether a given dispute is worth fighting. They get one email per dispute: the amount, the reason code, a plain-English read of how strong the evidence is, and three buttons — *File* (send the

assembled packet now), *Hold* (pause for a closer look), and *Accept loss* (don't fight it). In auto-file mode the strong cases file themselves and the email is just a heads-up; in owner-approval mode nothing files until they tap.

What runs on every dispute (the inside)

- **The dispute intake.** The webhook lands on a Function URL. A small Lambda verifies the processor's signature so a forged dispute can't enter, throws away the processor's inevitable retries with an idempotency key, and reads the dispute: amount, currency, reason code, the charge it disputes, and the `due_by`. It writes a row to the disputes table, records the deadline, and arms a one-off EventBridge Scheduler job for a safe margin before that deadline — the watch that guarantees the case can never silently run out of time. Then it enqueues the gather job.
- **The evidence and packet engine.** A gather Lambda pulls the four sources — order, delivery proof, customer messages, agreed policy — into an S3 evidence bucket and writes an index row per piece, marking honestly where something is missing. An assemble Lambda then makes one Bedrock Haiku 4.5 call: it hands the model the gathered evidence and the reason code and gets back a structured rebuttal that ties each real piece to the specific claim the bank is testing, plus a winnability read. It renders the packet to a PDF in S3. No model decides whether to file, and no model is allowed to write evidence that wasn't gathered.
- **The filing desk.** Takes the assembled packet and, per the mode and the owner's choice, files it through the processor's API — always before the deadline. If the owner hasn't answered and hasn't chosen to accept the loss, the deadline guard files the assembled packet at the safety cutoff rather than

let the case lapse. When the processor later posts `dispute.closed`, it records won or lost against the case, so the trail is complete. A monthly summary writes a short narrative: disputes received, value defended, win rate by reason code.

In plain words

An £85 order from your store is disputed as "product not received." At 02:14 the processor posts `charge.dispute.created`; the bank's evidence is due in fourteen days. The intake verifies it, records the deadline, and arms the watch. The gather Lambda pulls the order (placed, paid, shipped to the billing address), the carrier record (delivered, signed for, with a timestamp three days after the order), the support thread (the customer messaged "got it, thanks" the next week), and the shipping policy they accepted at checkout. Bedrock assembles a rebuttal: "Reason code: product not received. Carrier confirms delivery on 14 March, signed; customer acknowledged receipt on 21 March; shipped to the verified billing address." It rates the case strong. The owner gets the summary, taps *File*, and the packet is submitted that morning — ten days early. Three weeks later `dispute.closed` arrives: won. The £85 stays, and so would have happened automatically if the owner had been on holiday.

The cost of running this is about $2.40 a month. The cost of *not* running it is every dispute you lose because the deadline slipped past while you were busy — the disputed amount, plus the non-refundable £15–20 dispute fee, on every single one.

DESIGN RULES THAT SHAPED EVERY DECISION

- It never fabricates evidence. Every line in the packet traces to a real record — an order, a tracking event, a message, a policy.
- The deadline is sacred. A one-off watch files the assembled packet before the cutoff, so a case can never lapse by default.
- An unwinnable case gets said out loud. If the proof isn't there, the system recommends accepting the loss, not wasting the fee.
- No model touches money. Whether to file and when are plain code; Bedrock only assembles the rebuttal from gathered evidence.
- Auto-file or owner-approval is a setting. You choose how much happens without you; either way nothing files late.
- Every dispute and outcome is logged. Audit a chargeback next year and you can see what was filed, when, and why.

Why this shape

Most small businesses handle chargebacks one of three ways: they don't fight them at all (and quietly write off the losses), they fight them by hand under time pressure (and miss half the deadlines), or they pay a percentage-of-recovery service that takes a cut of every win. The first leaves money on the table. The second is exactly the kind of dull, deadline-bound work that slips when the week gets busy — and a missed deadline isn't a delay, it's an automatic loss. The third works but is expensive at SMB volume and opaque about what it files in your name.

The setup above keeps your own systems — the order, the carrier, the inbox, the policy — as the only source of truth, and adds a small system that *reacts* the moment a dispute opens. It gathers the real evidence, assembles a rebuttal a bank will actually read, and files it early. Strong cases can clear in one tap, so you spend your attention only where it matters. Weak cases get flagged as weak, so you don't burn a fee on a fight you can't win. And the deadline can never beat you, because a guard files before the cutoff whether you're watching or not.

The next four posts walk through each piece in turn: how a dispute gets caught, how the evidence gets gathered, how an evidence packet gets built, and how a response gets filed on time. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 30, 2026 PART 2 OF 7 · CHARGEBACK RESPONDER SERIES ~7 MIN READ

How a dispute gets caught

Everything downstream depends on two things landing correctly the instant a dispute opens: that it really came from your processor, and exactly when the bank's evidence is due. This post is about the front door — a Stripe-style dispute webhook arriving on a Lambda Function URL, the signature check that stops a forged one, the de-duplication that survives the processor's retries, and the deadline that gets written down and armed with a watch so nothing can quietly run out of time.

KEY TAKEAWAYS

- The dispute webhook lands on a Lambda Function URL — no API Gateway, just a verified front door.
- The processor's signature is checked first; an unsigned or forged dispute never gets past the front door.
- Every event is de-duplicated by its dispute id, so the processor's retries can't open the same case twice.
- The bank's evidence deadline is read straight off the dispute and written down before anything else happens.
- A one-off deadline watch is armed the instant the dispute is caught — the case can never silently run out of time.

The front door: a Function URL, not an API Gateway

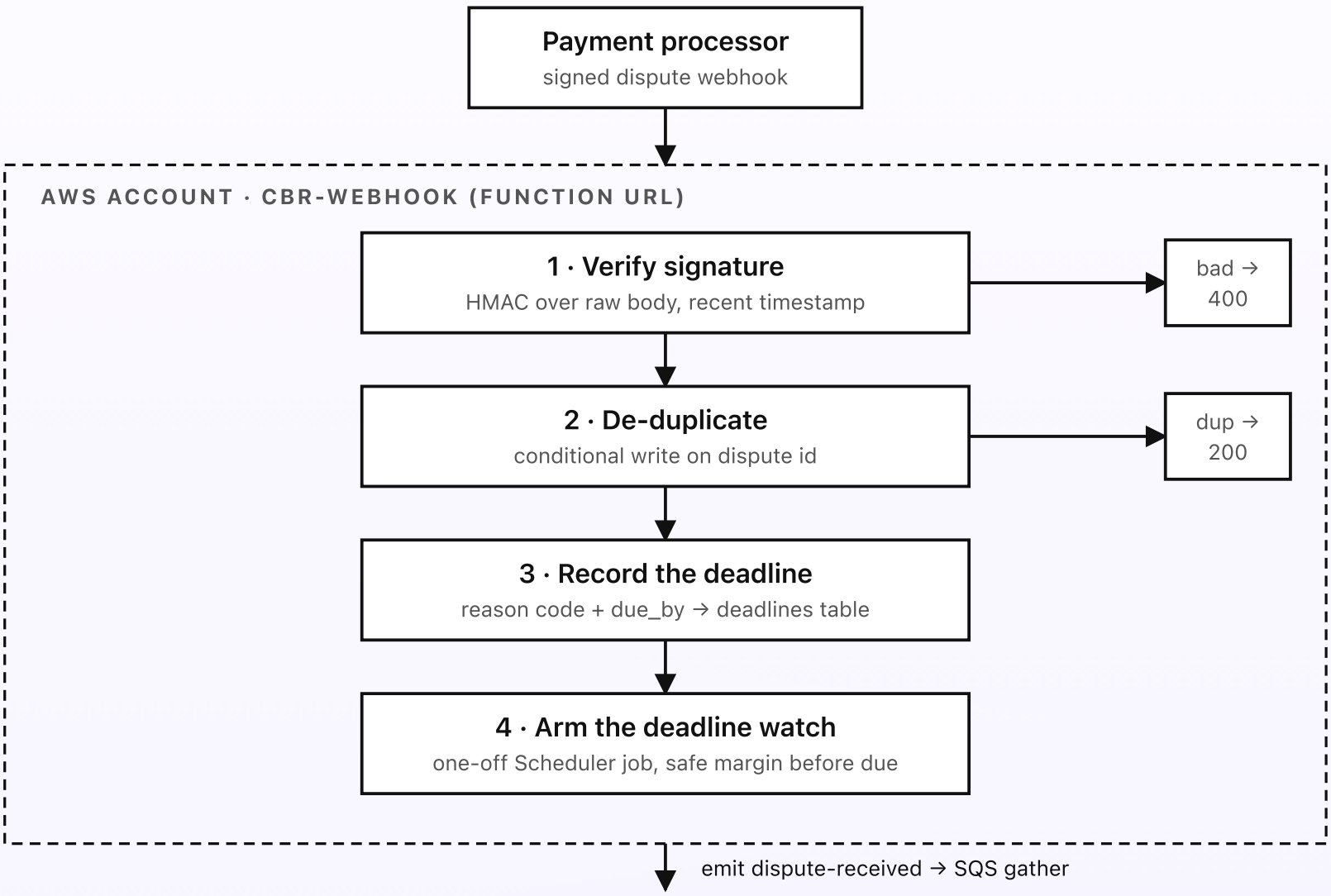
A dispute arrives as an HTTP POST from the processor — a JSON body describing the event, signed with a shared secret. There's exactly one caller (your processor), low volume (a handful of disputes a day at most), and no need for the routing, throttling, or request-validation machinery an API Gateway brings. So the endpoint is a plain Lambda Function URL: a single HTTPS address that invokes one function directly. It costs nothing when idle and rounds to nothing under load at this scale. The Function URL is set to `AuthType: NONE` — the request is authenticated by the processor's signature, checked inside the function, not by IAM.

Verify the signature before trusting a single field

The very first thing the function does — before it reads the amount, the reason code, or anything else — is verify the request really came from your processor. The processor signs each webhook with a secret you both hold; it sends the signature in a header (Stripe puts it in `Stripe-Signature` with a timestamp and an HMAC of the raw body). The function recomputes the HMAC over the *raw* body using the signing secret from Secrets Manager, compares it in constant time, and checks the timestamp is recent so an old captured request can't be replayed. If the signature doesn't match, the function returns `400` and stops. Nothing else in the system ever sees an unverified dispute.

De-duplicate, then write the deadline down

Processors retry webhooks — if your endpoint is slow or returns an error, the same dispute event may arrive several times, and the `created` and an early `updated` can look almost identical. So the function de-duplicates on the dispute id with a conditional write: it tries to create the row in the disputes table only if one doesn't already exist. A duplicate loses the race, the function returns `200` (so the processor stops retrying), and no second case is opened. On the winning write it reads the two fields that matter most — the reason code (which decides what evidence the packet will need) and `evidence_details.due_by` (the bank's hard deadline) — and records the deadline in its own table before doing anything slower.



Nothing slower than a database write happens until the deadline is recorded and watched.

Fig 2. The front door, gate by gate. Verify the signature, de-duplicate on the dispute id, record the deadline, arm the one-off watch — then, and only then, hand the case to the evidence engine.

Arm the watch before anything slow happens

Here's the ordering that matters: the function records the deadline and arms its watch *before* it kicks off the slow work of gathering evidence. The moment the deadline is in the table, the function creates a one-off EventBridge Scheduler job — an `at(...)` rule timed for a safe margin before `due_by` (a day, by default) — targeting the deadline-guard Lambda. If everything downstream went perfectly, that job will find the case already filed and do nothing. But if gathering stalls, or the owner never taps a button, the watch is the backstop that still files before the bank's cutoff. Arming it first means the safety net exists even if the next step crashes.

Only after the deadline is recorded and watched does the function emit a `dispute.received` event onto the EventBridge bus and drop a job on the gather queue. It returns `200` to the processor in well under a second — the heavy lifting happens asynchronously, off the webhook's critical path, which is exactly why a slow carrier API can never cause the processor to think the webhook failed and retry.

Why this order, every time

The sequence — verify, de-duplicate, record the deadline, arm the watch, then hand off — is deliberate. Verifying first means no forged or replayed dispute ever

touches your data. De-duplicating second means the processor's retries are free; the endpoint is idempotent by construction. Recording and watching the deadline third and fourth means the one thing you can never recover from — a missed filing window — is locked in before any fragile, slow, or failable work begins. Everything after the front door can be retried; the deadline cannot be un-missed. So the front door's whole job is to make the deadline safe, fast, and exactly once.

DESIGN RULES FOR THE FRONT DOOR

- One Function URL, signature-checked. No API Gateway, and no field is trusted before the HMAC verifies.
- Idempotent by dispute id. The processor can retry the same event forever and only one case is ever opened.
- Deadline first, evidence second. The bank's due-by is written and watched before any slow work runs.
- Arm the watch eagerly. The safety net is created up front, so a crash downstream still can't cause a late filing.
- Answer the processor fast. Acknowledge in under a second and do the real work asynchronously off the webhook path.

PART 3 OF 7

JUNE 30, 2026 PART 3 OF 7 · CHARGEBACK RESPONDER SERIES ~7 MIN READ

How the evidence gets gathered

A rebuttal is only as good as the evidence under it, so this is the part that does the digging. This post is about how the system gathers the four things a card network's rules actually ask for — what was ordered, that it was delivered, that the customer engaged with it, and what they agreed to — each pulled from the system that really holds it, stored once, and indexed so the next step knows exactly what it has and, just as importantly, what it doesn't.

KEY TAKEAWAYS

- One gather Lambda pulls four things, because four things are what a card network's rules actually weigh.
- The order record, proof of delivery and tracking, the customer's own messages, and the agreed policy — each from its real source.
- Everything gathered is written once to an S3 evidence bucket and indexed, one row per piece, in DynamoDB.
- The reason code decides which pieces matter most, so the gather is targeted, not a blind sweep.
- When a piece is missing, the index records it as missing — the system never papers over a gap.

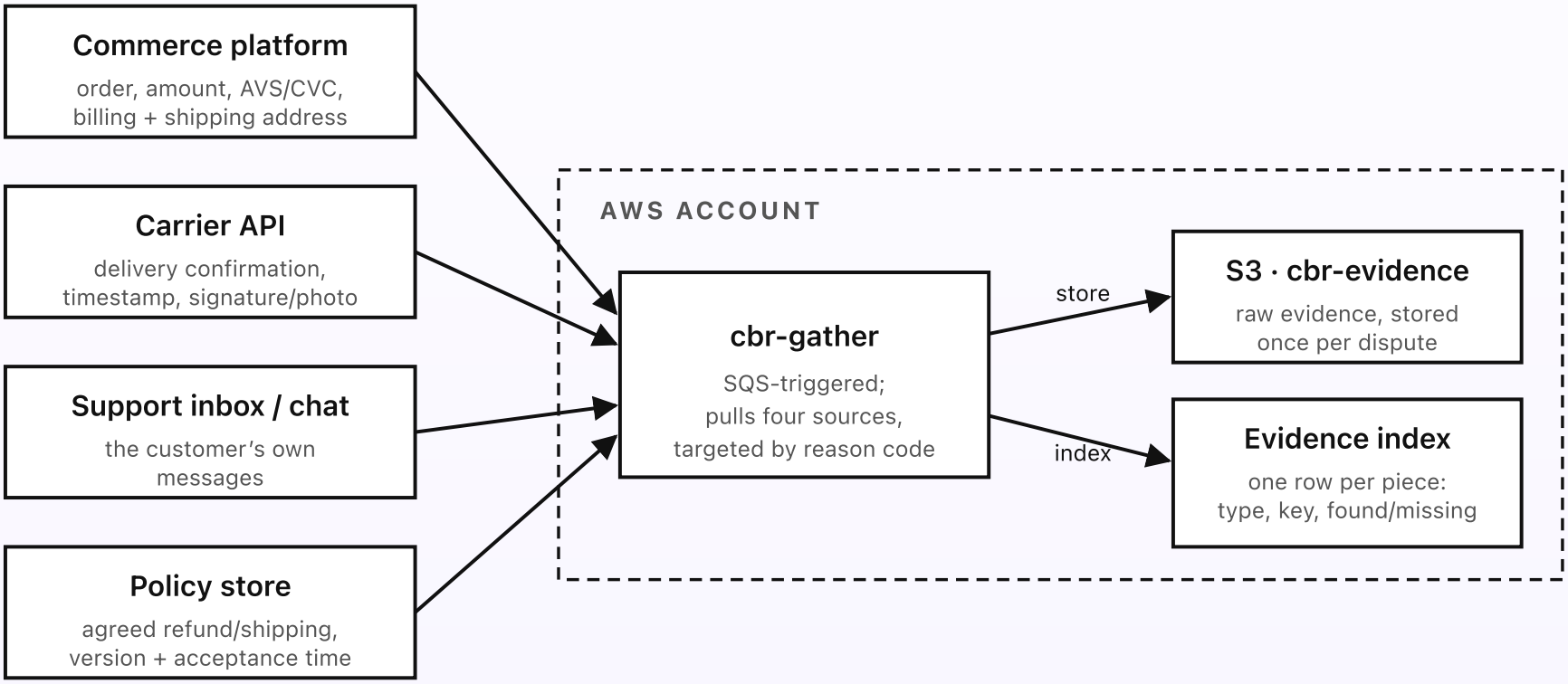
Four things, because that's what banks weigh

A card network doesn't want a folder of everything you have; it wants the few specific things that answer the dispute. Across the common reason codes, those few things come down to four. The **order record**: what was bought, when, for how much, the billing and shipping addresses, and the AVS and CVC results captured at the time of sale — this is what proves a real, authorised purchase happened. The **proof of delivery and tracking**: the carrier's confirmation that the goods reached the customer, with a timestamp and a signature or photo where one exists — this answers "I never got it." The **customer's own messages**: the support thread or chat where they asked about, used, or acknowledged the order — a customer who emailed "thanks, it arrived" has undercut their own dispute.

And the **agreed policy**: the refund, shipping, or subscription terms the customer accepted at checkout, with the version and the timestamp of their acceptance — this answers “I was charged after I cancelled” or “they never told me it was final sale.”

Targeted by reason code, not a blind sweep

The dispute’s reason code, read at the front door, tells the gather Lambda which of the four pieces carry the weight. A *product-not-received* dispute lives or dies on the delivery proof, so that source is pulled first and hardest. A *fraudulent / unauthorised* dispute turns on the order record — the AVS match, the CVC result, the shipping-to-billing address, the customer’s prior order history. A *subscription-cancelled* or *credit-not-processed* dispute turns on the agreed policy and the cancellation timestamp. The gather always collects all four where they exist (a strong packet leans on more than one), but it knows which is decisive, so it spends its effort and its retries where the case will actually be won or lost.



Targeted by the reason code. Each piece stored once. A missing piece is recorded as missing — never invented.

Fig 3. The gather step. One Lambda reads the four real sources, writes each piece once to S3 under a per-dispute prefix, and indexes it in DynamoDB — including, honestly, the pieces it couldn't find.

Store once, index every piece

Each piece the gather Lambda retrieves is written to the S3 evidence bucket under a prefix keyed by the dispute id — the order JSON, the carrier's tracking response, the exported message thread, the policy snapshot at the version the customer accepted. Storing the raw source means the packet built later is reproducible: anyone can see exactly what the rebuttal was assembled from. Alongside each S3 write, the Lambda puts one row into the evidence index in DynamoDB: the dispute id, the evidence type, the S3 key, the source it came from, the timestamp it was gathered, and a flag for whether it was found. The index is the manifest the next step reads — it never has to crawl S3 to discover what evidence exists.

Carriers and inboxes are flaky, so gather is built to retry. It runs off the SQS queue with a dead-letter queue behind it: a transient carrier-API error is retried a couple of times, and only a genuinely stuck gather lands in the DLQ with an alarm, where it surfaces to a human instead of silently blocking the case. A delivery confirmation that isn't available yet (the parcel is still in transit) isn't an error — it's recorded as not-yet-available, and the deadline watch will trigger a re-gather closer to the cutoff.

A missing piece is data, not a blank

The single most important rule in this step: when a piece of evidence doesn't exist, the system records that it doesn't exist. No delivery confirmation for a digital product. No support thread because the customer never wrote in. No signature because the carrier doesn't capture one for low-value parcels. Each gap is written to the index as *missing*, with the reason. This is what keeps the whole system honest downstream: the packet-builder is handed a truthful manifest of what's there and what isn't, so it can never be tempted — or asked — to fill a hole with something invented. A gap in the evidence becomes a known fact about the case's strength, not a silent omission.

DESIGN RULES FOR GATHERING EVIDENCE

- Four sources, every time: order, delivery proof, customer messages, agreed policy. Those are what banks weigh.
- Targeted by reason code. The decisive source for this dispute is pulled first and hardest.
- Store the raw source in S3 once, so the packet built from it is always reproducible.
- Index one row per piece, so the next step reads a manifest instead of crawling storage.
- Missing is recorded as missing. A gap is a known fact about the case, never a blank to be filled.

PART 4 OF 7

JUNE 30, 2026 PART 4 OF 7 · CHARGEBACK RESPONDER SERIES ~7 MIN READ

How an evidence packet gets built

Gathering evidence is mechanical; turning it into a rebuttal a bank will actually read is the part where the system earns its keep. This post is about how one Bedrock call assembles the packet — mapping each piece of real evidence to the precise thing the dispute's reason code is contesting, writing a tight cover narrative, and rendering it to a PDF — and, crucially, how it stays grounded: it never fabricates a tracking number or a message, and when the evidence simply isn't there it says the case is unwinnable and recommends taking the loss.

KEY TAKEAWAYS

- One Bedrock Haiku 4.5 call turns the gathered evidence and the reason code into a structured rebuttal.
- A reason-code playbook tells the model which claim each piece of evidence is meant to answer.
- The packet is rendered to a PDF in S3, with every claim citing the exact evidence it rests on.
- The model gives an honest winnability read — and recommends accepting the loss when the proof isn't there.
- It is grounded strictly in the gathered evidence: it never writes a tracking number or a message that doesn't exist.

The reason-code playbook does the thinking up front

A good rebuttal answers the *specific* thing the bank is testing, and that thing is set by the reason code. So the assemble Lambda starts from a small playbook — a plain text file in S3, editable without a deploy — that maps each common reason code to the claim it makes and the evidence that rebuts it. *Product not received* → rebut with proof of delivery to the customer's address, plus any message acknowledging receipt. *Unauthorised / fraudulent* → rebut with the AVS and CVC match, the shipping-to-billing address, the customer's device or order history. *Subscription cancelled* → rebut with the agreed terms and the cancellation timestamp showing the charge predated it. The playbook means the model isn't

guessing what a strong response looks like for this dispute — it's following the structure card networks actually score against.

One grounded Bedrock call

The Lambda reads the evidence index from the previous step, pulls the relevant pieces out of S3, and makes a single Bedrock Haiku 4.5 call. The prompt hands the model three things: the dispute (amount, reason code, what the customer claims), the playbook entry for that reason code, and the gathered evidence — the order fields, the tracking events, the message excerpts, the policy text — each tagged with its evidence-index id. The model's job is narrow and mechanical: write a tight cover narrative and a point-by-point rebuttal in which *every* claim cites the evidence id it rests on. It is told, firmly, to use only the evidence provided and to flag anything the playbook expects but the manifest marks missing. It returns structured JSON — the narrative, the cited points, the list of attached exhibits, and a winnability verdict — not free-form prose. Structured output is what lets the next step render a clean packet and lets the filer attach the right exhibits in the right fields.

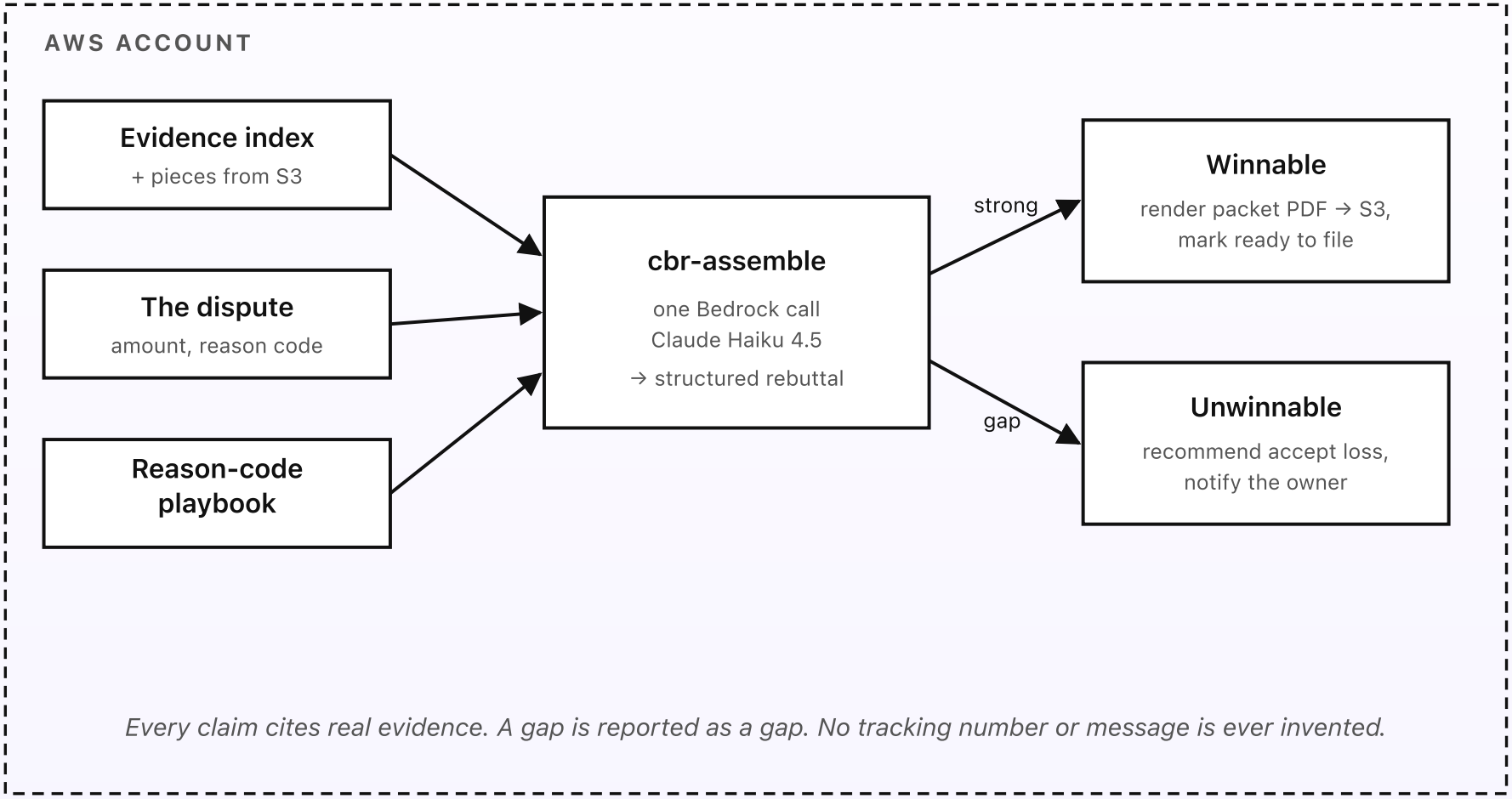

AWS ACCOUNT
Evidence index
+ pieces from S3
The dispute
amount, reason code
Reason-code playbook
cbr-assemble
one Bedrock call
Claude Haiku 4.5
→ structured rebuttal
strong
gap
Winnable
render packet PDF → S3, mark ready to file
Unwinnable
recommend accept loss, notify the owner
Every claim cites real evidence. A gap is reported as a gap. No tracking number or message is ever invented.

Fig 4. Assembly. The evidence, the dispute, and the reason-code playbook feed one grounded Bedrock call; a strong case becomes a packet PDF ready to file, a case with a decisive gap becomes an honest recommendation to accept the loss.

Render the packet to a PDF

The structured rebuttal is turned into a single PDF in the S3 evidence bucket: a cover page with the dispute reference and the cover narrative, the point-by-point rebuttal, and then the exhibits — the order confirmation, the carrier's proof of delivery, the relevant message excerpts, the accepted policy — each labelled and cross-referenced from the narrative. Rendering happens in the Lambda from a fixed template, so the layout is consistent and nothing about it depends on the model. Many processors also take evidence as discrete fields and file attachments through their API rather than one PDF; the assemble step produces both shapes from the same structured output, so the filer can submit whatever this processor expects.

An honest winnability read

The same call returns a winnability verdict — strong, mixed, or weak — with a one-line reason grounded in the manifest. Strong: the decisive evidence for this reason code is present and consistent. Mixed: the evidence is there but has a soft spot (delivered, but to a slightly different address; acknowledged, but with a complaint). Weak: the decisive piece is missing or actively unhelpful — no proof of delivery on a not-received claim, or a support thread where the customer was promised a refund that never came. On a weak case the system doesn't quietly file a hopeless packet; it writes a *recommend accept-loss* outcome and tells the

owner why. Fighting a chargeback you'll lose still costs the dispute fee and your attention, so knowing when to stand down is part of doing this well.

Why it can't fabricate

The guardrail against invented evidence isn't a polite instruction — it's the structure. The model is given the evidence manifest and told to cite an evidence id on every claim; the renderer then checks each cited id actually exists in the index before it puts the claim in the packet. A claim that cites nothing, or cites a piece marked missing, is dropped, not printed. So the worst a model drift can do is produce a *weaker* packet — never a fictional one. The customer's tracking number is the carrier's tracking number or it isn't in the packet at all; the "customer acknowledged receipt" line exists only if a real message is attached behind it. This is what makes the system safe to point at a bank: it can only ever say true things about what you have.

DESIGN RULES FOR BUILDING THE PACKET

- Start from the reason-code playbook, so the rebuttal answers the exact claim the bank is testing.
- One grounded Bedrock call, structured output, every point citing a real evidence id.
- Render from a fixed template, in both PDF and field-and-attachment form, so any processor's API can take it.
- Give an honest winnability read, and recommend accepting the loss when the decisive evidence is missing.
- Cited-evidence checks at render time mean a model can only weaken a packet, never invent one.

PART 5 OF 7

JUNE 30, 2026 PART 5 OF 7 · CHARGEBACK RESPONDER SERIES ~7 MIN READ

How a response gets filed on time

Filing the right packet is worth nothing if it lands a day late, so this post is about time. It walks through the deadline watch — the one-off scheduled job armed the moment the dispute is caught — the owner-approval and auto-file modes that decide how much happens without you, and the guard underneath both: a hard cutoff that files the assembled, real-evidence packet before the deadline so a forgotten tap can never turn a winnable case into a default loss.

KEY TAKEAWAYS

- The deadline watch is a one-off EventBridge Scheduler job, armed the moment the dispute was caught.
- Two modes: auto-file (strong packets file themselves) or owner-approval (nothing files until you tap).
- Underneath both sits a deadline guard that fires a safe margin before the bank's due-by.
- If nothing has been filed and you haven't chosen to accept the loss, the guard files the assembled packet.
- It never files late, and it never files a packet you've told it to drop — the deadline can't beat you.

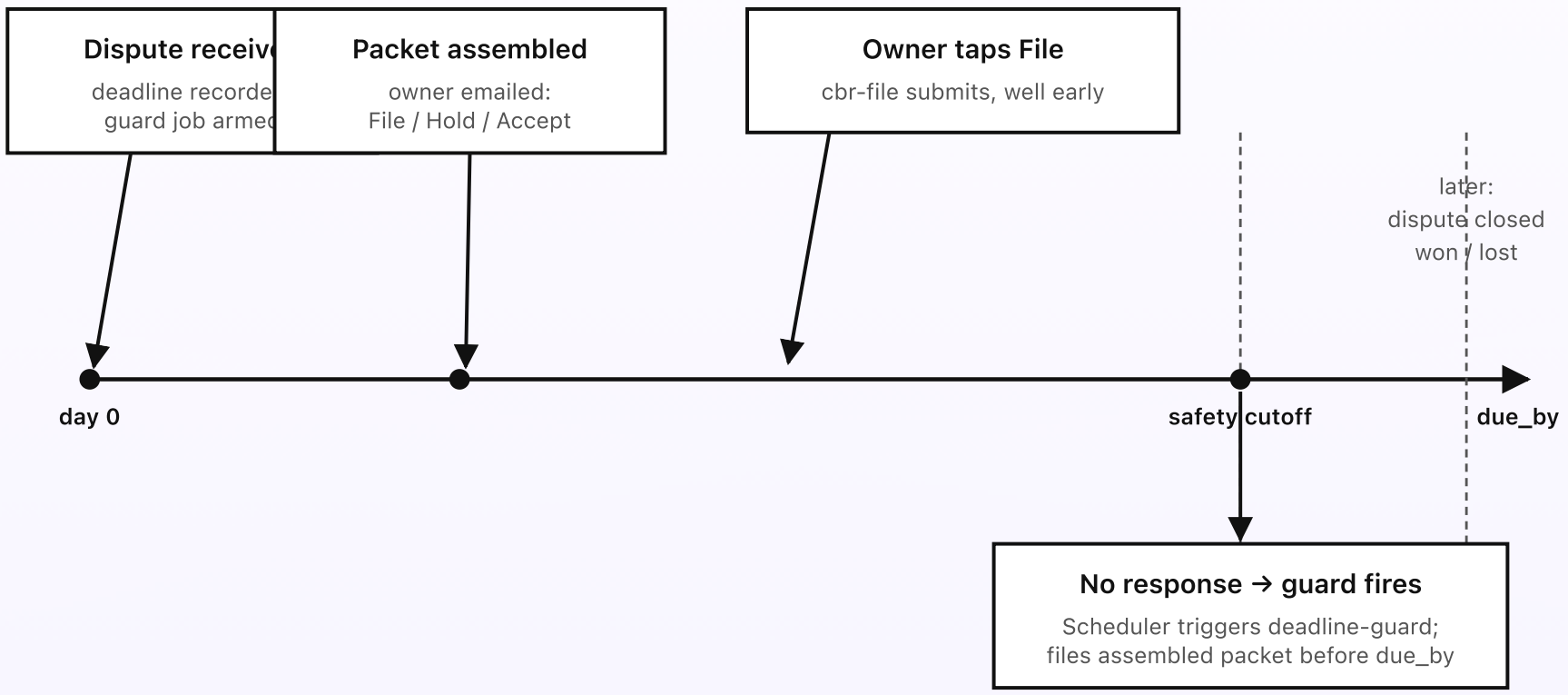
The watch that was armed at the front door

Back in Part 2, the very first thing the intake did after recording the deadline was arm a one-off EventBridge Scheduler job — an `at(...)` rule timed for a safe margin before the processor's `due_by`, targeting the deadline-guard Lambda. That job is the spine of this whole post. It exists from the second the dispute is caught, before any evidence is gathered, before any packet is built. Everything else in the filing step is about giving the owner a chance to decide — but the watch is the promise that whatever the owner does or doesn't do, the case will be filed before the cutoff or deliberately dropped, never simply forgotten.

Two modes: how much happens without you

How a case reaches the processor depends on a single setting. In **owner-approval mode**, the assemble step emails the owner a summary — amount, reason code, the winnability read, and the buttons — and nothing is filed until they tap *File*. This is the default for businesses that want a human eye on every response. In **auto-file mode**, a packet the assembler rated *strong* is filed automatically as soon as it's ready, and the owner's email becomes a heads-up rather than a gate; only *mixed* and *weak* cases wait for a decision. This suits higher volumes where most disputes are clear-cut and the owner only wants to be pulled in on the judgement calls. Either mode can be set per-account and overridden per dispute.

The owner's three buttons land on a Function URL, the same signed, single-use-token pattern used across the series so a forwarded email link can't be replayed. *File* submits the assembled packet now. *Hold* pauses for a closer look — useful when the owner wants to add a note or check something — but, crucially, Hold does not disarm the deadline guard. *Accept loss* records the decision not to fight and disarms the guard, so the case is closed cleanly and nothing files.



The guard files a winnable case before the cutoff whether or not anyone taps a button — never late.

Fig 5. The filing timeline. The deadline-guard job is armed at day zero; the owner can file early, but if no one responds the Scheduler fires the guard at the safety cutoff and files the assembled packet — always before the bank's due-by.

The deadline guard: a backstop, not a guess

When the one-off Scheduler job fires at the safety cutoff, the deadline-guard Lambda doesn't blindly submit something. It reads the case state and decides: if the dispute is already filed, it does nothing. If the owner chose *accept loss*, it does nothing — that was a deliberate decision, and the guard never overrides it. If the assembler rated the case *weak* and auto-file isn't on, it does nothing but raise a last-call alert, because filing a hopeless packet just burns the fee. But if the case is winnable, assembled, and simply un-actioned — the owner was busy, on holiday, asleep — the guard files the assembled packet. The reasoning is blunt: a real-evidence packet filed at the last safe moment beats a guaranteed default loss every time. Before the cutoff the guard also triggers a re-gather if a piece (like a delivery confirmation) was still pending earlier, so it files the freshest evidence available.

Filing through the API, exactly once

Filing itself is the `cbr-file` Lambda calling the processor's evidence-submission API with the key from Secrets Manager — either the assembled fields and attachments or the packet PDF, whichever this processor takes. Two cases must not file the same dispute twice — the owner tapping *File* a moment before the guard fires, say — so filing is guarded by an idempotency key tied to the dispute id and a conditional state transition: the case moves to `filed` only from `ready`, and a second attempt finds it already filed and stops. On a successful submit, the state, the submission id the processor returns, and the timestamp are written to the disputes table; the deadline-guard job, if it hasn't already fired, is deleted so it can't act on a closed case.

Tracking the outcome

Weeks later the processor posts `charge.dispute.closed` with the result. That webhook comes in the same verified front door, and the system records *won* or *lost* against the case — closing the loop so every dispute has a final state, and feeding the monthly summary that reports your win rate by reason code. A lost case isn't a system failure: some disputes are genuinely the customer's to win, which is exactly why the winnability read and the accept-loss path exist. What the system guarantees isn't that you win every chargeback — it's that you never lose one to a deadline you forgot.

DESIGN RULES FOR FILING ON TIME

- The deadline guard is armed at intake and is the backstop for everything — the case can't lapse.
- Auto-file or owner-approval is a setting; in both, a winnable case still gets filed before the cutoff.
- Hold doesn't disarm the guard; only an explicit Accept-loss does. Inaction never becomes a silent loss.
- Filing is idempotent — a conditional state transition means a dispute is never submitted twice.
- The guard never files a case marked accept-loss, and never files a hopeless packet just to file something.

PART 6 OF 7

JUNE 30, 2026 PART 6 OF 7 · CHARGEBACK RESPONDER SERIES ~6 MIN READ

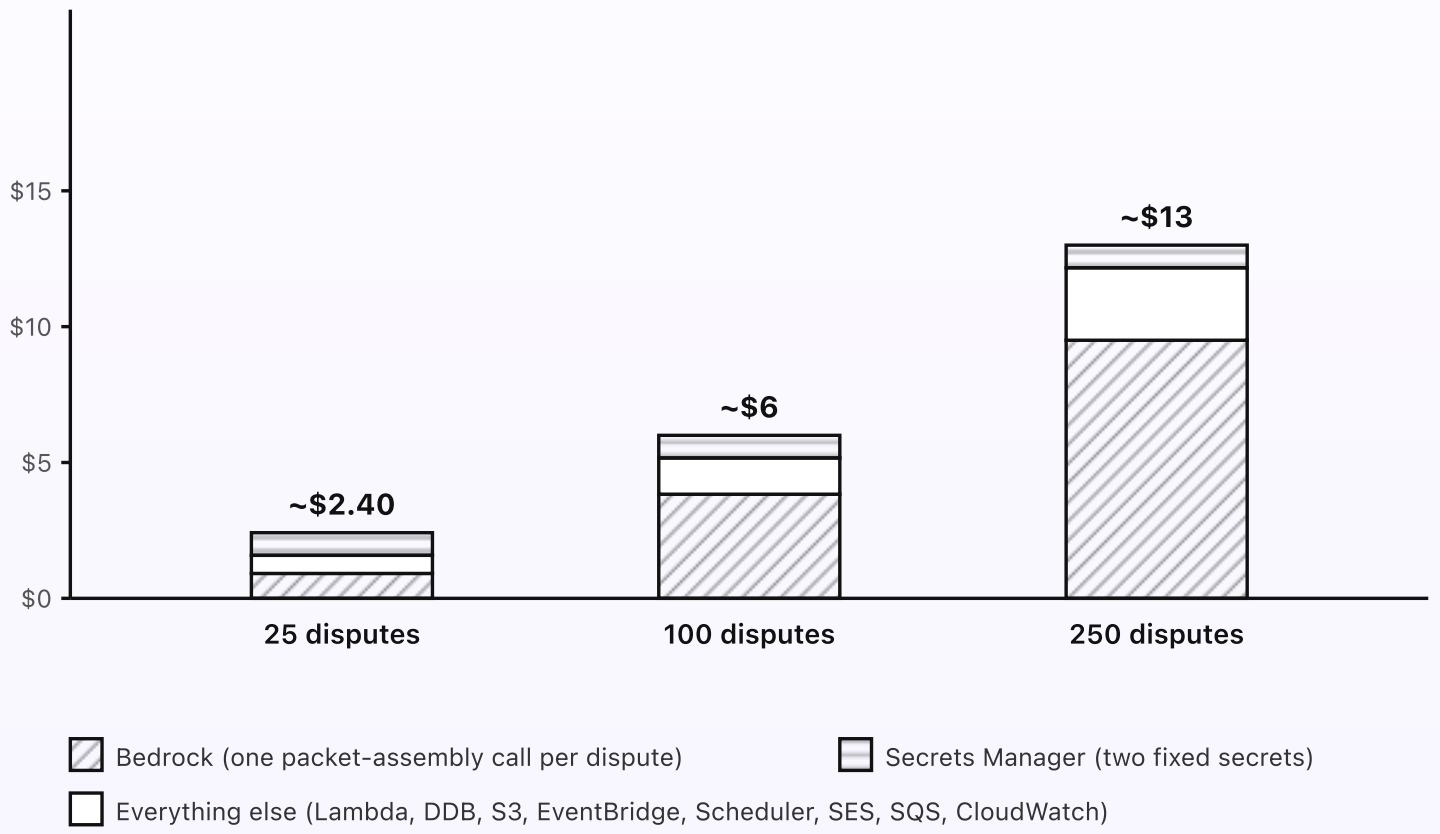
What the chargeback responder costs

This is one of the cheapest systems in the series, and the reason is worth stating plainly: catching the dispute, gathering the evidence, watching the deadline, and filing the response are all but free — they're plain Python, a few API calls, and some storage. The whole bill is one variable line, the Bedrock call that assembles each packet, sitting on top of two tiny fixed costs. This post breaks down exactly where the $2.40 a month goes, what changes at ten times the volume, and why the maths is lopsided in your favour.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (about 25 disputes a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- Catching, gathering, watching, and filing are all but free — plain code, a few API calls, a little storage.
- The one real variable is the Bedrock call that assembles each packet; everything else rounds to pennies.
- At 100 disputes it's around $6; at 250 it's around $13. One recovered order pays for a year of it.

Cost at three volumes



Assembling the packet is the only cost that grows — catching, gathering, watching, and filing are essentially free.

Fig 6. Monthly cost at three dispute volumes. Only the Bedrock slice grows with the number of disputes; the fixed Secrets Manager cost stays flat and everything else rounds to pennies.

The breakdown at ~25 disputes a month

Component	What it does	~/month
Bedrock (Haiku 4.5)	One packet-assembly call per dispute	$0.95
Secrets Manager	Two secrets — processor key, carrier key — at $0.40 each	$0.80
Lambda	webhook, gather, assemble, file, guard, ack, summary	$0.20
DynamoDB on-demand	disputes, evidence index, deadlines	$0.10
S3	Gathered evidence + packet PDFs	$0.10
SES + SQS + EventBridge	Owner alerts, work queue, dispute events, Scheduler	$0.10
CloudWatch + Budgets	Logs at 7-day retention, one cost alarm	$0.15
Total		**~$2.40**

Where the dollars actually go

Bedrock (the bulk). Each dispute gets one Haiku 4.5 call to assemble the rebuttal: the prompt carries the dispute, the reason-code playbook entry, and the

gathered evidence — a few thousand input tokens — and returns the structured packet, a couple of thousand output tokens. At Haiku pricing that's a few cents per dispute. Across 25 disputes it's under a dollar. The monthly summary adds one larger call — a narrative of disputes received, value defended, and win rate by reason code — for a couple of cents. This is the only line that grows with volume, because it's the only thing done once per dispute that isn't plain arithmetic.

Secrets Manager (the fixed floor). Two secrets — the processor API key and the carrier API key — at $0.40 each. This is the largest *fixed* cost in the system, which tells you how cheap everything else is. It doesn't grow with disputes; it's the same $0.80 whether you handle five disputes or five hundred.

Lambda runtime. The webhook, gather, assemble, file, deadline-guard, ack-handler, and summary functions are all small and short. The webhook answers in well under a second; gather spends most of its time waiting on the carrier and the inbox; the guard runs at most once per dispute. The Lambda total lands around $0.20 at this volume — arm64 and tight memory sizing keep it there.

DynamoDB on-demand. Three small tables — `cbr-disputes`, `cbr-evidence`, `cbr-deadlines`. A handful of writes per dispute and a few reads. Pennies a month at any SMB volume.

S3 and storage. The raw gathered evidence and the packet PDFs — a few MB per dispute, a few hundred MB a year. Effectively free, even with versioning on and a multi-year lifecycle for the audit trail.

SES, SQS, EventBridge, CloudWatch. Outbound owner alerts at $0.10 per thousand; the work queue with its dead-letter queue; the dispute-event rule and the one-off Scheduler jobs; logs at 7-day retention. Each is cents a month here.

What doesn't cost money

- **API Gateway.** Replaced by two Lambda Function URLs — one for the webhook, one for the file/hold buttons.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Everything is event- or schedule-driven.
- **A scheduled poller.** Disputes arrive by webhook, so there's nothing to poll. The only schedule is the one-off deadline guard per dispute.
- **Models on the decisions.** Whether to file and the deadline maths are plain code. Bedrock fires only to assemble the rebuttal and write the monthly summary.

How the cost scales — and why the maths is lopsided

Only Bedrock grows with dispute count, and it grows linearly: a few cents per dispute. So 100 disputes a month lands around $6, 250 around $13, and 1,000 around $40 — at which point you'd look at batching the summary and caching playbook entries, but the assembly call itself stays the cost. Everything else is flat or rounds to nothing.

Now weigh that against what a chargeback costs. A single disputed order is the amount itself — £85, £200, whatever it was — plus a non-refundable dispute fee of £15–20 that you pay win or lose. Recover one £85 order a month and the system has paid for itself roughly thirty times over. Set an AWS Budgets alarm at $25/month so anything unusual pages you, and the normal-volume bill sits

comfortably underneath it. The cheapest part of this system is the system; the expensive part is the chargebacks you were already losing.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, DynamoDB schemas, the SES setup, the EventBridge rule, and the Scheduler deadline-watch config.

PART 7 OF 7

JUNE 30, 2026 PART 7 OF 7 · CHARGEBACK RESPONDER SERIES ~12 MIN READ

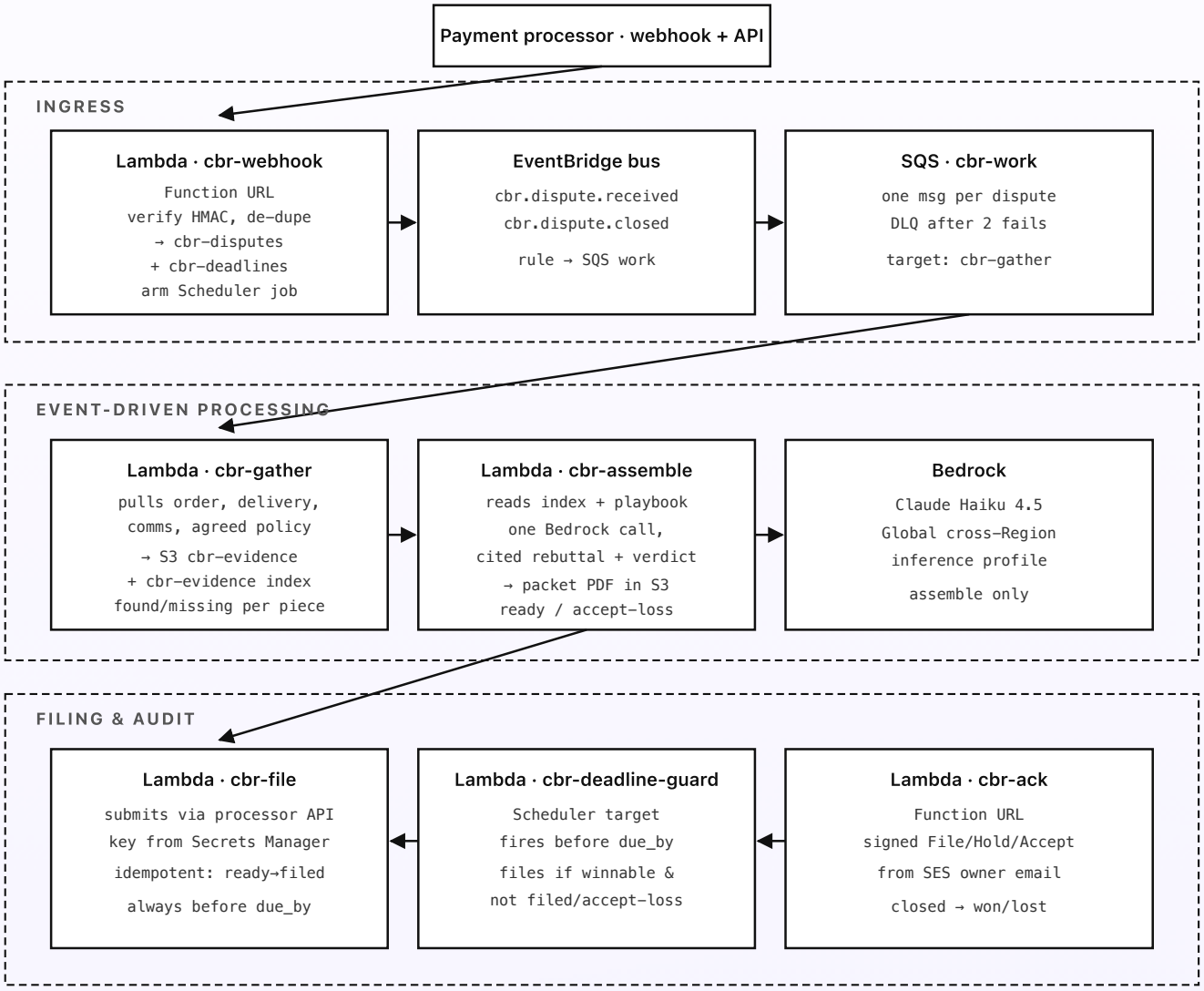
Engineering reference: the chargeback responder architecture

Same system, drawn for engineers. Region, service names, resource identifiers, the Bedrock model id, the Lambda inventory, IAM scopes, the SES configuration, the two Function URLs, the EventBridge dispute rule and the one-off Scheduler deadline watch, and the DynamoDB schemas for the disputes, the evidence index, and the deadlines. Read it alongside the previous six posts — this one is the build sheet.

Region and account shape

Default region: **eu-west-2** (London). Lambda Function URLs, EventBridge Scheduler, SES outbound, DynamoDB, S3, and Bedrock cross-Region inference are all in good shape there, and it keeps the data close to a UK or EU customer base. A second region for multi-region resilience isn't worth the extra work at SMB volume — the failure mode for a chargeback responder is a single dispute taking an hour longer to file, and the deadline guard already gives a day of slack, so a regional blip never causes a missed deadline. One AWS account dedicated to the responder keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



The deadline guard files a winnable case before the cutoff — and nothing in the packet is ever fabricated.

Fig 7. AWS topology, in three regions of the diagram: ingress (the verified webhook arming the deadline watch), event-driven processing (gather then assemble, with Bedrock used only to assemble), and filing and audit (the filer, the deadline guard backstop, and the owner's buttons). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

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

- **`cbr-webhook`** — Lambda Function URL, `AuthType: NONE`; authenticated by the processor's signature. Verifies the HMAC over the raw body using the signing secret from Secrets Manager and checks the timestamp window. On `charge.dispute.created`: de-duplicates with a conditional `PutItem` on `cbr-disputes`, records `due_by` and the reason code in `cbr-deadlines`, creates a one-off EventBridge Scheduler rule targeting `cbr-deadline-guard`, emits `cbr.dispute.received`, and returns `200`. On `charge.dispute.closed`: records won/lost on the case and deletes any remaining Scheduler rule. Memory: 256 MB. Timeout: 15 s.
- **`cbr-gather`** — SQS trigger on `cbr-work`. Reads the dispute, then pulls the four sources targeted by the reason code: the order from the commerce platform, proof of delivery from the carrier API (key in Secrets Manager), the customer's messages from the support inbox/chat, and the accepted policy snapshot. Writes each raw piece to `s3://cbr-evidence/<dispute_id>/` and an index row to the `cbr-evidence` table with a found/missing flag; a not-yet-

available delivery confirmation is recorded for re-gather, not treated as an error. Enqueues assembly. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*

- **`cbr-assemble`** — triggered after gather. Reads the evidence index and the reason-code playbook from `s3://cbr-evidence/playbook.txt`, pulls the cited pieces from S3, and makes one Bedrock Haiku 4.5 call (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) returning structured JSON: cover narrative, cited rebuttal points, exhibit list, winnability verdict. Drops any claim whose cited evidence id is absent or marked missing, renders the packet PDF (and the field-and-attachment form) to S3, and sets the case `ready` or `recommend_accept_loss`. Emails the owner via SES. In auto-file mode, a `strong` case is handed straight to `cbr-file`. Memory: 1024 MB. Timeout: 60 s.
- **`cbr-file`** — submits the assembled evidence through the processor's evidence API using the key from Secrets Manager. Idempotent: a conditional update moves the case from `ready` to `filed` only once, so the owner's tap and the guard can never double-file. Writes the processor's submission id and timestamp to `cbr-disputes` and deletes the Scheduler rule. Memory: 256 MB. Timeout: 30 s.
- **`cbr-deadline-guard`** — EventBridge Scheduler target, one per dispute, fired a safe margin (default 24 h) before `due_by`. Reads case state: if `filed` or `accept_loss`, does nothing; if `weak` and auto-file off, raises a last-call alert only; otherwise triggers a re-gather of any pending piece and calls `cbr-file` on the assembled packet. Memory: 256 MB. Timeout: 60 s.
- **`cbr-ack`** — Lambda Function URL, `AuthType: NONE`; verifies a signed, single-use token (HMAC over `(dispute_id, action, nonce, expiry)` keyed from

Secrets Manager) so a forwarded email link can't be replayed. On *File*: calls `cbr-file`. On *Hold*: sets `held` without disarming the guard. On *Accept loss*: sets `accept_loss` and deletes the Scheduler rule. Always writes the action to `cbr-disputes`. Memory: 256 MB. Timeout: 15 s.

- **`cbr-summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's disputes and outcomes; calls Bedrock Haiku 4.5 to write a one-paragraph narrative (disputes received, value defended, win rate by reason code, recommended-accept-loss count); emails it via SES to the owner. Memory: 512 MB.

Storage

- **DynamoDB · `cbr-disputes`** — one row per dispute. PK `dispute_id`; attributes: `processor`, `charge_id`, `reason_code`, `amount`, `currency`, `customer`, `state` (received/gathering/assembled/ready/held/filed/accept_loss/won/lost), `winnability`, `submission_id`, `created_at`, `filed_at`. On-demand.
- **DynamoDB · `cbr-evidence`** — the evidence index, one row per gathered piece. PK `dispute_id`; SK `evidence_type` (order/delivery/comms/policy); attributes: `s3_key`, `source`, `gathered_at`, `status` (found/missing/pending), `note`. On-demand.
- **DynamoDB · `cbr-deadlines`** — one row per dispute deadline. PK `dispute_id`; attributes: `due_by`, `safety_cutoff`, `schedule_arn`, `filed` (bool). On-demand. The authoritative record the guard checks against.
- **S3 · `cbr-evidence`** — one prefix per dispute holding the raw gathered evidence (order JSON, carrier tracking, message exports, policy snapshot) and

the compiled packet PDF, plus the editable `playbook.txt` at the bucket root. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years for the audit trail.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `cbr-assemble` for the rebuttal, and `cbr-summary` for the monthly narrative. Neither `cbr-webhook`, `cbr-gather`, `cbr-file`, nor `cbr-deadline-guard` ever calls Bedrock — every money or deadline decision is deterministic.
- **Grounding.** The assemble prompt carries only the gathered evidence with its index ids; the renderer drops any claim citing an absent or missing id. The model cannot introduce evidence, only arrange what was gathered.
- **Heavier model.** `anthropic.claude-sonnet-4-6-...` is configured but off by default; worth switching `cbr-assemble` to Sonnet only for high-value disputes where a more careful narrative is justified, via a per-dispute amount threshold.
- **Embeddings.** Not used. There's no corpus to search — each dispute's evidence is gathered fresh. No Knowledge Base, no S3 Vectors.

EventBridge and Scheduler config

- **Dispute rule** — an EventBridge rule on the default bus matching `cbr.dispute.received`, target the `cbr-work` SQS queue (which triggers `cbr-gather`).

- **Deadline watch** — one-off Scheduler rules created by `cbr-webhook`, one per dispute, with an `at(YYYY-MM-DDTHH:MM:SS)` expression set to `due_by` minus the safety margin, target `cbr-deadline-guard`, and `--action-after-completion DELETE` so the rule self-cleans. Deleted early by `cbr-file` or `cbr-ack` when the case is filed or accept-loss.
- **`cbr-monthly-summary`** — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `cbr-summary` Lambda.

SES and the two Function URLs

- **SES outbound only.** Verify a sender identity at `disputes@your-company.com` with DKIM and SPF on the parent domain; out of sandbox by request. Used for the per-dispute owner alert and the monthly summary. There is no SES inbound — disputes arrive by webhook, not email.
- **Webhook Function URL** — `cbr-webhook`, public, signature-checked inside the function. This is the address registered in the processor's webhook settings.
- **Ack Function URL** — `cbr-ack`, public, signed single-use token in each button link. The owner's email renders File, Hold, and Accept-loss as links to this URL.

IAM (least privilege per Lambda)

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

- **cbr-webhook role:** `dynamodb:PutItem` / `UpdateItem` on `cbr-disputes` and `cbr-deadlines`; `scheduler:CreateSchedule` / `DeleteSchedule`;

`events:PutEvents` on the default bus; `secretsmanager:GetSecretValue` on the webhook signing secret. *No* `bedrock:*`.

- **cbr-gather role:** `sqs:ReceiveMessage` / `DeleteMessage` on `cbr-work`; `s3:PutObject` on `cbr-evidence`; `dynamodb:PutItem` on the `cbr-evidence` table; `secretsmanager:GetSecretValue` on the carrier key; outbound network to the carrier, commerce, and inbox hosts. *No* `bedrock:*`.
- **cbr-assemble role:** `s3:GetObject` / `PutObject` on `cbr-evidence`; `dynamodb:Query` on the evidence index; `bedrock:InvokeModel` on the Haiku ARN; `ses:SendEmail` from the verified sender; `secretsmanager:GetSecretValue` on the ack-link signing secret.
- **cbr-file role:** `dynamodb:UpdateItem` on `cbr-disputes` and `cbr-deadlines`; `scheduler:DeleteSchedule`; `s3:GetObject` on the packet; `secretsmanager:GetSecretValue` on the processor API key.
- **cbr-deadline-guard role:** the `cbr-file` permissions plus `dynamodb:GetItem` on the deadlines and disputes tables, and permission to re-invoke `cbr-gather`.
- **cbr-ack role:** `dynamodb:UpdateItem` on `cbr-disputes`; `scheduler:DeleteSchedule`; permission to invoke `cbr-file`; `secretsmanager:GetSecretValue` on the ack-link signing secret.

Idempotency and the deadline invariant

Two invariants hold the system together. First, *a dispute is filed at most once*: filing is a conditional state transition from `ready` to `filed`, so the owner's tap, an auto-file, and the deadline guard all race safely — whoever wins moves the state, the others find it already filed and stop. Second, *a winnable dispute is never*

filed late: the one-off Scheduler rule is armed before any slow work runs and is only deleted when the case is genuinely filed or deliberately dropped, so there is no path where a case is forgotten. The guard fires on a timer that AWS owns, independent of whether any earlier step succeeded.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a metric for alerting.
- **SQS DLQ:** `cbr-work` has a dead-letter queue; a gather that fails twice lands there with an alarm, so a flaky carrier API never silently blocks a case from being assembled in time.
- **Alarms:** any case within 12 h of `due_by` still in `ready` (the guard should have it, but page anyway); DLQ depth > 0; `cbr-file` failures > 0; `cbr-webhook` signature-verification failures > 5/hour.
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `cbr-cost-alarm` subscribed to the owner's email.

Config and secrets

The processor API key and webhook signing secret live in Secrets Manager under `cbr/processor/*`; the carrier API key under `cbr/carrier/key`; the ack-link signing secret under `cbr/links/secret`. The mode (auto-file vs owner-approval), the safety-margin hours, the high-value Sonnet threshold, and the

owner's email live in Parameter Store under `/cbr/config/`. The reason-code playbook is plain text in S3 so it can be tuned without a deploy. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) and AWS SAM. The opinionated bits: turn on S3 versioning for `cbr-evidence` so a packet is always reproducible and a bad playbook edit rolls back in one click; give `cbr-gather` an SQS source with a DLQ so one unreachable carrier never wedges the pipeline; and keep the deadline guard's schedule creation in `cbr-webhook` so the safety net is armed at the earliest possible moment. Total deployable surface: seven Lambdas, three DynamoDB tables, one S3 bucket, one SQS queue with a DLQ, one EventBridge rule on the default bus (plus the per-dispute Scheduler rules), two Function URLs, an SES sender identity, and one Budgets alarm.

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