7-PART SERIES · FREE COMPANION

Refund handler

A serverless handler that reads each refund request, checks it against your own written refund policy, drafts a clear and kind reply, and routes the easy ones for one-tap approval — while sending anything out of policy or high-value straight to a human. It never issues money on its own; a person approves every refund. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Refund handler

PART 1 OF 7

MAY 17, 2026 PART 1 OF 7 · REFUND HANDLER SERIES ~5 MIN READ

A refund handler on AWS for a few dollars a month

Refund requests are some of the most stressful email a small business gets. The customer is already unhappy. The person answering is often busy, or new, or both — and the policy lives in a doc nobody has read in months. So replies come out slow, inconsistent, and sometimes wrong: a refund given that the policy says no to, or one refused that the policy clearly allows. This post walks through the design of a small handler that reads each request, checks it against your own written policy, drafts a calm and kind reply, and hands the easy ones to a person for a single tap. It never sends money on its own.

KEY TAKEAWAYS

- Three sources feed one queue: a help inbox, a contact-form webhook, and a manual paste lane.
- Every request ends in one of four outcomes: in policy, out of policy, high-value, or not covered.
- The checker only acts on what your policy actually says — it cites the line it used.
- Nothing sends money on its own. A person approves every refund with one tap.
- Designed on AWS for about $3/month at typical small-business volume.

The whole system on one page

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

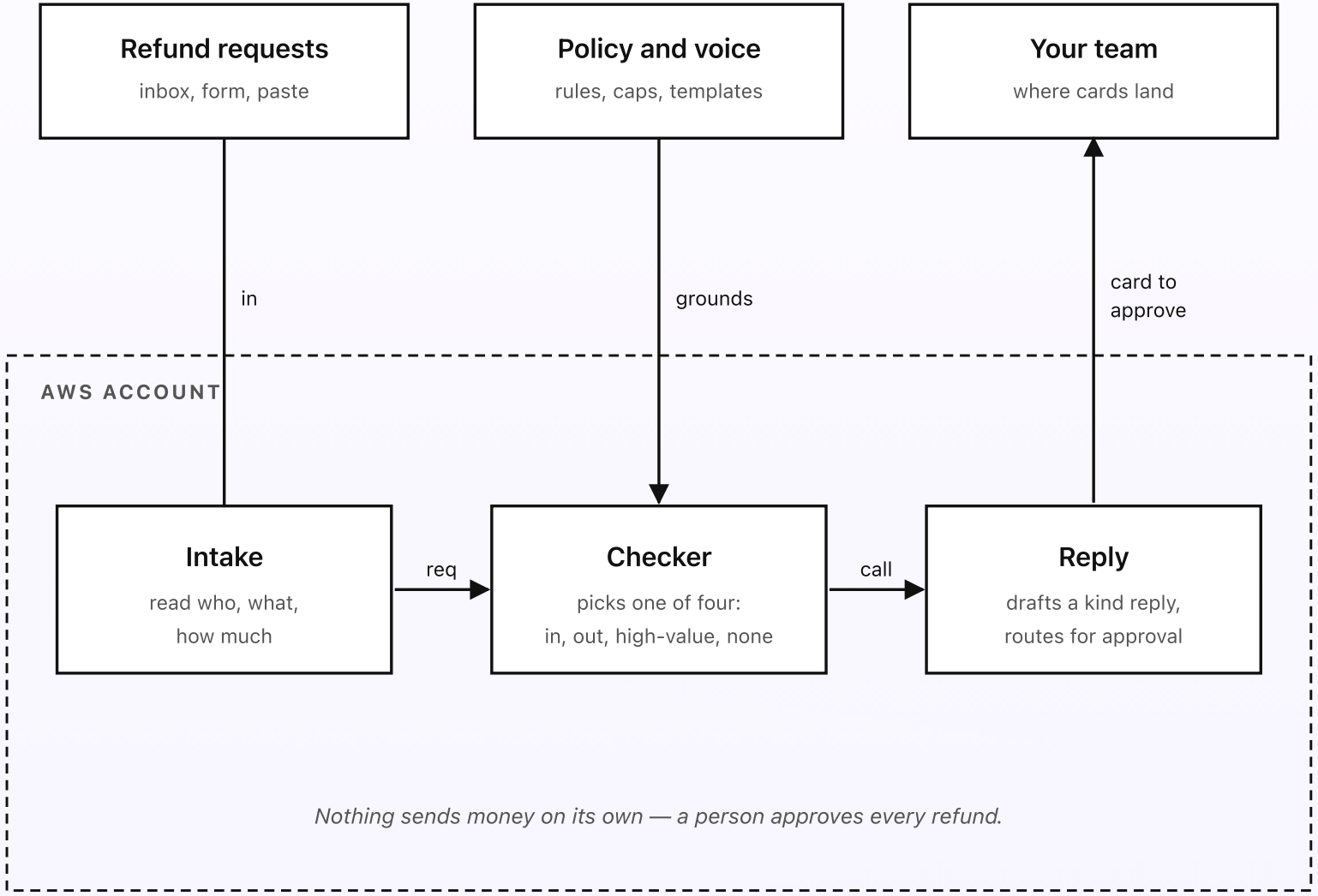

Refund requests
inbox, form, paste
Policy and voice
rules, caps, templates
Your team
where cards land
in
grounds
card to approve
AWS ACCOUNT
Intake
read who, what, how much
req
Checker
picks one of four: in, out, high-value, none
call
Reply
drafts a kind reply, routes for approval
Nothing sends money on its own — a person approves every refund.

Fig 1. Three sources outside, three pieces inside AWS. Requests flow in from a help inbox, a contact-form webhook, and a manual paste lane. The Checker reads each one against your policy and picks one of four outcomes. The Reply piece drafts a kind answer and routes it to a person to approve.

What you set up once (the outside)

- **Refund requests.** Three ways a request gets in. A help inbox — customers email or your team forwards an email to a dedicated address. A contact-form webhook — the "request a refund" form on your site posts straight to the handler. And a manual paste lane — for the request that came in by phone or live chat, a rep pastes it into a small form so it gets the same treatment as everything else. All three are covered in Part 2.
- **A policy folder.** Two short Google Docs in a Drive folder. The *policy* doc is your refund policy in plain prose — the return window, the conditions (unused, original packaging, proof of purchase), the exceptions (final-sale items, custom orders), and any dollar caps ("refunds over $200 need a manager"). The *voice* doc holds the tone and a reply template per outcome — what a kind "yes," a clear "no," and a "we need a bit more" actually sound like in your words.
- **Your team.** The people who approve refunds. Each gets a card in Slack (or email if Slack isn't set up) with the original request, the policy line that applies, the drafted reply, the dollar amount, and an "Approve" button. Low-value, clearly in-policy cases get the one-tap card. Out-of-policy or high-value cases go to a named approver with the reason attached.

What runs on every request (the inside)

- **The intake.** Three sources feed one queue. The intake reads each incoming request — an email, a form post, or a pasted note — and pulls out the parts that matter: who the customer is, what they bought, the order or receipt reference, how much, and what they're asking for. It writes a clean request record and drops it on the queue. Bedrock Haiku 4.5 (a small, cheap model) does the reading; the rest is plain code.
- **The checker.** The heart of the system. For each request, it pulls the exact policy passages that apply — not the whole policy, just the lines about this kind of case — and decides only from them. *In policy*: the policy clearly allows it and it's under the cap — route to a one-tap approve. *Out of policy*: the policy clearly says no — draft a kind decline that quotes the rule, route to a person to confirm. *High-value*: allowed, but over the dollar cap — route to a named approver. *Not covered*: nothing in the policy fits — route to a human to decide, never guessed at. The checker cites the policy line behind every decision.
- **The reply.** Reads the voice doc, drafts a reply for the chosen outcome in your tone, and quotes the policy line the checker used so the customer sees the "why," not just the answer. The draft is attached to the approval card — it is never sent on its own. On approve, the reply goes out via SES and the request is marked resolved. A weekly digest summarizes what came in, what was approved, and what's still waiting. Every action is logged so a refund can be read back months later.

In plain words

A customer emails: "The blender I bought three weeks ago stopped working. I'd like a refund." The handler reads it: customer Dana, order #4821, blender, $89,

asking for a refund, 21 days since delivery. The checker pulls the policy lines about the 30-day window and the "faulty item" exception, sees the request is inside the window and the item is reported faulty, and lands on *in policy*. It drafts: "Hi Dana — sorry the blender stopped working. Our policy covers faulty items within 30 days, so we'll refund the $89 to your original payment. You'll see it in 5–7 days…" with the policy line quoted underneath. A card lands in the support Slack with that draft and an Approve button. Sam taps Approve. The reply goes out. The whole thing took Sam four seconds, and the answer was right because it came from the actual policy.

The cost of running this is about $3 a month at SMB volume. The cost of *not* running it is the refund email that sat for two days and turned a fixable problem into a one-star review — or the refund that got approved against policy because nobody had time to check.

DESIGN RULES THAT SHAPED EVERY DECISION

- Nothing sends money on its own. A person approves every refund — this rule has no exceptions.
- Four outcomes, always. In policy, out of policy, high-value, not covered. There is no fifth.
- Every decision cites the policy line it came from. No answer without a "why."
- Not covered goes to a human. The checker never guesses at a case the policy doesn't address.
- The policy lives in Drive. Changing a window, a cap, or an exception doesn't need a deploy.
- Every request and decision is logged. Audit a refund next year and you can see exactly why it went out.

Why this shape

Most teams handle refunds in one of three ways: whoever opens the email decides on the spot, a manager has to be pulled in for every one, or there's a policy doc that's technically the rule but nobody reads it under pressure. The first is fast but inconsistent — two customers with the same case get different answers depending on who replied. The second is consistent but slow, and burns a manager's day. The third is the false comfort of having a policy that doesn't actually shape the replies going out.

The setup above keeps the policy where the team already edits it, but adds a small system that *reads* that policy on every single request and drafts the reply from it. The easy cases — which are most of them — become a one-tap approve. The hard cases get flagged early, with the policy line and the reason attached, so the person deciding has what they need. And because the money never moves without a human tap, the system can be helpful without ever being dangerous.

The next four posts walk through each piece in turn: how a refund request arrives, how it gets checked against policy, how a reply gets drafted, and how a refund gets approved. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 17, 2026 PART 2 OF 7 · REFUND HANDLER SERIES ~4 MIN READ

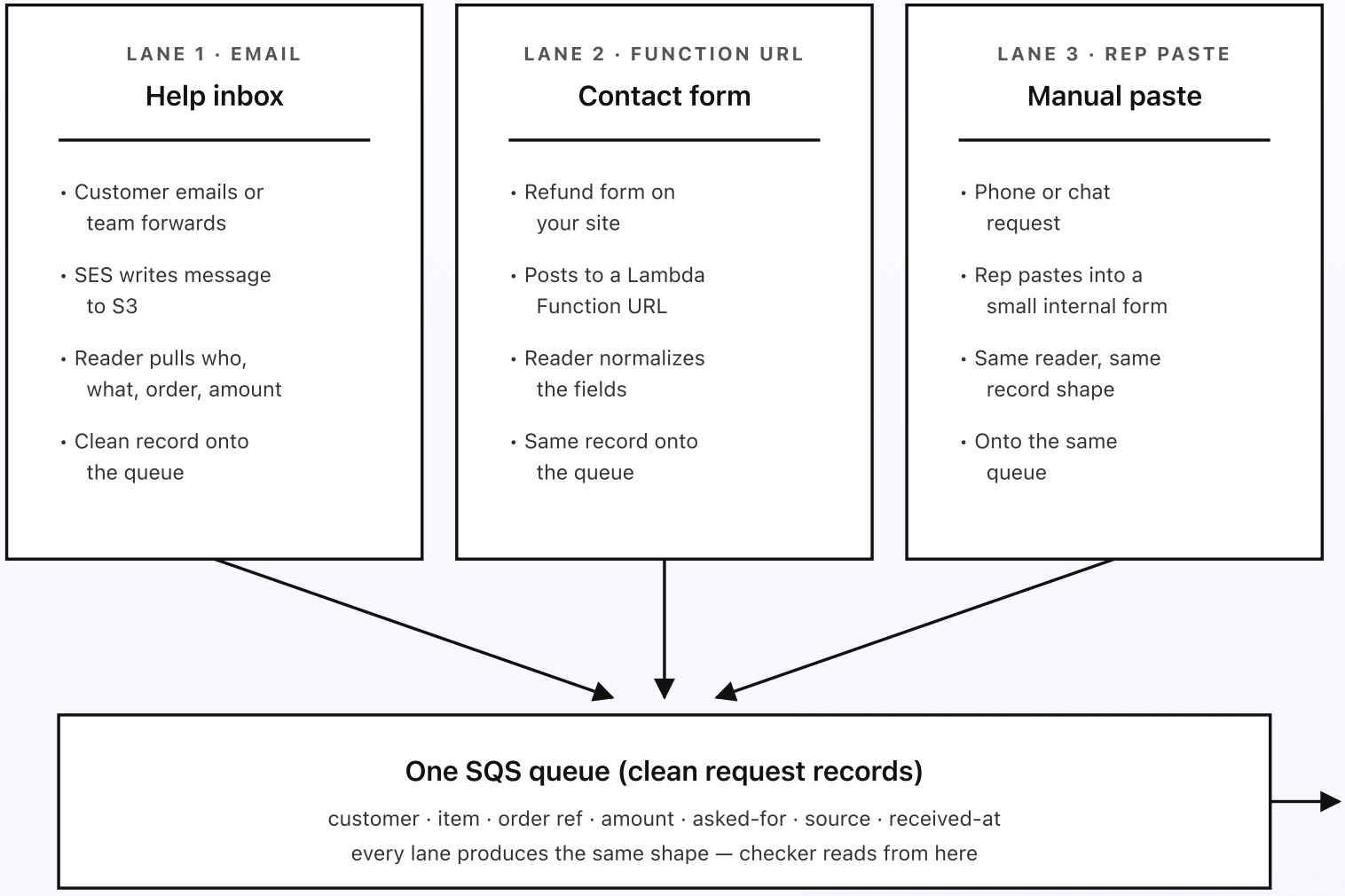
How a refund request arrives

The handler can only help with requests it actually sees. So the first job is catching every refund request, no matter how it came in. There are three ways one gets into the queue: a customer emails (or your team forwards an email) to a help address, someone fills in the refund form on your site, or a rep pastes in a request that arrived by phone or live chat. The first two are automatic. The third exists because in real life plenty of refund asks never start as an email at all.

KEY TAKEAWAYS

- Three intake lanes feed one queue: a help inbox, a contact-form webhook, and a manual paste lane.
- Inbound emails arrive via SES; a small reader pulls out who, what, order, and amount.
- The contact form posts straight to a Lambda Function URL — no API Gateway.
- The paste lane lets a rep drop a phone or chat request into the same flow.
- All three land as one clean request record on a single SQS queue for the checker.

Three lanes into one queue



Every lane produces the same record shape — the checker never has to care how a request arrived.

Fig 2. Three lanes converge on one queue. Email, web form, and rep paste all produce the same clean request record. The reader does the work of turning each into the same shape, so the checker downstream only ever sees one kind of thing.

Lane 1: the help inbox

The most common lane. Set up a dedicated inbound address — something like `refunds@your-company.com` — via Amazon SES. Customers email it directly, or your team forwards the original refund email to it. SES writes the raw message to `s3://rf-raw-mime/`. The S3 write triggers a reader Lambda that walks the message, finds the customer's text (and any forwarded original underneath it), and calls Bedrock Haiku 4.5 — a small, fast, cheap model — to pull out the parts that matter: the customer's name and email, the item, the order or receipt reference if it's mentioned, the amount, and what they're actually asking for (full refund, partial, replacement).

The model prompt is short and strict: "Read this refund request. Return JSON only with these fields. If a field isn't in the text, leave it blank — do not guess an order number or an amount." A blank field is fine; a made-up order number is not, because it would send the wrong refund down the wrong path. The cleaned record goes onto the SQS queue, and the checker picks it up from there.

Lane 2: the contact form

Plenty of businesses already have a "request a refund" form on their site. Lane 2 wires that form straight into the handler. The form posts to a Lambda Function URL — a plain HTTPS endpoint on a single Lambda, with no API Gateway in front of it (which keeps the cost at zero when the form is quiet). The form already has

structured fields — name, email, order number, reason — so the reader has less to pull out; it just normalizes them into the same record shape and drops it on the queue.

The Function URL checks a shared secret the form includes, and a small rate limit keeps a bot from flooding the queue. Because the fields are already separate, this is the cleanest lane — the request record is almost complete before any model touches it.

Lane 3: manual paste

Some refund asks never arrive as text you control. The customer called. They mentioned it in a live chat that closed. They told a rep in the shop. Forcing those into “please email us so the system sees it” is a fight you don’t need — the rep is right there and can just write it down.

Lane 3 is a small internal form: the rep pastes or types what the customer said, adds the order number if they have it, and submits. It posts to the same Function URL as Lane 2 with a flag marking the source as “paste,” and the same reader turns it into the same record. Marking the source matters later — the audit trail shows a pasted request came from a person, not from a verified email address, so the approver knows to double-check the order reference.

Why everything funnels to one queue

Three lanes in, but only one queue the checker reads from. That’s on purpose. If each lane had its own path through the system, every “why did this refund get approved?” question would mean checking three different flows. Funneling

everything into one queue with one record shape means there is exactly one path a request can take, and the checker, the drafter, and the audit log all see the same thing. The queue also gives you a safety net: if the checker has a bad moment, requests wait safely in the queue (and a dead-letter queue catches anything that fails repeatedly) instead of getting lost.

Next post: how the checker reads a request, pulls the exact policy lines that apply, and picks one of four outcomes — grounded only in what your policy actually says.

PART 3 OF 7

MAY 17, 2026 PART 3 OF 7 · REFUND HANDLER SERIES ~5 MIN READ

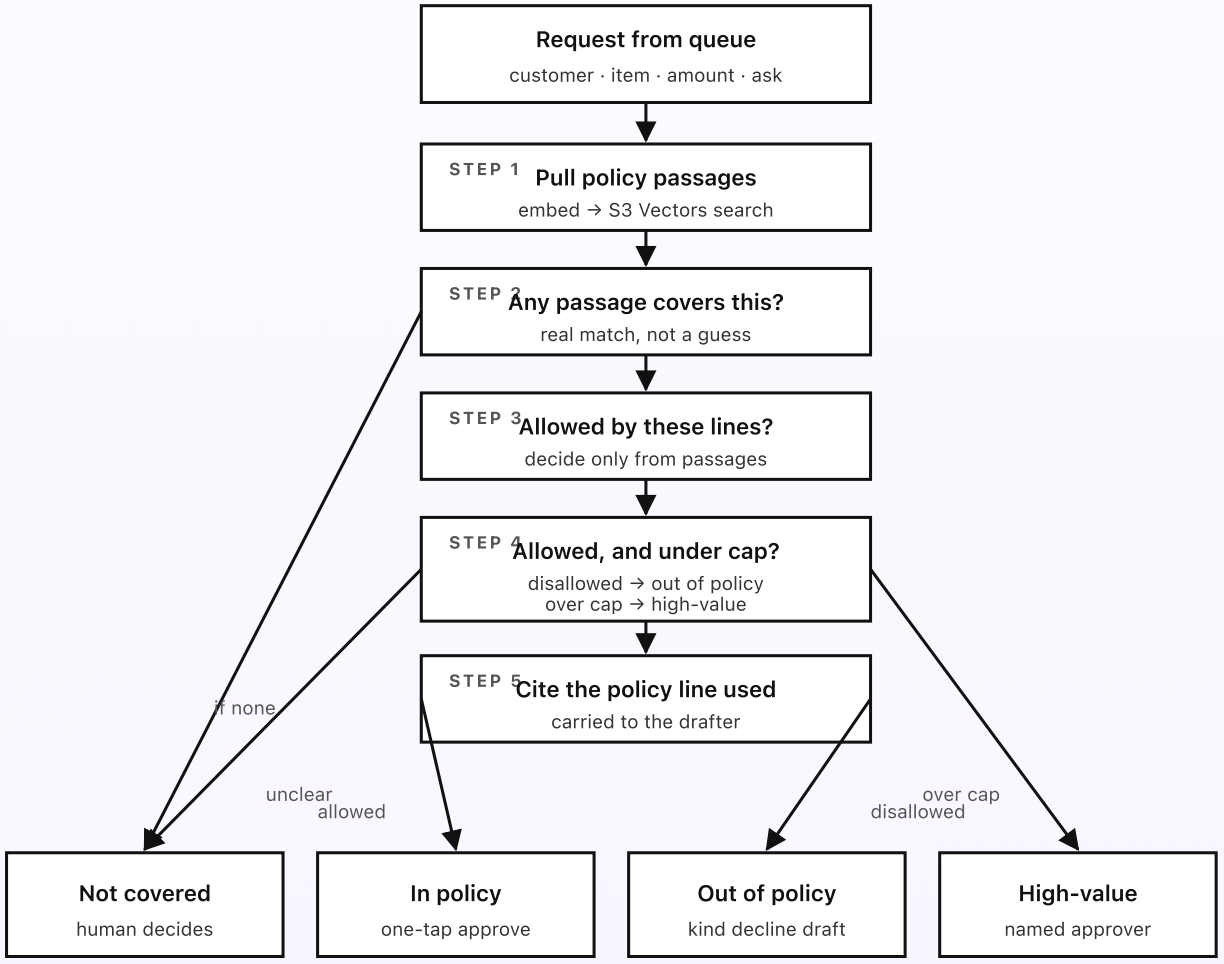
How a refund gets checked against policy

This is the heart of the system. A request comes off the queue, and the checker has to decide what your policy says about it. The trick is keeping that decision honest: the checker only acts on what the policy actually says, and it says so by quoting the line it used. To do that it pulls the few policy passages that apply to this exact case, decides only from them, and refuses to guess when nothing fits. The result is always one of four outcomes.

KEY TAKEAWAYS

- The policy lives in a Drive doc; a sync job indexes it into Amazon S3 Vectors for fast lookup.
- For each request, the checker pulls only the policy passages that apply — not the whole doc.
- It decides only from those passages and cites the exact line behind every decision.
- Four outcomes per request: in policy, out of policy, high-value, or not covered.
- If nothing in the policy fits, the request is marked not-covered and sent to a human — never guessed.

The decision flow, per request



The policy doc holds every rule — change a line and the next request uses the new rule.

Fig 3. The checker's decision tree, per request. Five steps decide which of four outcomes applies. The policy doc holds every rule; the checker only reads it — and cites the line behind every decision.

Grounding: the checker reads your policy, not its own memory

The danger with any model deciding refunds is that it answers from a general idea of what refund policies usually look like, rather than from yours. A "14-day window" policy gets treated like the "30-day" one the model saw more often in training, and a customer gets the wrong answer. Grounding fixes this. Your policy lives in a Drive doc you write in plain prose. A small `policy-sync` job mirrors it to S3 every fifteen minutes and breaks it into short passages — one per rule or exception. Each passage is turned into a row of numbers (an "embedding," a math fingerprint of the meaning) by Titan Text Embeddings V2 and stored in Amazon S3 Vectors, a cheap store built for exactly this kind of lookup.

When a request comes in, the checker turns the request into the same kind of fingerprint and asks S3 Vectors for the closest few policy passages. Those — and only those — are handed to the model. So the model is never deciding from memory; it's deciding from the exact lines of your policy that match this case. Change a window from 30 days to 14 in the doc, and within fifteen minutes the next request is judged by 14.

Four outcomes, always

Every request, every time, lands in exactly one of four buckets. The names are plain on purpose.

- **In policy.** The pulled passages clearly allow the refund and the amount is under the cap. This is the easy lane — route a one-tap approve card with the draft attached. Most requests, most days, land here.
- **Out of policy.** A passage clearly disallows it — the item is past the window, or it's a final-sale item. Draft a kind decline that quotes the exact rule, and route it to a person to confirm before it goes out. A "no" still gets a human tap, because a wrongly-refused refund is its own kind of damage.
- **High-value.** The policy allows it, but the amount is over the dollar cap ("refunds over $200 need a manager"). Route to the named approver in the policy, not the one-tap lane. Same draft, higher bar.
- **Not covered.** No passage really matches — the case is genuinely new (a goodwill ask, a shipping mishap the policy never anticipated). Mark it not-covered and send it straight to a human. The checker never invents a rule to cover a gap.

Cheap on the easy cases, careful on the hard ones

Most requests are clear — in the window, normal item, normal amount. Those run on Claude Haiku 4.5, a small fast model, because the decision is easy once the right policy passages are in front of it. A few requests are genuinely tricky: conflicting passages, a partial-refund judgment call, an exception inside an exception. For those, the checker escalates the single decision to Claude Sonnet 4.6, a stronger model, and pays a few cents more for a better read. The split keeps

the bill low while still giving the hard cases the care they need. Either way the rule is the same: decide only from the pulled passages, and cite the line.

Why every decision carries a citation

The cited policy line isn't decoration. It does three jobs. It lets the approver sanity-check the decision in two seconds — "yes, that's the right rule." It becomes the "why" the drafter quotes to the customer, so the reply explains itself. And it goes into the audit log, so a refund questioned six months later can be traced back to the exact policy line that justified it. A decision without a citation is just an opinion; a decision with one is something the whole team can stand behind.

Next post: how the reply gets drafted — kind, plain, in your voice, quoting the policy line — and the four guardrails between the draft and the person who approves it.

PART 4 OF 7

MAY 17, 2026 PART 4 OF 7 · REFUND HANDLER SERIES ~5 MIN READ

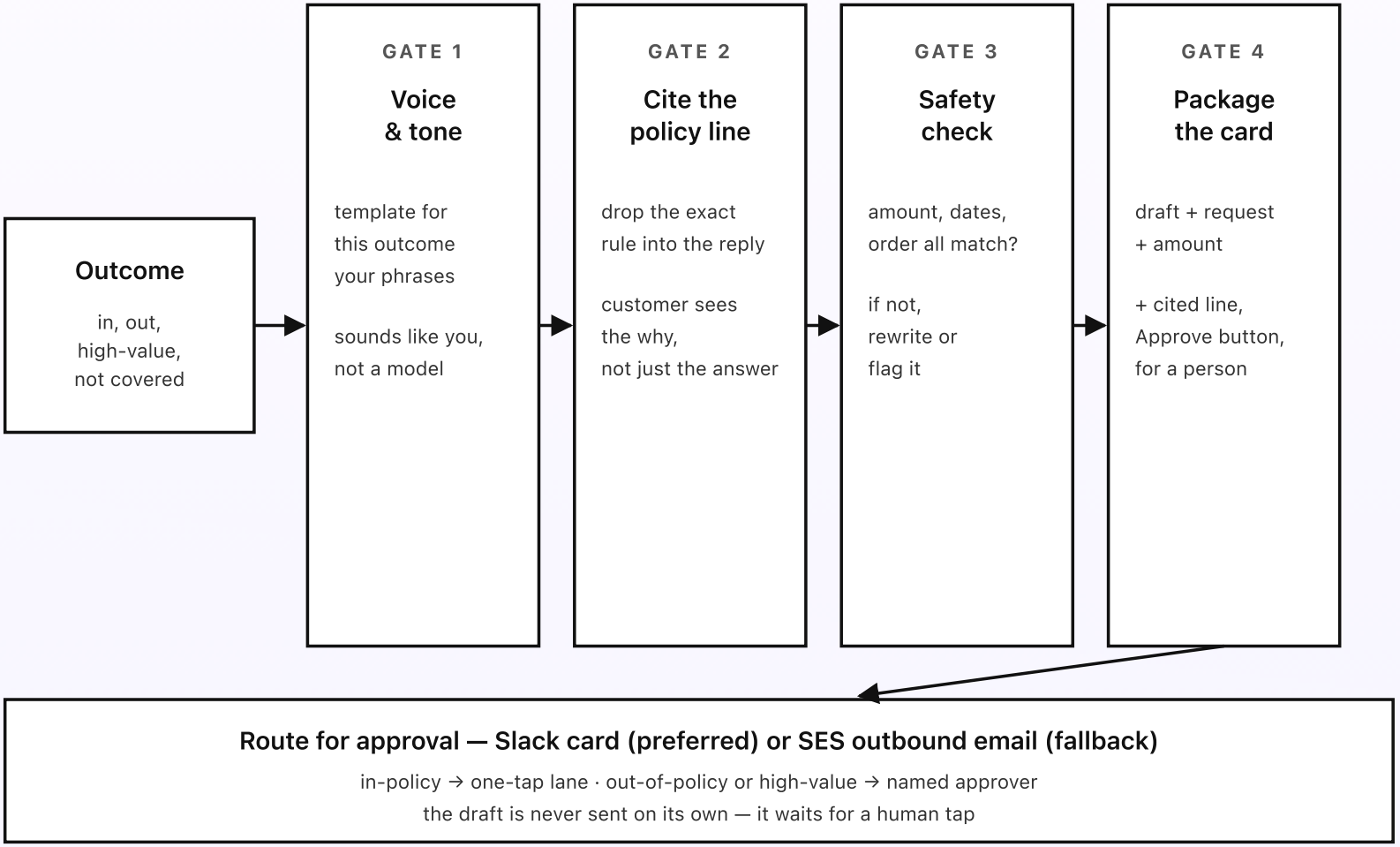
How a refund reply gets drafted

The checker picked an outcome — in policy, out of policy, high-value, or not covered. Now the drafter has to turn that into something a real person would be glad to receive: kind, clear, in your voice, and honest about the “why.” Get the tone wrong and a fair decision still lands badly. Get the facts wrong and you’ve promised a refund the policy didn’t. Four small guardrails sit between the raw outcome and the draft that reaches a person to approve.

KEY TAKEAWAYS

- The draft is written from the voice doc — your tone, your phrases, one template per outcome.
- The reply quotes the exact policy line the checker cited, so the customer sees the “why.”
- A safety check makes sure the draft doesn’t promise more than the outcome allows.
- The draft is never sent on its own — it’s attached to a card for a person to approve.
- Out-of-policy and high-value drafts are written with extra care and the reason up front.

Four guardrails on every draft



Each gate is a deterministic check — and the draft never leaves without a person approving it.

Fig 4. Four guardrails between the outcome and the drafted reply. Write in your voice. Quote the policy line. Check the facts. Package the card. Then route it to a person — the draft is never sent on its own.

Gate 1: write in your voice

The voice doc in your Drive folder has one short reply template per outcome and a few notes on tone — how warm, how formal, whether you sign off with a first name. A “yes” template opens with a quick acknowledgment of the problem and lands on the good news. A “no” template leads with empathy and explains the rule plainly. The drafter fills the template with this request’s details and rewrites lightly so it reads like a person wrote it, not a form. The point of the template is that the structure is yours; the model only fills and smooths.

This is where a generic system fails. A reply that says “We regret to inform you that your request does not meet our criteria” is technically correct and completely cold. The voice doc is how you make sure the “no” still sounds like your shop.

Gate 2: quote the policy line

Every draft includes the exact policy line the checker cited, in plain words the customer can read. A “yes” says which rule covers them (“our policy covers faulty items within 30 days”). A “no” quotes the rule that applies (“final-sale items can’t be returned, which is noted on the product page”). Showing the reason does two things: it makes the answer feel fair rather than arbitrary, and it heads off the follow-up email asking “why?” before it’s sent. The reason is the same line that’s in the audit log, so the customer, the approver, and the record all agree.

Gate 3: the safety check

This gate is plain code, not a model. It re-reads the finished draft against the request and the outcome and checks a short list of things that must be true. The refund amount in the draft matches the request and doesn't exceed it. A draft for an "out of policy" outcome doesn't accidentally say "yes." No order number, date, or dollar figure appears that wasn't in the request. If any check fails, the draft is sent back for one rewrite, and if it fails again it's flagged for a human with a note about what looked off. Models are good writers and occasionally careless with numbers; this gate is the seatbelt.

Gate 4: package the card

The last gate assembles the approval card: the drafted reply, the original request, the amount, the cited policy line, and the buttons a person uses to act. In-policy, under-cap cases get the simple one-tap card — *Approve* or *Edit*. Out-of-policy and high-value cases get the same card but routed to the named approver in the policy, with the reason up front so the harder decision is easy to make. Not-covered cases get a card with no draft — just the request and a note that the policy doesn't address it, so a person can decide from scratch.

The card goes to Slack by default (a Slack message with action buttons is faster to act on than an email) and falls back to email if Slack isn't set up. Either way, the draft sits on the card and waits. It is never sent on its own — which is the whole point of the system, and the subject of the next post.

Why the guardrails exist

None of these gates are clever. They're the small care a thoughtful support lead would take writing the reply themselves — sound like us, explain the reason, double-check the numbers, and never hit send without a second look. Putting them in code as four small steps makes that care part of every reply, instead of something you're trusting a busy person to remember at 4:55pm on a Friday.

Next post: how a refund gets approved — what the Approve and Edit buttons actually do, how the money only moves after a human tap, and how every action is logged.

PART 5 OF 7

MAY 17, 2026 PART 5 OF 7 · REFUND HANDLER SERIES ~5 MIN READ

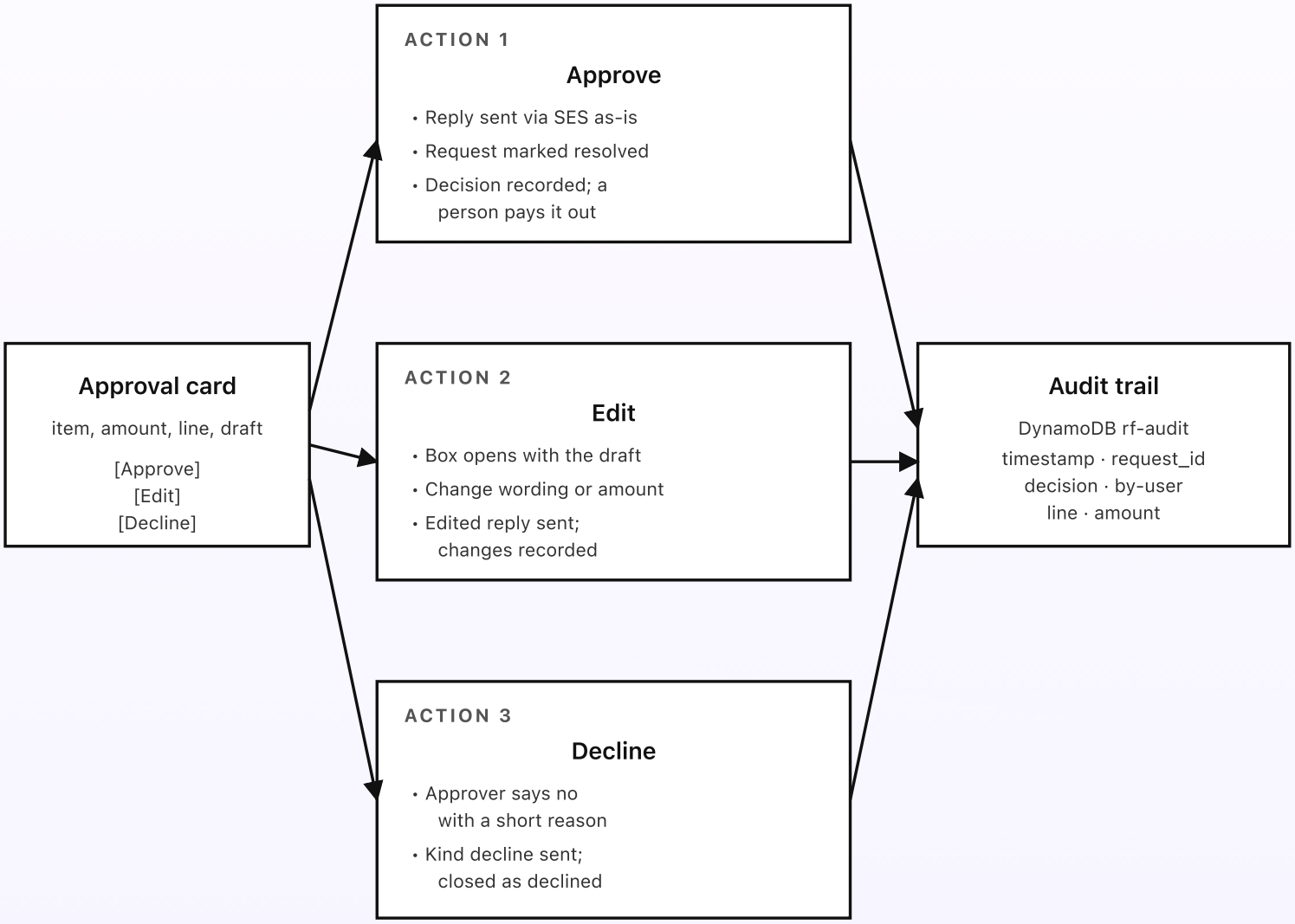
How a refund gets approved

A card lands in Sam's Slack at 9:14am. A customer wants $89 back on a faulty blender, the policy covers it, and there's a drafted reply with an Approve button. What happens when Sam taps it? And just as important — what happens when Sam doesn't agree with the draft, or decides the answer should be no? This post walks through the three things a person can do on a card — approve, edit, decline — and how the reply, the money, and the audit trail all stay in sync. The one rule underneath all of it: the money only moves after a human tap.

KEY TAKEAWAYS

- Three actions per card: *approve* (send the draft as-is), *edit* (change it first), *decline* (say no, with a reason).
- The reply goes out via SES; nothing is sent before a person taps.
- The refund itself is never issued by the system — approve records the decision for a person to pay out.
- Every action writes an audit row: who, when, the decision, the cited policy line, the amount.
- The Approve button is a Slack interactive message backed by a Function URL.

Three actions on the card



The money only moves after a human tap — approve records the decision, a person pays it out.

Fig 5. Three actions per card, three different effects. Approve sends the draft as-is and records the decision. Edit lets the approver change it first. Decline says no with a reason. Every action writes to the audit trail, and the system never moves money on its own.

Action 1: approve (the most common)

Sam reads the card, agrees, and taps *Approve*. The tap goes to a Function URL Lambda — `approve-handler` — which does three things in order. First, it sends the drafted reply to the customer via SES, exactly as written. Second, it marks the request resolved in DynamoDB and posts the decision where your finance step expects it — a row that says "refund $89 on order #4821, approved by Sam." Third, it writes an `action: approved` row to `rf-audit` with the user, the timestamp, the cited policy line, and the amount.

One thing it does *not* do is move money. The system records that a refund was approved; a person (or your existing payments step, with its own controls) issues the actual refund. That separation is deliberate — the handler is allowed to draft, send replies, and record decisions, but issuing money is a step that always stays in human hands. A bug in the handler can, at worst, send a too-kind email; it can never empty the till.

Action 2: edit (the small fix)

Sometimes the draft is 90% right. The amount should be a partial, not a full. The customer mentioned a detail the reply should acknowledge. The tone needs a touch more warmth for a regular. Sam taps *Edit*, and a box opens with the draft

already in it. Sam changes what needs changing — including the amount, if the refund should differ from what was asked — and sends.

The edited reply goes out via SES, and the change is recorded against the original draft in `rf-audit`: what the draft said, what Sam changed it to, and the new amount. Recording the edit matters — it's how you later see whether the drafts are usually good (rarely edited) or whether the policy doc needs a tune-up (often edited the same way). A pattern of identical edits is a signal that a rule should be rewritten, not re-edited every time.

Action 3: decline (the "no")

Sometimes the answer is no — either because the checker already marked it out-of-policy and Sam is confirming, or because Sam is overriding an in-policy draft for a reason the policy doesn't capture (a customer who's clearly gaming the window, say). *Decline* asks for a short reason, sends the kind decline reply, and closes the request as declined.

The required reason isn't bureaucracy — it's the audit trail's memory. A declined refund is exactly the kind of decision a customer might push back on, and "declined by Sam on May 19, reason: outside 30-day window, item used" is what lets anyone later see it was a fair call. An override of an in-policy draft is logged a little more loudly — it shows up in the weekly digest — because a manager probably wants to know when the system said yes and a human said no.

Every action is logged, every refund is traceable

The `rf-audit` table records every approve, edit, and decline with the user, the timestamp, the cited policy line, the amount, and a snapshot of the draft before and after any edit. Months later, when someone asks “why did we refund this?” or “why didn’t we?”, the answer is one lookup: the request, the policy line that applied, the person who decided, and what they sent. There’s no “I think Sam handled that one” — the record is exact.

This kind of traceability is what makes the speed safe. The handler can make refunds feel instant for the easy cases without anyone losing sight of why each one went out. Fast and accountable aren’t a trade-off here; the audit log is what lets you have both.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the heavy model barely shows up on the bill.

PART 6 OF 7

MAY 17, 2026 PART 6 OF 7 · REFUND HANDLER SERIES ~3 MIN READ

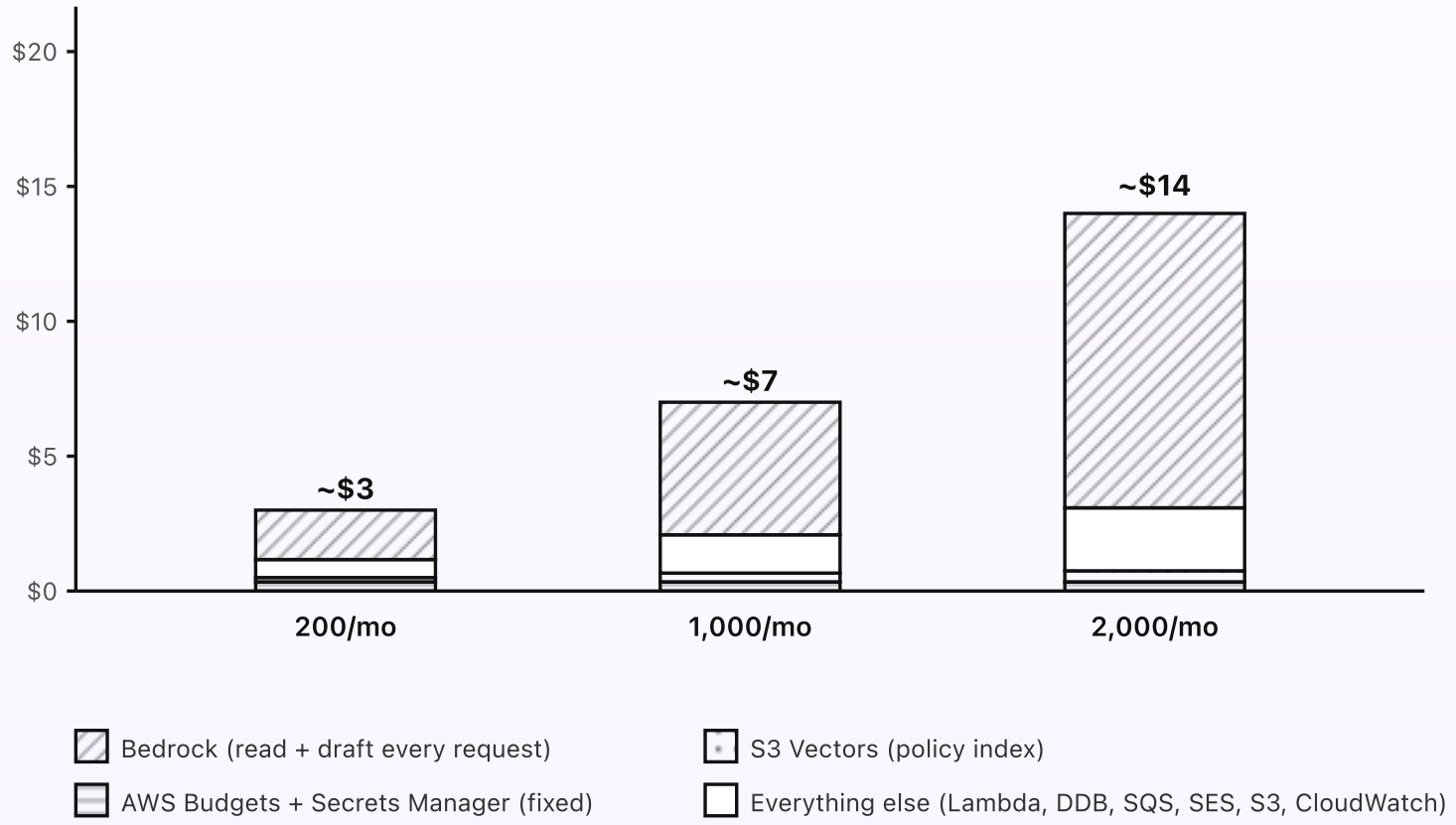
What the refund handler costs

The refund handler only does work when a request comes in. Each one reads a short email, looks up a few policy lines, drafts a reply, and posts a card to Slack. There's no daily job grinding through a big list, no always-on server. The biggest line on the bill is the model calls — reading and drafting — and even those are cheap because most cases run on a small model. At typical SMB volume, the whole thing is a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- About $3/month at typical SMB volume (around 200 requests a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The model calls are the biggest slice — reading each request and drafting each reply.
- Most cases run on the cheap model; the heavier model fires only on the few hard ones.
- At 1,000 requests the bill is around $7. At 2,000 requests it's around $14.

Cost at three volumes



The model calls are the dominant cost — and they only happen when a real request comes in.

Fig 6. Monthly cost at three request volumes. Bedrock is the dominant slice because the model reads and drafts on every request. The policy index and the fixed amounts stay tiny. The bill grows with request count, not with anything always-on.

Where the dollars actually go

Bedrock (the bulk). Two model calls per request in the common case: one small read to pull out who/what/how-much, and one small draft to write the reply — both on Claude Haiku 4.5. Each is a few thousand input tokens (the request plus a handful of policy passages) and a few hundred output tokens, so a fraction of a cent per request. The few genuinely hard cases escalate the single decision to Claude Sonnet 4.6 at a few cents each; at maybe one in twenty requests, that adds up to a small amount. Across all requests, Bedrock is the largest line on the bill — and it's still measured in single-digit dollars.

S3 Vectors (the policy index). Your policy is a short doc — a few dozen passages. Storing those as vectors and searching them on each request costs cents a month. It re-indexes only when you edit the policy, which is rare. Effectively a rounding error.

Lambda runtime. The intake reader, the checker, the drafter, the approve-handler, and the policy-sync job. Each fires only on a real request (or every 15 minutes for the tiny policy sync). Milliseconds each. Pennies a month at any of these volumes.

DynamoDB on-demand. Two small tables: `rf-requests` (the live state of each request) and `rf-audit` (the permanent record). A few reads and writes per request. Pennies.

SQS. One queue plus a dead-letter queue. The first million requests a month are free; an SMB never gets close. Effectively free.

SES. Inbound for the help-inbox lane and outbound for the approved replies: $0.10 per thousand messages each way. A couple of cents a month at SMB volume.

S3 + storage. The raw inbound messages and the mirrored policy doc. A few hundred KB. Effectively free.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the form webhook and the approve button.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Everything fires only on a real request.
- **A big knowledge base.** The policy is a short doc, so the vector index is tiny — cents, not dollars.
- **The heavy model on every request.** Claude Sonnet 4.6 runs only on the few hard cases; most requests use the cheap model.

How the cost scales

Bedrock grows roughly with request count, because the model reads and drafts on every request. Everything else grows slowly or not at all — the policy index doesn't care how many requests come in, and the fixed costs are fixed. So the bill at 5,000 requests a month is around $32; at 10,000 it's around $62. The main lever if volume climbs is the cheap/heavy split: keep more cases on Haiku 4.5 by

tightening the policy passages, and the per-request cost drops. Past those volumes you'd also batch the read and the draft into a single call — an optimization, not a redesign.

Set an AWS Budgets alarm at $15/month so anything unusual — a runaway loop, a spam flood on the form — pages you before the bill matters. The handler's normal-volume bill stays under that ceiling for a typical SMB.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, the S3 Vectors index, Lambda inventory, IAM scopes, DynamoDB schemas, and the SES rule set.

PART 7 OF 7

MAY 17, 2026 PART 7 OF 7 · REFUND HANDLER SERIES ~8 MIN READ

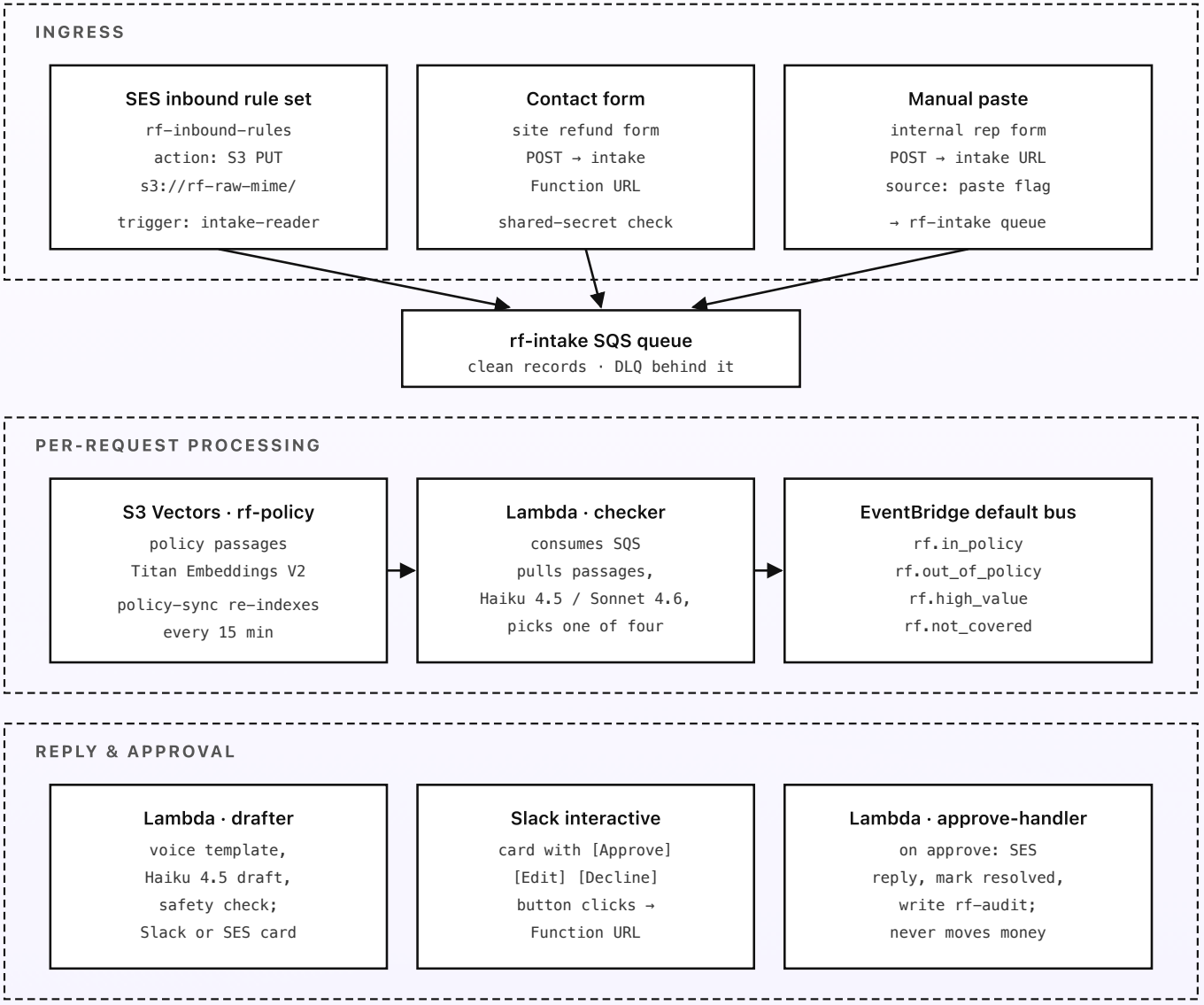
Engineering reference: the refund handler architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, the S3 Vectors policy index, Lambda inventory, IAM scopes, the SES inbound rule set, the DynamoDB schemas, and the Slack interactive flow. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound, Bedrock Global cross-Region inference, S3 Vectors, and Lambda Function URLs are all available there. A second region for resilience isn't worth the extra work at SMB volume — the failure mode for an SMB is a refund email sitting unanswered, which the SQS queue and DLQ already protect against, not a regional outage. One AWS account dedicated to the handler (separate from other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. Because the handler can send replies and record decisions but never moves money, the account never needs payment credentials at all.

Topology



Nothing sends money on its own — and every interaction is logged to rf-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into one queue), per-request processing (the checker grounds against the S3 Vectors policy index and emits an outcome event), reply and approval (the drafter writes and the approver's decision is recorded). Every Lambda is event- or queue-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.

- **`intake-reader`** — S3 PUT trigger on `s3://rf-raw-mime/` for the email lane, and the target of the intake Function URL for the form and paste lanes. Parses the MIME (or the JSON body), and for free-text email calls Bedrock Haiku 4.5 to extract `{customer, email, item, order_ref, amount, asked_for}` as strict JSON with blanks for anything not present. Writes a clean record to the `rf-intake` SQS queue. Memory: 256 MB. Timeout: 30 s.
- **`intake-url`** — Lambda Function URL, `AuthType: NONE`, verifies a shared secret header and applies a small token-bucket rate limit before handing the body to the same reader path. Backs both the contact form (Lane 2) and the manual paste form (Lane 3, with a `source=paste` flag). Memory: 256 MB. Timeout: 15 s.
- **`checker`** — SQS event source on `rf-intake` (batch size 1 for clean per-request retry semantics). Embeds the request via Titan Text Embeddings V2 (`amazon.titan-embed-text-v2:0`, 1024-dim), queries the `rf-policy` S3

Vectors index for the top-k passages, and calls Bedrock Haiku 4.5 (`global.anthropic.claude-haiku-4-5-20251001-v1:0`) with a decide-only-from-these-passages prompt. If the model returns low confidence or conflicting passages, re-runs the single decision on Claude Sonnet 4.6 (`global.anthropic.claude-sonnet-4-6-20250930-v1:0`). Writes state to `rf-requests` and emits one of `rf.in_policy`, `rf.out_of_policy`, `rf.high_value`, `rf.not_covered` with the cited passage id. Memory: 512 MB. Timeout: 60 s.

- **`drafter`** — EventBridge rule on the four outcome events. Fetches the voice template for the outcome from `s3://rf-policy-source/voice.txt`, calls Haiku 4.5 to draft the reply, then runs a deterministic `safety_check()` (amount ≤ requested, outcome/answer agreement, no invented order/date/amount). Posts an approval card to Slack via `chat.postMessage` (Block Kit, with Approve/Edit/Decline) for the right channel/DM, or sends an email card via SES outbound. `not_covered` posts a card with no draft. Memory: 512 MB. Timeout: 30 s. *No money movement.*
- **`approve-handler`** — Lambda Function URL, `AuthType: NONE`; verifies the Slack signing secret. Handles Approve (send reply via SES `SendRawEmail`, mark `rf-requests` resolved, write `rf-audit`, post the decision row to the configured finance sink), Edit (open a modal pre-filled with the draft; on submit, send the edited reply and log the diff), and Decline (require a reason, send the decline reply, close as declined). Never calls any payment API — the actual payout stays a human/finance step. Memory: 256 MB. Timeout: 15 s.
- **`policy-sync`** — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Docs/Drive API (service-account credentials in Secrets Manager under `rf/drive/sa`) to export the policy and voice docs to `s3://rf-policy-`

`source/` only if changed, splits the policy into passages, embeds each with Titan V2, and upserts them into the `rf-policy` S3 Vectors index (deleting passage ids that no longer exist). Memory: 512 MB. Timeout: 60 s.

- **`digest`** — EventBridge Scheduler target, weekly Monday 9am. Reads `rf-requests` and `rf-audit` for the past week; posts a Slack summary: requests in, approved, edited, declined, and any in-policy drafts a human overrode. No Bedrock; a plain summary table. Memory: 256 MB.

Storage

- **DynamoDB · `rf-requests`** — one row per request, its live state. PK `request_id`; attributes: `source` (email/form/paste), `customer`, `item`, `order_ref`, `amount`, `outcome`, `cited_passage_id`, `status` (queued/awaiting-approval/resolved/declined). On-demand.
- **DynamoDB · `rf-audit`** — one row per write action of any kind. PK `(request_id, ts)`; attributes: `action` (approved/edited/declined), `by_user`, `cited_passage_id`, `amount`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **S3 Vectors · `rf-policy`** — the embedded policy passages. One vector per passage with metadata `{passage_id, heading, text}`. Re-indexed by `policy-sync` on every change.
- **S3 · `rf-policy-source`** — mirrored policy and voice docs as plain text. Versioning enabled, so a bad policy edit can be rolled back in one click.
- **S3 · `rf-raw-mime`** — raw inbound MIME from the help inbox. Lifecycle to Glacier at 30 days; expiry at 7 years.

- **SQS · `rf-intake`** — the single intake queue. `rf-intake-dlq` behind it with `maxReceiveCount: 5` so a poison record lands in the DLQ for inspection instead of looping.

Bedrock

- **Foundation models.** `global.anthropic.claude-haiku-4-5-20251001-v1:0` for the request read, the policy decision on the common path, and the reply draft; `global.anthropic.claude-sonnet-4-6-20250930-v1:0` for the hard-case decision only. Both via the Global cross-Region inference profile.
- **Embeddings.** `amazon.titan-embed-text-v2:0` at 1024 dimensions, for both the policy passages (at index time) and each incoming request (at query time). Used into Amazon S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The Sonnet path fires on a small fraction of requests, so its share of the token spend stays low.

EventBridge config

- **Outcome rule** — one EventBridge rule on the default bus matching `detail-type` in (`rf.in_policy`, `rf.out_of_policy`, `rf.high_value`, `rf.not_covered`). Target: `drafter` Lambda.
- **`rf-policy-sync`** — EventBridge Scheduler, `rate(15 minutes)`. Target: `policy-sync` Lambda.
- **`rf-weekly-digest`** — EventBridge Scheduler, `cron(0 9 ? * MON *)` in TZ. Target: `digest` Lambda.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `refunds.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `rf-inbound-rules`: one rule with recipient `refunds@your-company.com` → spam scan → S3 PUT to `s3://rf-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-reader`.
- SES outbound for the approved replies and email-card fallback: verify a sender identity at `support@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **checker role:** `sqs:ReceiveMessage` + `DeleteMessage` on `rf-intake`; `bedrock:InvokeModel` on the Titan, Haiku, and Sonnet ARNs; `s3vectors:QueryVectors` on the `rf-policy` index; `dynamodb:PutItem` + `GetItem` on `rf-requests`; `events:PutEvents` on the default bus.
- **drafter role:** `s3:GetObject` on the voice key; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token; `ses:SendRawEmail` for the email-card fallback; outbound network access to `slack.com`. *No* `dynamodb:*` writes beyond status.
- **approve-handler role:** `ses:SendRawEmail` from the verified sender; `dynamodb:PutItem` on `rf-requests` and `rf-audit`; `secretsmanager:GetSecretValue` on the Slack signing secret. *No* payment-service permissions of any kind — the role literally cannot move money.

- **intake-reader / intake-url roles:** `s3:GetObject` on `rf-raw-mime`; `bedrock:InvokeModel` on the Haiku ARN; `sqs:SendMessage` on `rf-intake`; `secretsmanager:GetSecretValue` on the form shared-secret.
- **policy-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on `rf-policy-source`; `bedrock:InvokeModel` on the Titan ARN; `s3vectors:PutVectors` + `DeleteVectors` on the `rf-policy` index; outbound network to `www.googleapis.com`.

Slack interactive flow

Approval cards are posted via the `chat.postMessage` Web API with Block Kit blocks containing the Approve, Edit, and Decline buttons. Button clicks are sent by Slack to the configured Interactivity request URL, which is the `approve-handler` Function URL. `approve-handler` verifies the Slack signing secret on the inbound request, parses the `action_id` (`approve`, `edit`, `decline`), opens a modal where needed (Edit and Decline open modals; Approve is one-tap), and processes the response on submit. High-value and out-of-policy cards are routed to the named approver's DM rather than the shared channel.

The Slack app needs `chat:write` and `im:write`, plus the Interactivity URL configured. The bot token lives in Secrets Manager under `rf/slack/bot-token`; the signing secret is `rf/slack/signing-secret`.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch

metric for alerting.

- **Alarms:** `rf-intake-dlq` depth > 0 (a request failed to process); checker failure rate > 1% in 24h; approve-handler signature-verification failures > 5/hour (might mean the Slack secret rotated); Bedrock token spend anomaly via a daily Cost Anomaly Detection monitor.
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `rf-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

The Google service-account credentials for the Docs/Drive API live in Secrets Manager under `rf/drive/sa`. The Slack bot token and signing secret are under `rf/slack/*`. The contact-form shared secret is under `rf/form/secret`. SES sender identity lives in IAM and the verified-domain config. The dollar cap, the named approver per category, the top-k for policy retrieval, and the finance-sink target all live in Parameter Store under `/rf/config/`. 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: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for `rf-policy-source` so a bad policy edit rolls back in one click, and keep the `approve-handler` role free of any payment permissions so the "never moves money" guarantee is enforced by IAM,

not just by code. Total deployable surface: around seven Lambdas, two DDB tables, one S3 Vectors index, three S3 buckets, one SQS queue plus its DLQ, one EventBridge rule on the default bus (plus the Scheduler rules), one SES rule set, and one Budgets alarm.

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