7-PART SERIES · FREE COMPANION

Receipt organizer

A serverless organizer that turns a photo of a receipt into a tidy expense record. Staff forward or snap a receipt; the system reads the vendor, date, total, and tax, sorts it into the right expense category, flags anything unclear for a quick human check, and files it where the bookkeeper needs it. No more shoebox of receipts at tax time. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Receipt organizer

PART 1 OF 7

MAY 9, 2026 PART 1 OF 7 · RECEIPT ORGANIZER SERIES ~5 MIN READ

A receipt organizer on AWS for a few dollars a month

Every small business has a shoebox. It might be a literal box of crumpled receipts, a phone camera roll full of blurry photos, or an email folder nobody opens. At tax time it all has to be read, sorted, and added up — usually by the owner, late at night, in a hurry. This post walks through the design of a small organizer that takes a photo of a receipt, reads the vendor, date, total, and tax off it, sorts it into the right expense category, flags anything it isn't sure about for a quick human check, and files the clean record where the bookkeeper needs it.

KEY TAKEAWAYS

- Three ways a receipt comes in: forward it by email, snap it on a phone, or upload it on the web.
- Every receipt ends in one of four results: filed, needs-review, duplicate, or rejected.
- The system reads vendor, date, total, and tax, then picks a category from your own chart of accounts.
- Anything unclear is flagged for a quick human check — a wrong number never files itself silently.
- 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.

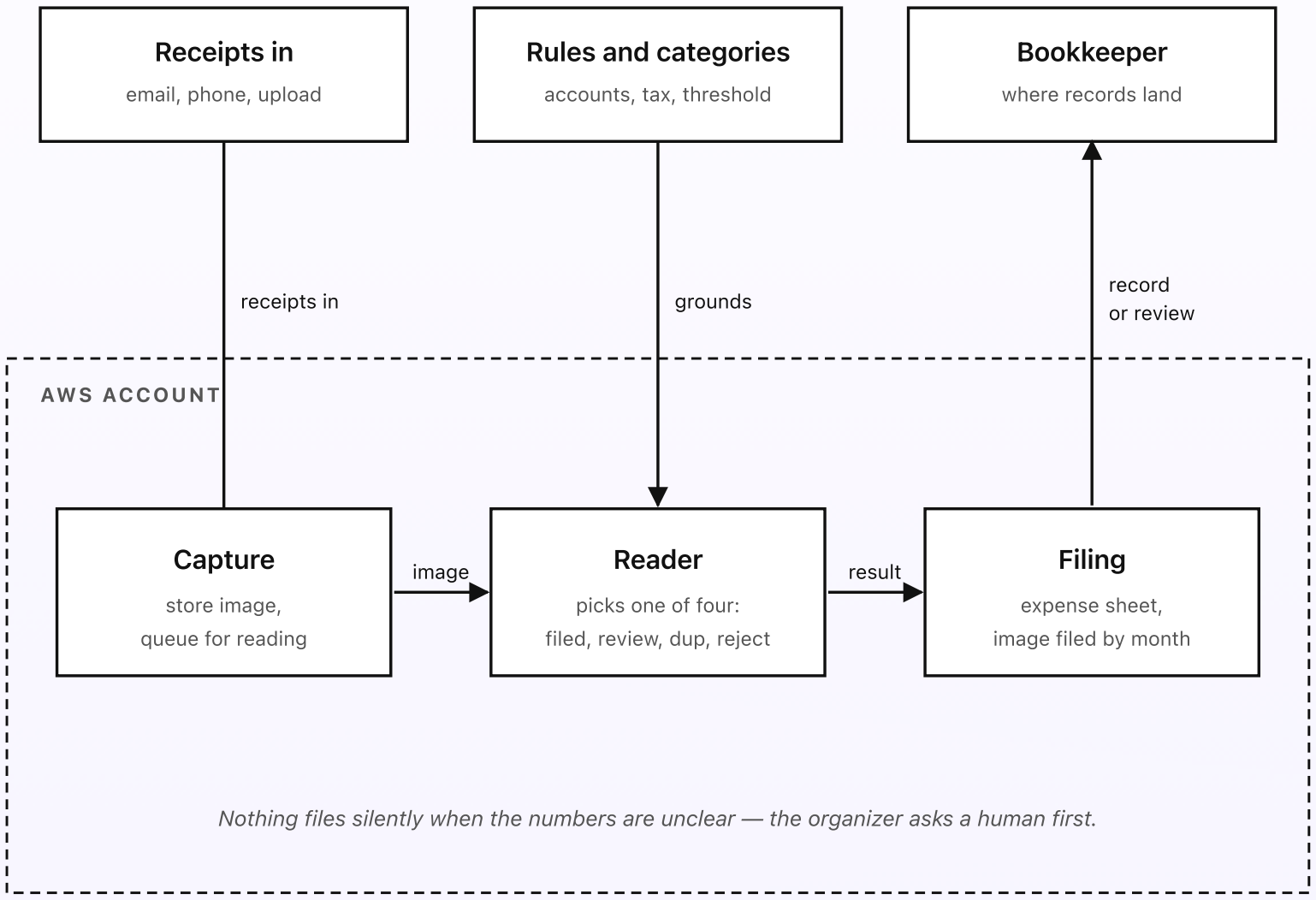

Receipts in
email, phone, upload
Rules and categories
accounts, tax, threshold
Bookkeeper
where records land
receipts in
grounds
record
or review
AWS ACCOUNT
Capture
store image,
queue for reading
image
Reader
picks one of four:
filed, review, dup, reject
result
Filing
expense sheet,
image filed by month
Nothing files silently when the numbers are unclear — the organizer asks a human first.

Fig 1. Three sources outside, three pieces inside AWS. Receipts flow in by email, phone, or upload. The Reader reads each one and picks one of four results. Filing writes the clean record to the expense sheet and sends only the unclear ones to the bookkeeper.

What you set up once (the outside)

- **Receipts in.** Three ways to send a receipt, covered in Part 2. Forward it by email to a dedicated address like `receipts@your-company.com`. Snap a photo on a phone (a saved home-screen shortcut opens the camera and uploads straight away). Or drag an image or PDF onto a simple web page. Whichever way it arrives, the system stores the original and starts reading it.
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc holds your chart of accounts (the list of expense categories your bookkeeper actually uses — "meals," "fuel," "software," "office supplies," and so on), some per-vendor hints ("anything from Shell goes to fuel"), the tax rules for your region, and the confidence threshold — how sure the system has to be before it files a receipt without asking. The *voice* doc holds the short wording of the review request the bookkeeper sees.
- **Bookkeeper.** The person who keeps the books. Clean, confident records land straight in the expense sheet. Only the unclear ones — a blurry total, an unknown vendor, a category the system isn't sure about — land in a small review queue with the receipt image, the fields the system read, the category it proposed, and an Approve button. Most days the queue is short.

What runs on every receipt (the inside)

- **The capture.** Three sources feed one place. An email-forward lane (forward a receipt to `receipts@your-company.com` and the system pulls the image or PDF

out of the message), a mobile-snap lane (a phone shortcut uploads the photo straight to a private upload link), and a web-upload lane (drag a file onto a page). Each one stores the original image, gives it an id, and drops it on a queue to be read. Nothing is processed in line — the queue lets a busy lunch-hour batch of receipts wait its turn without losing any.

- **The reader.** This is where the work happens. First, Amazon Textract reads the receipt image and pulls out the raw text and the obvious fields — vendor name, date, total, tax. Textract is good at receipts specifically; it knows what a total looks like. Then a Bedrock Haiku 4.5 call (a small, cheap AI model) takes that text, picks the right category from your chart of accounts, and double-checks the numbers add up. The reader then picks one of four results. *Filed*: every field is clear and the total checks out — file it. *Needs-review*: something is below the confidence threshold — a blurry total, an unknown vendor — so send it to the bookkeeper to confirm. *Duplicate*: the same vendor, date, and total already exist — flag it so the same lunch isn't claimed twice. *Rejected*: the image isn't a receipt at all (somebody forwarded a newsletter) — set it aside.
- **Filing.** For a filed receipt, write one clean row to the expense sheet: date, vendor, total, tax, category, who submitted it, and a link to the stored image. File the image itself in a folder by month so the bookkeeper can find the original in one click. For a needs-review receipt, post it to the review queue instead; once the bookkeeper approves (or fixes the category), it files the same way. A weekly digest lists what was filed and what's still waiting. A monthly summary writes a short plain-English note: spend by category, the biggest line items, anything still unreviewed.

In plain words

Your sales rep buys lunch for a client. $64.20 at a restaurant, $5.84 of that is tax. On the way out, she forwards the emailed receipt to `receipts@your-company.com`. Thirty seconds later the organizer has read it: vendor “Harbour Grill,” date today, total $64.20, tax $5.84. It picks the category “meals and entertainment” because the rules doc says restaurant receipts go there, and it’s confident, so it files the row in the expense sheet and tucks the image into this month’s folder. The rep does nothing else. The bookkeeper never touches it.

Now the harder one: the rep snaps a photo of a crumpled fuel receipt and the total is smudged. Textract reads “$8?.40” with low confidence on the middle digit. The organizer doesn’t guess. It marks the receipt needs-review and drops it in the bookkeeper’s queue with the image and the fields it could read. The bookkeeper glances at the photo, types “$83.40,” taps Approve, and it files. The cost of running all of this is about $3 a month. The cost of *not* running it is the shoebox — and the deductions nobody could prove at tax time.

DESIGN RULES THAT SHAPED EVERY DECISION

- A receipt is sent once, any way that's convenient. Email, phone, or upload — all three feed one pipeline.
- Four results, always. Filed, needs-review, duplicate, or rejected. There is no fifth.
- Unclear numbers never file themselves. Below the confidence threshold, a human confirms first.
- Duplicates are caught, not claimed twice. The same vendor, date, and total raises a flag.
- The expense sheet lives in Drive. Adding a category or fixing a vendor hint doesn't need a deploy.
- Every record links back to its original image. Audit a deduction next year and the proof is one click away.

Why this shape

Most small businesses handle receipts in one of three ways: a shoebox dealt with once a year, a phone camera roll nobody sorts, or a spreadsheet somebody types by hand. The shoebox works until tax time, when a weekend disappears into reading faded thermal paper. The camera roll is worse — the photos are there, but turning two hundred of them into categorized expenses is the same manual job, just digital. And the hand-typed spreadsheet is accurate right up until the week somebody gets busy and stops typing.

The setup above keeps the part humans are good at — deciding the tricky calls — and hands off the part they hate: reading the same fields off hundreds of receipts. The system reads every receipt the moment it arrives, files the clear ones on its own, and only asks for help on the genuinely unclear ones. The bookkeeper's job shrinks to a short daily queue instead of a yearly mountain. And because every record links back to its image, the proof is there if anyone ever asks.

The next four posts walk through each piece in turn: how a receipt gets captured, how a receipt gets read, how a receipt gets categorized, and how a receipt reaches the books. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 9, 2026 PART 2 OF 7 · RECEIPT ORGANIZER SERIES ~4 MIN READ

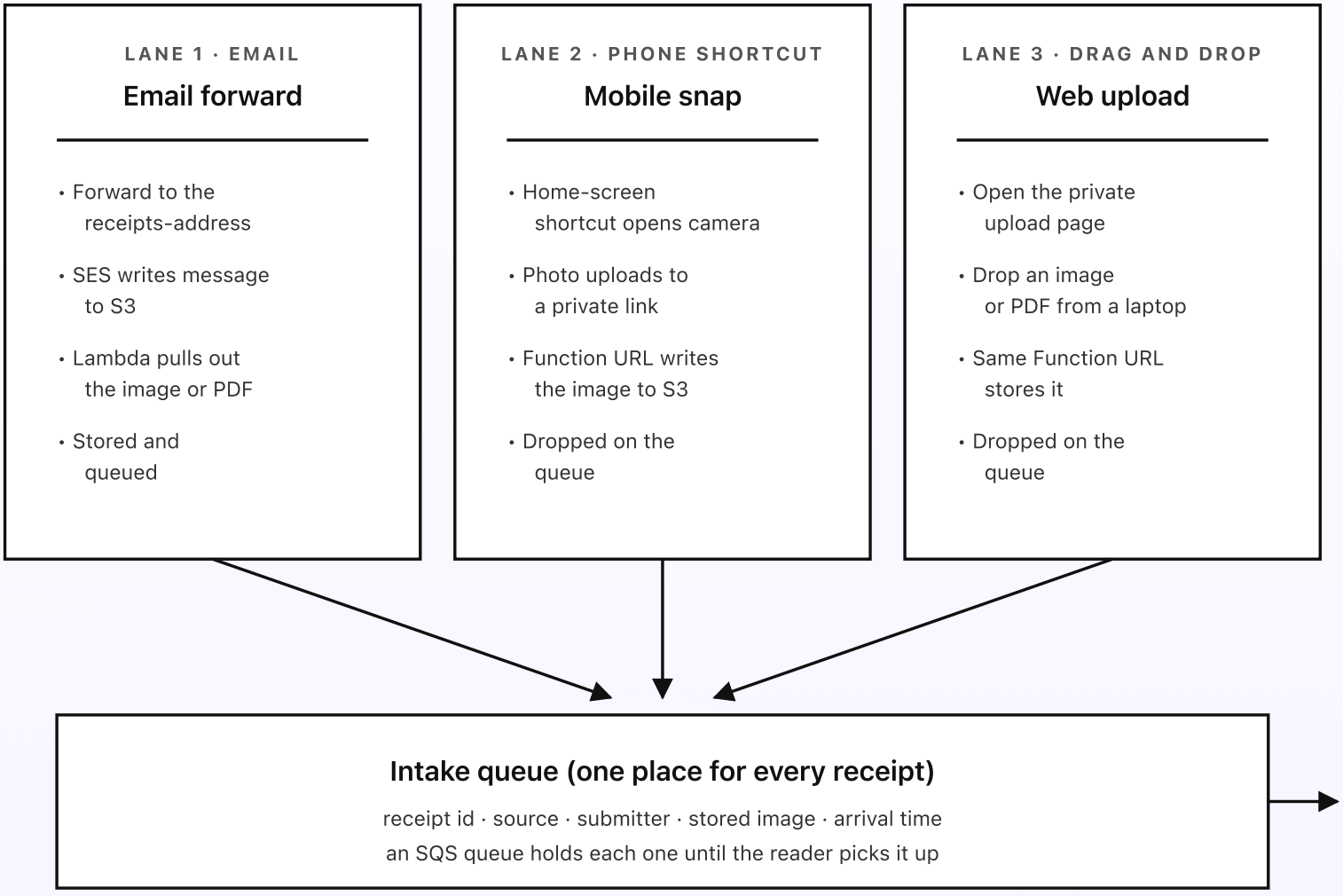
How a receipt gets captured

The organizer can only organize what it receives. So the first job is making it dead simple to send a receipt the moment you have it — not at the end of the week, not at tax time, but right there at the counter or the petrol pump. There are three ways a receipt gets in: forward it by email, snap a photo on a phone, or upload a file on a web page. All three exist because in real life people pay in different ways, and the one that needs the most steps is the one that never happens.

KEY TAKEAWAYS

- Three capture lanes feed one queue: email forward, mobile snap, and web upload.
- Email forwards arrive through Amazon SES; the image or PDF is pulled out of the message.
- The phone lane is a saved home-screen shortcut that opens the camera and uploads in one tap.
- The web lane is a single drag-and-drop page behind a private link.
- Every lane stores the original, gives it an id, and drops it on the queue — no receipt is lost.

Three lanes into one queue



Every lane stores the original first — nothing is read in line, so a lunch-hour rush just waits its turn.

Fig 2. Three lanes converge on one intake queue. Email, phone, and web all store the original image first, then drop it on the queue. The reader picks receipts off the queue one at a time, so a sudden batch never overwhelms anything or gets lost.

Lane 1: email forward

The lane most receipts arrive on, because so many receipts are already emails. Set up a dedicated inbound address — something like `receipts@your-company.com` — through Amazon SES. Anyone on the team forwards a receipt to that address and the organizer takes it from there. SES writes the raw message to `s3://ro-raw-mime/`. The S3 write triggers a capture Lambda. The Lambda walks the message to find the attachment — an image (JPEG, PNG, HEIC) or a PDF — or, if the receipt is in the email body itself, screenshots the rendered HTML. It stores the original in `s3://ro-receipts/` under a fresh receipt id, records who forwarded it (from the original sender line), and drops a message on the intake queue.

This lane covers online purchases, supplier invoices that come by email, and anything a staff member can forward from their phone's mail app in two taps. It's the path that needs no app and no new habit beyond "forward it."

Lane 2: mobile snap (the lane for paper)

Paper receipts — the petrol station, the hardware store, the taxi — need a camera. Rather than ask staff to install and log into an app, the mobile lane is a saved home-screen shortcut. You set it up once on each phone: open a private link, tap "Add to Home Screen," and now there's an icon that opens straight to the camera. Snap the receipt, and the photo uploads in one tap to a Lambda Function URL — a

private web address that accepts the file directly, with no API Gateway in front of it.

The Function URL is protected by a long, unguessable token baked into each person's shortcut, so only your team can post to it. The Lambda writes the photo to `s3://ro-receipts/`, tags it with the submitter, and drops it on the queue — the same queue the email lane feeds. From the reader's point of view, a photo from a phone and a forwarded PDF look identical.

Lane 3: web upload

Sometimes a batch of receipts is already on a laptop — downloaded PDFs, scans, a folder of images from a trip. Forcing those onto a phone to re-photograph them is a fight you don't need. Lane 3 is a single page behind a private link: drag a file (or several) onto it and they upload. It hits the same Function URL as the phone lane, stores each file in `s3://ro-receipts/`, and drops each one on the queue.

The web lane is the most occasional of the three — most days nobody uses it — but it's the right tool for the catch-up session where somebody clears a backlog in one go.

Why everything funnels through one queue

Three lanes in, but only one queue the reader watches. That's deliberate. If each lane read its own receipts in its own way, a bug in the phone path wouldn't show up the same as a bug in the email path, and a busy afternoon could have three lanes all trying to call Textract at once. Funneling everything through one queue means the reader processes receipts at a steady pace, retries cleanly if Textract is

briefly busy, and never loses one — a receipt sits safely on the queue until it's been read and filed. Anything that fails repeatedly lands in a side queue (a dead-letter queue) for a human to look at, instead of silently vanishing.

Next post: how the reader actually reads a receipt — pulling the vendor, date, total, and tax off the image, and how it decides when it's sure enough to file on its own.

PART 3 OF 7

MAY 9, 2026 PART 3 OF 7 · RECEIPT ORGANIZER SERIES ~5 MIN READ

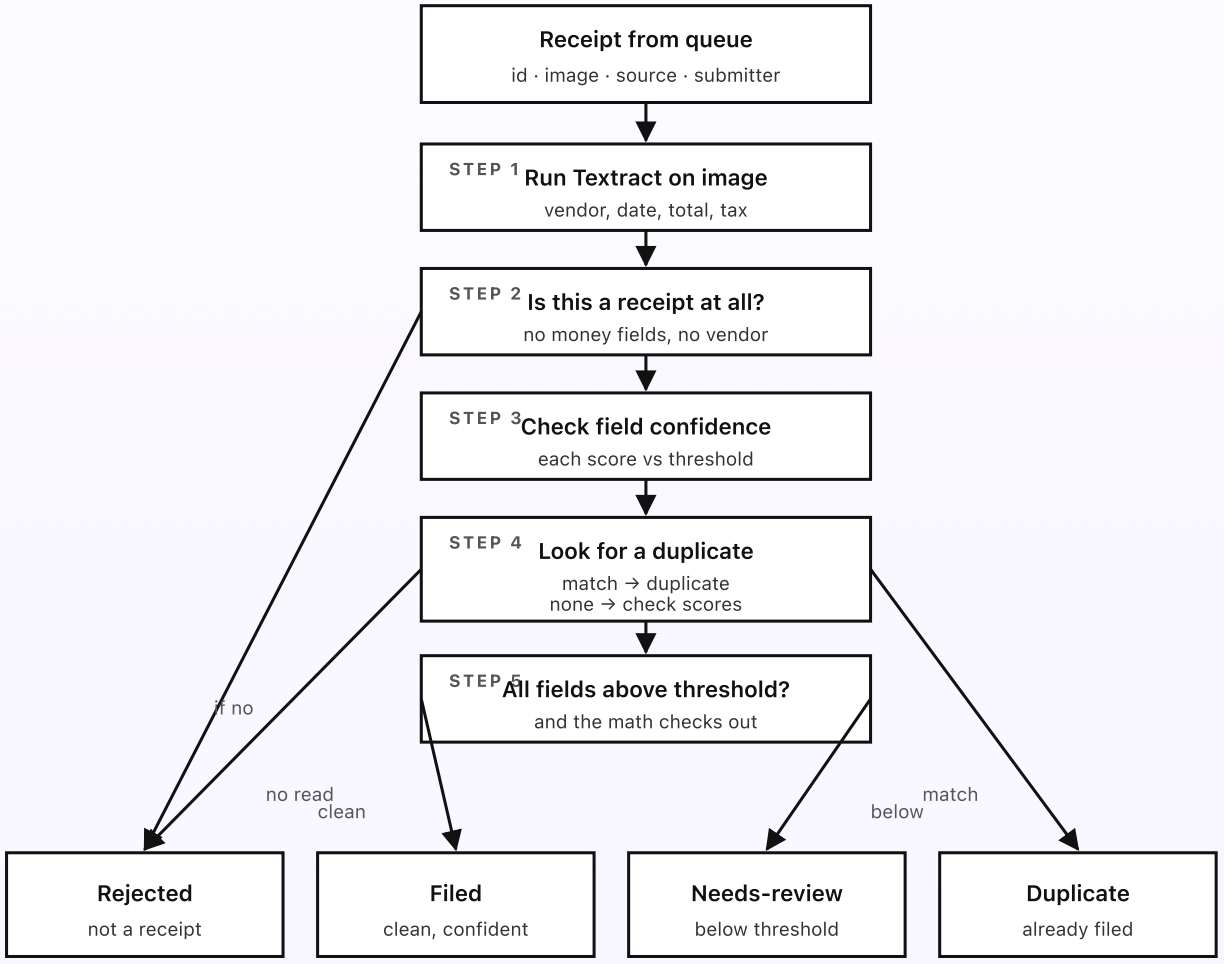
How a receipt gets read

A receipt comes off the queue. The reader Lambda runs Amazon Textract on the image, gets back the vendor, date, total, and tax — each with a score for how sure Textract is — and then has to decide what to do with it. Most receipts are clean and file themselves. Some are blurry, some are duplicates, some aren't even receipts. The whole decision is plain rules over Textract's confidence scores. No guessing on the numbers that matter.

KEY TAKEAWAYS

- Amazon Textract reads the receipt and returns vendor, date, total, and tax with a confidence score each.
- The confidence threshold lives in the rules doc — below it, a human confirms before anything files.
- Four results per receipt: filed, needs-review, duplicate, or rejected.
- A duplicate check on vendor, date, and total stops the same purchase being claimed twice.
- The reader never invents a number it couldn't read — an unclear total goes to a human, not a guess.

The decision flow, per receipt



The threshold lives in the rules doc — raise it for more checks, lower it for more auto-filing.

Fig 3. The reader's decision tree, per receipt. Five steps decide which of four results applies. Textract reads the fields; the rules doc holds the threshold; the reader only enforces it.

What Textract reads, and how sure it is

Textract has a feature built specifically for receipts and invoices — `AnalyzeExpense`. You hand it the image, and it hands back named fields: vendor name, transaction date, the total, the tax amount, and often the individual line items. The useful part is that each field comes with a confidence score from 0 to 100 — how sure Textract is that it read that value correctly. A crisp digital PDF scores in the high 90s on every field. A crumpled photo of faded thermal paper might read the vendor fine but score the total at 71 because one digit is smudged.

That score is the whole game. The reader doesn't treat "$83.40 at 71% sure" and "$83.40 at 99% sure" the same way. The first one is a question for a human; the second one files itself. The rules doc holds the threshold — the line between "file it" and "ask a human" — with a sensible default of 90 for the money fields (total and tax) and a slightly lower bar for vendor name, where a small misread matters less.

Four results, always

Every receipt, once read, lands in exactly one of four buckets. The names are plain on purpose.

- **Filed.** Every field read cleanly, the total and tax are above the money threshold, the math is sane (tax isn't larger than the total), and it's not a duplicate. The record is written straight to the expense sheet. Most receipts, most days, file themselves.
- **Needs-review.** At least one field came back below the threshold — a blurry total, a date that didn't parse, a vendor Textract couldn't read. The receipt goes to the bookkeeper's queue with the image and the fields it could read, so a human can confirm the one unclear value in a few seconds.
- **Duplicate.** A receipt with the same vendor, date, and total already exists. This catches the common case where someone forwards the email *and* snaps the paper copy, or uploads the same batch twice. It's flagged, not filed, so the same lunch never gets claimed twice.
- **Rejected.** Textract found no money fields and no vendor — it's not a receipt. Somebody forwarded a newsletter, a calendar invite, or a blank photo. It's set aside in a rejected folder with a one-line note, so nothing clutters the books and nothing is silently lost either.

The duplicate check, in plain terms

Before a clean receipt files, the reader does one more look: it searches the recent records for the same vendor, the same date, and the same total. If it finds one, the new receipt is marked a duplicate. The check is deliberately strict — all three have to match — because two genuine $12.00 coffees from the same café on the same day are possible, and the goal isn't to block real expenses. When in doubt, a near-match is sent to review rather than auto-rejected, so a human makes the final call on the borderline ones.

Why the reader follows rules instead of guessing

The reader could let the AI model decide everything — read the receipt, pick the numbers, file it. It doesn't. Two reasons. First, the money fields are the one place a wrong value does real damage: a total off by a digit flows into the books, the tax return, and an auditor's spreadsheet. So those fields are gated on Textract's own confidence score, and anything shaky goes to a human. Second, the read itself should be predictable — the same image always produces the same fields and the same result, so a bookkeeper can trust what they're seeing.

The AI model does play a part — but in the next step, not this one. Reading the fields is Textract's job. Deciding which category a clean receipt belongs to is where Bedrock comes in. That's the next post.

PART 4 OF 7

MAY 9, 2026 PART 4 OF 7 · RECEIPT ORGANIZER SERIES ~5 MIN READ

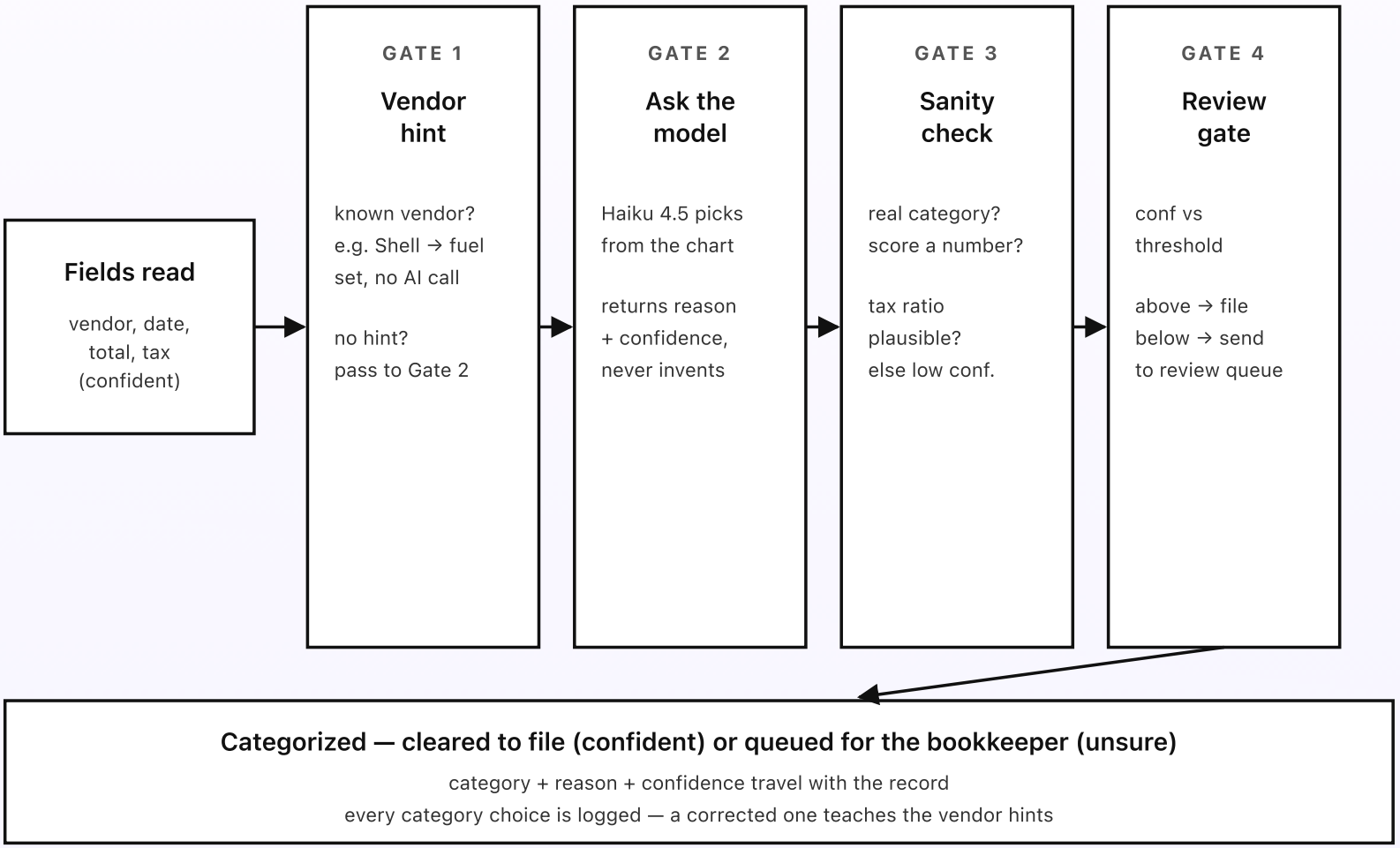
How a receipt gets categorized

The reader pulled clean fields off a receipt — vendor, date, total, tax. Now the organizer has to decide which expense category it belongs to: is this lunch a “meal,” the petrol “fuel,” the software subscription “software”? Get it wrong and the books are misleading and the tax return is off. So the category isn’t a wild guess from an AI model. Four small gates sit between the read fields and the filed record, and the AI only weighs in once it’s grounded in your own rules.

KEY TAKEAWAYS

- Categories come from your own chart of accounts in the rules doc — not a generic list.
- A vendor-hint lookup handles the easy cases first, with no AI call at all.
- Bedrock Haiku 4.5 picks the category for the rest, choosing only from your account list.
- A sanity check makes sure the model returned a real category and a confidence score.
- Low-confidence picks go to the bookkeeper's review queue, not straight to the books.

Four gates on every category decision



The chart of accounts is the only menu — the model picks from it, never beyond it.

Fig 4. Four gates between the read fields and the category. Try a vendor hint first. Ask the model only when needed. Sanity-check the answer. Then either file with confidence or send to the bookkeeper. The chosen category and its reason travel with the record.

Gate 1: the vendor hint (the free, instant case)

Most categorization is boring and repetitive. Anything from Shell or BP is fuel. Anything from your one software vendor is software. Anything from the office-supply shop down the road is office supplies. The rules doc holds a short list of these per-vendor hints — vendor name on the left, category on the right. Gate 1 checks the read vendor against that list first. If there's a match, the category is set right there, with no AI call at all. Over a few weeks of use, this list grows to cover the vendors you see most, and the share of receipts that even reach the model shrinks.

This is the cheapest possible path: a lookup in a short list. The hints are plain text a bookkeeper can edit in Drive, so adding "the new café near the office goes to meals" is a thirty-second edit, not a code change.

Gate 2: ask the model (only when there's no hint)

For a vendor with no hint — a one-off supplier, a shop nobody's seen before — the organizer asks Bedrock Haiku 4.5. The prompt is short and tightly framed: here are the read fields, here is the full chart of accounts, pick exactly one category from this list, give a one-line reason, and rate your confidence. The crucial constraint is that the model chooses *from your list*. It can't invent "travel-meals-

misc" that your bookkeeper has never heard of; it has to return one of the categories that already exists in your books.

Haiku is the right model here: this is a small, well-defined classification job, not heavy reasoning, so the cheap, fast model is plenty. It runs only on the receipts that got past Gate 1, which over time is the minority. The reason it returns matters too — "restaurant name and a tip line suggest a meal" — because if a human ends up reviewing it, that one line saves them re-reading the whole receipt.

Gate 3: the sanity check

An AI answer needs checking before it's trusted. Gate 3 confirms three things. The category the model returned actually exists in the chart of accounts (if it somehow returned something off-list, that's treated as no answer). The confidence is a real number, not missing. And the numbers are plausible for the category — a receipt where the tax is larger than the total, or a "fuel" receipt for $4,000, is suspicious enough to flag regardless of how confident the model was. Anything that fails a sanity check is downgraded to low confidence, which means it heads to a human at the next gate.

Gate 4: the review gate (the human check)

The last gate is the one that decides whether a person ever sees the receipt. It compares the category confidence against the threshold in the rules doc. Above the line — a clear vendor hint, or a confident model pick that passed the sanity check — and the receipt is cleared to file with its category attached. Below the line — an unsure model pick, a sanity-check downgrade, a category the business has

flagged as “always review” (some businesses want every “entertainment” receipt eyeballed) — and it’s sent to the bookkeeper’s review queue with the proposed category, the model’s reason, and an Approve button.

When the bookkeeper corrects a category — “no, that vendor is office supplies, not software” — the system offers to remember it as a new vendor hint. Accept, and the next receipt from that vendor skips the model entirely and files on its own. The system gets quieter the more it’s used.

Why the gates exist

None of these gates is clever on its own. They’re the steps a careful bookkeeper already takes — recognize the regulars on sight, look up the unfamiliar ones, double-check anything that smells wrong, and ask when genuinely unsure. Putting them in code as four small gates makes the AI a junior assistant that proposes, not a black box that decides. The chart of accounts is the only menu it gets to choose from, every choice comes with a reason, and the unsure ones always reach a human before they touch the books.

Next post: how a categorized receipt actually reaches the books — the clean record written to the expense sheet, the image filed by month, and what happens when the bookkeeper approves or corrects a reviewed one.

PART 5 OF 7

MAY 9, 2026 PART 5 OF 7 · RECEIPT ORGANIZER SERIES ~5 MIN READ

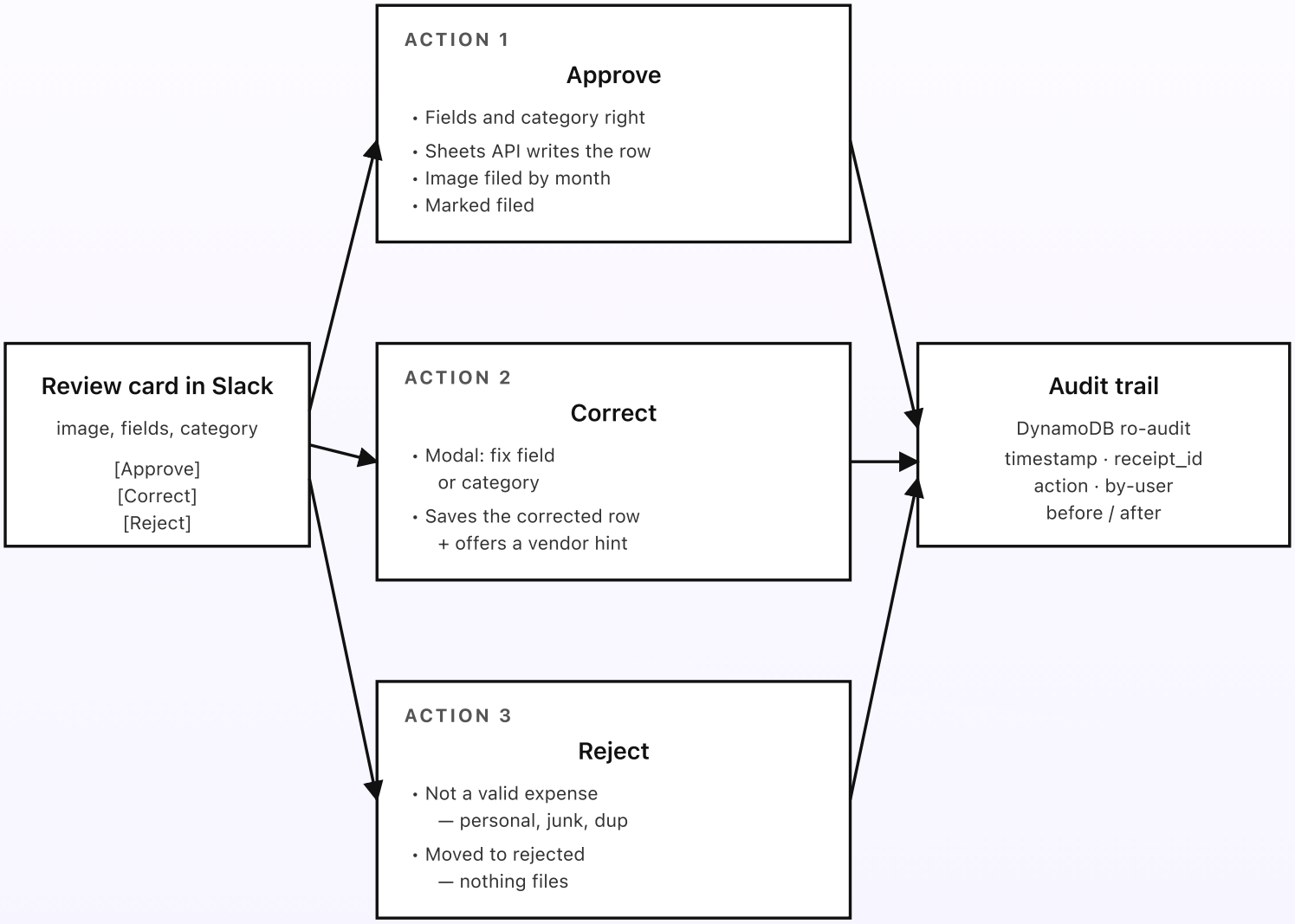
How a receipt reaches the books

A confident receipt files itself — one clean row in the expense sheet, the image tucked into this month's folder, done. The interesting case is the unsure one sitting in the bookkeeper's review queue. There's a card with the receipt image, the fields the system read, the proposed category, and three buttons. This post walks through the three things the bookkeeper can do — approve, correct, reject — and how the expense sheet, the stored image, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Confident receipts file straight to the expense sheet with no human touch.
- Three actions on a review item: *approve* (file as-is), *correct* (fix, then file), *reject* (set aside).
- Each action updates the expense sheet via the Sheets API and writes an audit row.
- A correction can teach a new vendor hint, so the next similar receipt files on its own.
- Every filed record links back to its original image — the proof is one click away.

Three actions on a review item



Every filed record links back to its image — the proof of any expense is one click away.

Fig 5. Three actions per review item, three different effects. Approve files as-is. Correct fixes a field or category, then files, and can teach a vendor hint. Reject sets it aside. Every action writes to the audit trail.

Action 1: approve (the most common)

The bookkeeper opens the queue first thing. The top card is a fuel receipt the system read at 88% on the total — just under the threshold — but the image is perfectly clear and "$83.40" is right there. She taps *Approve*.

The button submits to a Function URL Lambda. Three things happen, in order. First, the Sheets API writes one row to the expense sheet: date, vendor, total, tax, category, who submitted it, and a link to the stored image. Second, the receipt's image is moved into the folder for this month — `2026-05/` — so the original is filed where the bookkeeper expects it. Third, an `action: approved` row is written to `ro-audit` with the user, the timestamp, and the fields as filed. The card updates in place to "Filed by Maria," and it's gone from the queue.

Approving is the path for the receipts the system read correctly but wasn't quite sure enough to file on its own. It's a glance and a tap — seconds, not minutes.

Action 2: correct (fix, then file)

Some review items genuinely have something wrong. The total was misread — "$8?.40" should be "$83.40." The date didn't parse. The model guessed "software" for a vendor that's really office supplies. *Correct* opens a small modal

pre-filled with everything the system read, so the bookkeeper isn't retyping — she just fixes the one field that's wrong and hits Save.

On save, the Function URL Lambda writes the corrected row to the expense sheet, files the image, and writes an `action: corrected` audit row that records both the original read value and the corrected one. If the correction was a category change for a known vendor, the modal offers a checkbox: "Always file [vendor] as [category]?" Tick it and a new vendor hint is added to the rules doc, so the next receipt from that vendor sails through Gate 1 and never reaches a human again. This is how the queue shrinks over time — every correction is a chance to teach the system one more thing it won't have to ask about again.

Action 3: reject (set it aside)

Sometimes the right answer is "this shouldn't be in the books at all." A staff member forwarded a personal purchase by mistake. A duplicate slipped past the automatic check because the photo was cropped differently. The image is junk — a thumb over the lens. *Reject* moves the receipt to a rejected folder with a one-line reason and writes an `action: rejected` audit row. Nothing reaches the expense sheet.

Rejecting isn't deleting. The image and the record stay in the rejected folder, so if a staff member asks "what happened to that receipt I sent?" there's an answer — "rejected as a personal purchase by Maria on the 12th" — rather than a mystery. The monthly summary in Part 6 reports the count of rejects so nothing disappears quietly.

Every action is logged, every action is reversible

The `ro-audit` table records every approve, correct, and reject with the user who did it, the timestamp, and a snapshot of the record before and after. If a wrong total gets entered during a correction (fat-fingered a digit), a small “undo last action” admin command reads the previous-state snapshot and restores the row — and the undo is itself an audit row, so the trail stays clean. Because every filed record links back to its original image, an auditor or an accountant who questions a line next year can open the proof in one click.

This reversibility and this paper trail are the whole point at tax time. The books aren’t just a list of numbers somebody typed; every number traces back to a receipt, a person, and a moment — and the unclear ones were all confirmed by a human before they counted.

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 Textract is the only line that grows with the number of receipts.

PART 6 OF 7

MAY 9, 2026 PART 6 OF 7 · RECEIPT ORGANIZER SERIES ~3 MIN READ

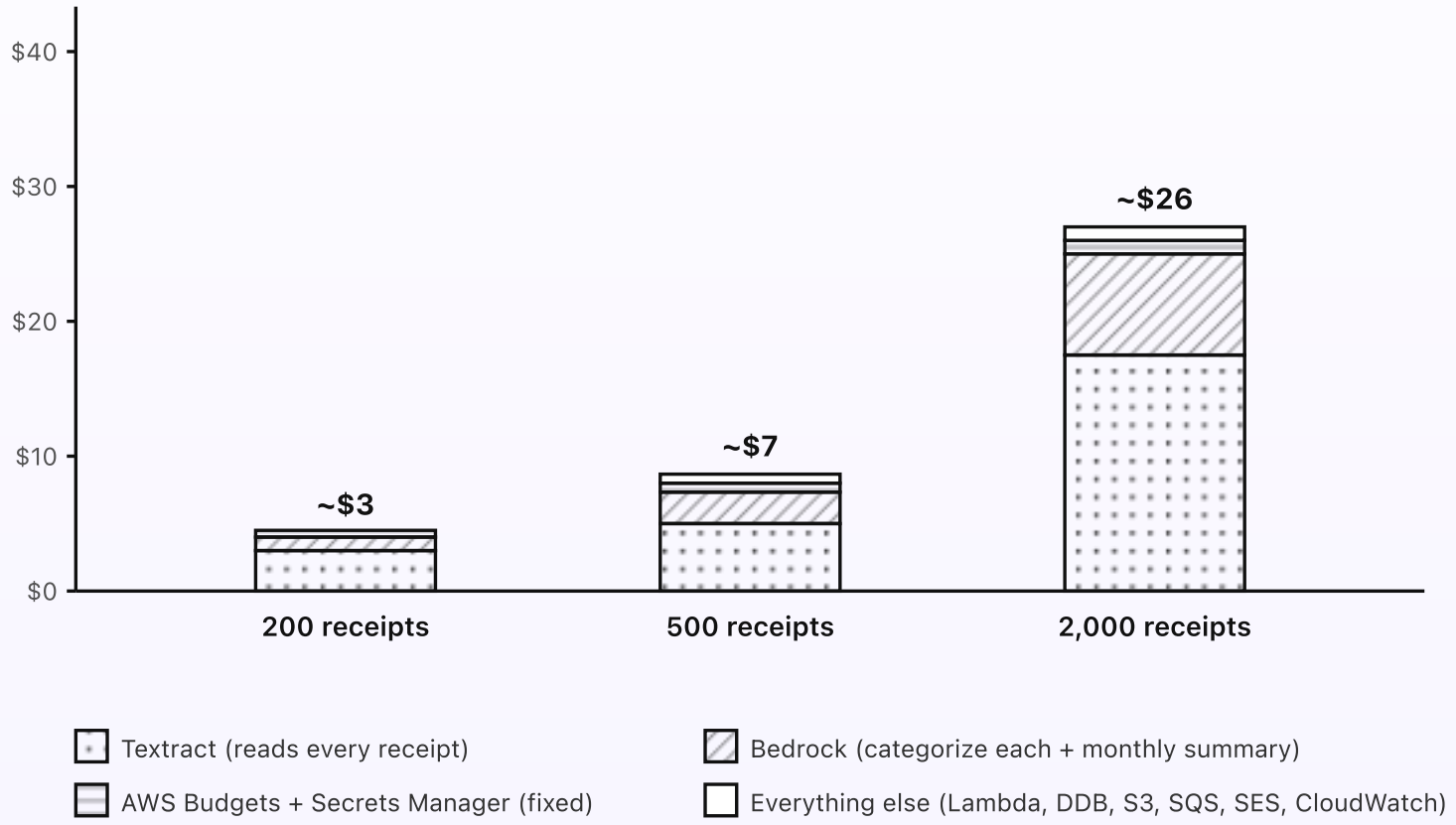
What the receipt organizer costs

The organizer is one of the cheaper systems in this whole series, but it has a twist the others don't: it does real work on every single receipt — reading it with Textract and categorizing it with a small AI model. So unlike a system that sweeps a list once a day, this one's cost grows with how many receipts you send. The good news is that per receipt the cost is fractions of a cent, so at typical small-business volume the bill is still a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $3/month at typical SMB volume (roughly 200 receipts a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- Textract is the line that grows with receipts — a cent or two each.
- Bedrock Haiku fires once per receipt to categorize, and once a month for the summary.
- At 500 receipts the bill is around $7. At 2,000 receipts it's around $26.

Cost at three volumes



The cost scales with receipts, not a daily sweep — and per receipt it is a couple of cents.

Fig 6. Monthly cost at three receipt volumes. Textract and Bedrock are the slices that grow, because both fire once per receipt. The everything-else bucket stays small — the Lambda and database work per receipt is tiny.

Where the dollars actually go

Textract (the line that grows). Every receipt is read once with Textract's receipt-and-invoice feature. The price is per page, and a receipt is almost always one page. At a cent or two per receipt, 200 receipts a month is a couple of dollars; 2,000 receipts is the largest single line on the bill. This is the cost of the system doing the actual reading you'd otherwise do by hand — and it's the cheapest part of the whole job.

Bedrock Haiku 4.5. Two callsites. The big one is categorizing — one small call per receipt that doesn't match a vendor hint. The input is the read fields and the chart of accounts (a few hundred tokens), and the output is one category, a reason, and a score (a few dozen tokens), so it's a fraction of a cent each. Because the vendor-hint lane handles the regulars with no AI call at all, the share of receipts that reach the model shrinks over time. The second callsite is the monthly summary: one larger call that writes a short spend-by-category note. A couple of cents a month.

Lambda runtime. The capture, reader, categorizer, filing, and action-handler Lambdas each do a small amount of work per receipt — pull an image, call Textract, call Bedrock, write a row. Each runs for a second or two. Across all of them, the Lambda total stays under a dollar even at 2,000 receipts.

DynamoDB on-demand. Three small tables: `ro-receipts`, `ro-review`, `ro-audit`. A handful of reads and writes per receipt. Pennies a month at any of these volumes.

S3 + storage. The stored receipt images, filed by month, plus the raw inbound email. A receipt image is small; even 2,000 a month is a modest amount of

storage. A few cents.

SQS and SES. The intake queue is essentially free at this scale. SES inbound for the email-forward lane is $0.10 per thousand messages received — cents a year for an SMB.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the upload and review-action endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Everything runs only when a receipt arrives.
- **A Knowledge Base.** Categorizing picks from a short, structured chart of accounts — a small prompt beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **A heavy model on every receipt.** Categorizing is a small, well-framed job, so it uses the cheap, fast Haiku model — not a larger one.

How the cost scales

Textract and Bedrock both grow roughly linearly with the number of receipts, because both fire once per receipt. So the bill at 5,000 receipts a month is around $65; at 10,000 it's around $130. The everything-else bucket barely moves — the database and Lambda work per receipt is tiny. The one lever that lowers cost is the vendor-hint list: every receipt that matches a hint skips the Bedrock call

entirely, so a business that trains its hints well over a few months sees the Bedrock slice shrink even as receipts climb.

Set an AWS Budgets alarm at $30/month for a typical SMB so anything unusual pages you before the bill matters — raise the ceiling if your receipt volume is genuinely high. The normal-volume bill stays well under that line.

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

PART 7 OF 7

MAY 9, 2026 PART 7 OF 7 · RECEIPT ORGANIZER SERIES ~8 MIN READ

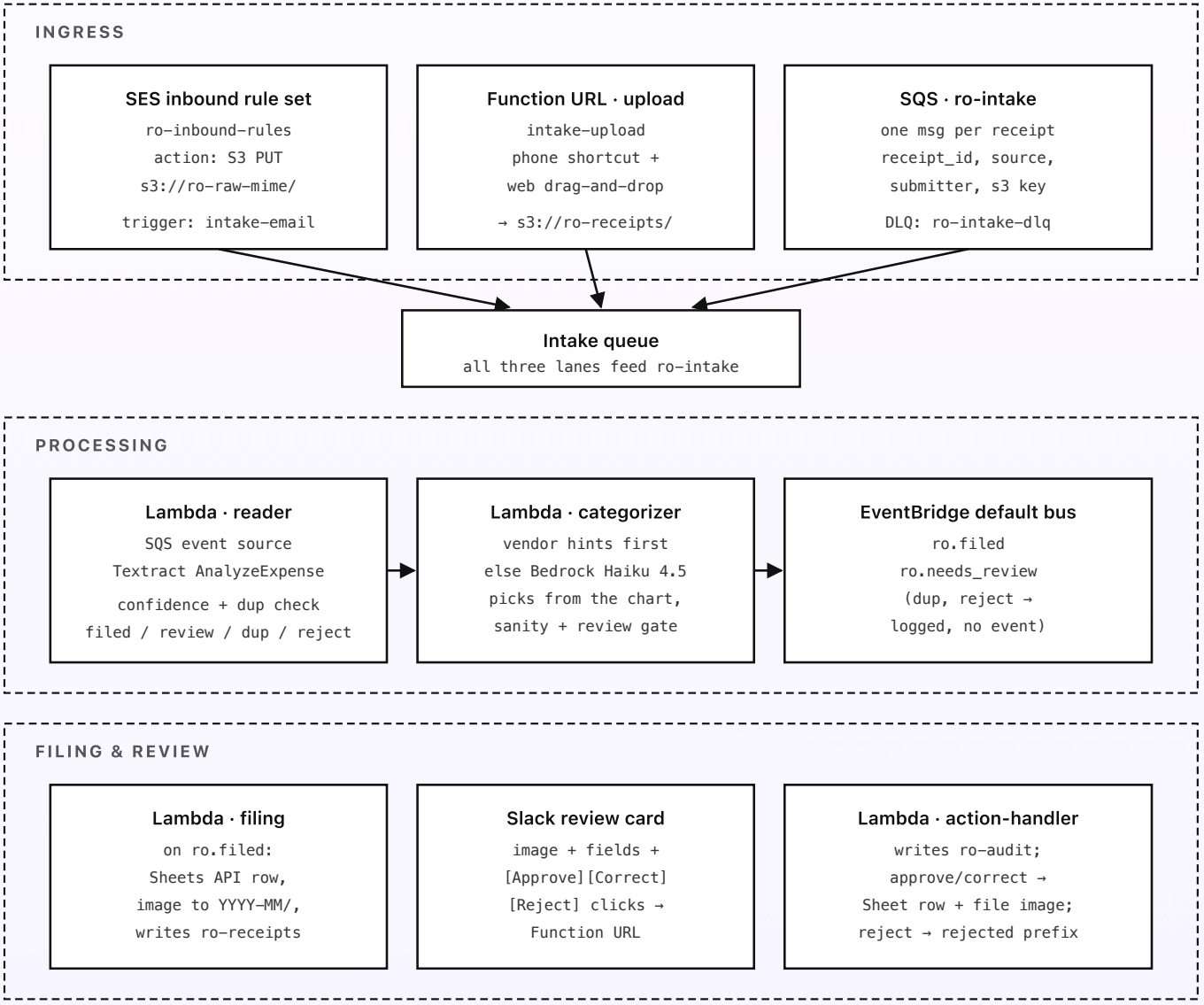
Engineering reference: the receipt organizer architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, the SQS intake config, the Function URL surfaces, the DynamoDB schemas, and the Slack review 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, Textract, Bedrock Global cross-Region inference, and the queue and Function URL surfaces are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is a receipt that waits an extra hour on the queue, not a regional outage. One AWS account dedicated to the organizer (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every filed record links to its image — and every interaction is logged to ro-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes onto one queue), processing (the reader and categorizer emitting events), filing and review (the record files or the review card ships and the response 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-email`** — S3 PUT trigger on `s3://ro-raw-mime/`. Parses the MIME tree, extracts the first image (JPEG/PNG/HEIC) or PDF attachment, or renders the HTML body to an image if the receipt is inline. Writes the original to `s3://ro-receipts/<receipt-id>`, records the forwarding sender as the submitter, and sends a message to the `ro-intake` SQS queue. Memory: 512 MB (HEIC decode and HTML render). Timeout: 60 s.
- **`intake-upload`** — Lambda Function URL, `AuthType: NONE`; verifies a per-device bearer token (issued from Parameter Store under `/ro/upload-tokens/`) before accepting the body. Serves the drag-and-drop upload page on GET and accepts a multipart file on POST. Writes the file to `s3://ro-receipts/`, tags the submitter, and enqueues to `ro-intake`. Used by both the phone shortcut and the web page. Memory: 256 MB. Timeout: 30 s.
- **`reader`** — SQS event source on `ro-intake` (batch size 1 for clean per-receipt retries). Runs Textract `AnalyzeExpense` on the image; for multi-page PDFs

uses the async `StartExpenseAnalysis` + completion via SNS. Reads the confidence threshold from `s3://ro-rules-source/rules.txt`; checks for duplicates by querying `ro-receipts` on a `(vendor, date, total)` GSI. Decides filed/needs-review/duplicate/rejected. For filed and needs-review, invokes the categorizer in-process, then emits `ro.filed` or `ro.needs_review`. Memory: 512 MB. Timeout: 120 s. Maximum receives 3, then to `ro-intake-dlq`.

- **`categorizer`** — invoked by `reader` (or deployable as its own function). Applies vendor hints from `rules.txt` first; on no match, calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) with the read fields and the chart of accounts, constrained to return one category from that list plus a reason and a confidence. Runs the sanity check (category in chart, score numeric, tax-to-total plausible) and the review-gate comparison. Memory: 256 MB. Timeout: 30 s.
- **`filing`** — EventBridge rule on `ro.filed`. Writes one row to the expense sheet via the Google Sheets API (service-account credentials in Secrets Manager under `ro/google/sa`): date, vendor, total, tax, category, submitter, image link. Moves the image to `s3://ro-receipts/YYYY-MM/` and writes the final record to `ro-receipts`. Memory: 256 MB. Timeout: 30 s.
- **`review`** — EventBridge rule on `ro.needs_review`. Posts a Slack review card via `chat.postMessage` with the receipt image, the read fields, the proposed category, the model's reason, and Approve/Correct/Reject buttons. Writes a pending row to `ro-review`. Memory: 256 MB. Timeout: 30 s.

- **`action-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies the Slack signing secret on the request body. Triggered by Slack button clicks (Approve/Correct/Reject) and modal submissions. On approve or correct, writes the row to the sheet via the Sheets API and files the image; on a category correction, optionally appends a vendor hint to `rules.txt`; on reject, moves the image to `s3://ro-rejected/`. Always writes to `ro-audit`. Memory: 256 MB. Timeout: 15 s.
- **`digest`** — EventBridge Scheduler target, weekly Sunday 6pm. Reads `ro-receipts` and `ro-review` for the week; posts a digest to a configured Slack channel summarizing what was filed and what's still waiting in review. No Bedrock; a plain summary table. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `ro-receipts` and `ro-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph spend-by-category note; emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB · `ro-receipts`** — one row per receipt. PK `receipt_id`; attributes: `source`, `submitter`, `vendor`, `date`, `total`, `tax`, `category`, `result` (filed/needs_review/duplicate/rejected), `field_scores`, `image_key`. GSI on `(vendor, date, total)` for the duplicate check. On-demand.
- **DynamoDB · `ro-review`** — one row per review item. PK `receipt_id`; attributes: `proposed_category`, `reason`, `read_fields`, `status` (pending/approved/corrected/rejected), `slack_ts`. On-demand.

- **DynamoDB · `ro-audit`** — one row per write action of any kind. PK `(receipt_id, ts)`; attributes: `action` (filed/approved/corrected/rejected), `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **S3 · `ro-receipts`** — the original receipt images, filed under `YYYY-MM/` once a record is filed. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years (the usual record-retention window).
- **S3 · `ro-rules-source`** — mirrored chart of accounts, vendor hints, tax rules, threshold, and the voice doc as plain text. Versioning enabled.
- **S3 · `ro-raw-mime`** — raw inbound MIME from forwarded receipts. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3 · `ro-rejected`** — images rejected at review or read time, kept with a one-line reason for audit.

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: `categorizer` for the per-receipt category pick, and `summary` for the monthly spend narrative. Sonnet 4.6 is not used — categorizing is a small, well-framed classification job that Haiku handles cleanly.
- **Embeddings.** Not used. The chart of accounts is a short structured list; a constrained prompt beats vector retrieval here. No Knowledge Base, no S3 Vectors.

- **Quotas.** Default account quotas are more than enough at SMB volume; the categorizer fires at most once per receipt, and the vendor-hint lane removes the regulars.

Textract

- **Receipts and single-page images.** Synchronous `AnalyzeExpense` in `reader` — returns `SummaryFields` (vendor, date, total, tax) and `LineItemGroups`, each with a confidence score.
- **Multi-page PDFs.** Async `StartExpenseAnalysis`; completion notified via SNS to a small continuation in `reader`. Most receipts are single-page, so the sync path dominates.
- **Formats.** JPEG, PNG, and PDF natively; HEIC from phones is converted to JPEG in `intake-email` / `intake-upload` before storage. No DOCX path — receipts are images, not documents.

Queue and Function URL config

- **`ro-intake`** — standard SQS queue; visibility timeout 180 s (over the reader's 120 s timeout); redrive to `ro-intake-dlq` after 3 receives. The reader's SQS event source uses batch size 1 so one bad image can't fail a batch.
- **`intake-upload` Function URL** — `AuthType: NONE`, per-device bearer token verified in code; CORS limited to the upload page origin; request body size cap enforced.
- **`action-handler` Function URL** — `AuthType: NONE`; Slack signing-secret verification on every request; rejects requests older than 5 minutes to block

replays.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `receipts.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `ro-inbound-rules`: one rule with recipient `receipts@your-company.com` → spam scan → S3 PUT to `s3://ro-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-email`.
- SES outbound for the monthly summary email: verify a sender identity at `books@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:

- **reader role:** `s3:GetObject` on `ro-receipts` and `ro-rules-source`; `textract:AnalyzeExpense` + `StartExpenseAnalysis` + `GetExpenseAnalysis`; `dynamodb:Query` on the `ro-receipts` dup GSI; `bedrock:InvokeModel` on the Haiku ARN (when categorizer runs in-process); `events:PutEvents` on the default bus.
- **filing role:** `secretsmanager:GetSecretValue` on `ro/google/sa`; `s3:CopyObject` + `PutObject` on `ro-receipts`; `dynamodb:PutItem` on `ro-receipts`; outbound network to `sheets.googleapis.com`.

- **review role:** `secretsmanager:GetSecretValue` on the Slack bot token; `dynamodb:PutItem` on `ro-review`; outbound network to `slack.com`.
- **action-handler role:** `dynamodb:PutItem` on `ro-audit` and `ro-review`; `secretsmanager:GetSecretValue` on the Sheets-API and Slack signing secrets; `s3:CopyObject` on `ro-receipts` and `ro-rejected`; outbound network to `sheets.googleapis.com`.
- **intake-email and intake-upload roles:** `s3:GetObject` / `PutObject` on the receipt and MIME buckets; `sqs:SendMessage` on `ro-intake`; `ssm:GetParameter` on `/ro/upload-tokens/` (upload only).

Slack review flow

Review cards are posted via the `chat.postMessage` Web API with Block Kit blocks: an image block for the receipt, a fields block for the read values, and an actions block with the Approve, Correct, and Reject buttons. Button clicks are sent by Slack to the configured Interactivity request URL, which is the `action-handler` Function URL. `action-handler` verifies the Slack signing secret, parses the `action_id` (`approve`, `correct`, `reject`), opens a modal where needed (Correct opens a pre-filled modal; Approve and Reject are one-tap), and processes the modal submission.

The Slack app needs `chat:write` and `files:read`, plus the Interactivity URL configured. The bot token lives in Secrets Manager under `ro/slack/bot-token`; the signing secret under `ro/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:** `ro-intake-dlq` depth > 0 (a receipt failed to process); reader Textract failure rate > 2% in 24h; action-handler signature-verification failures > 5/hour (might mean the Slack secret rotated).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $30/month threshold, alarm at 80% and 100%, posts to SNS topic `ro-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for the Google Sheets API live in Secrets Manager under `ro/google/sa`. Slack bot token and signing secret under `ro/slack/*`. SES sender identity lives in IAM and the verified-domain config. The confidence threshold, the "always review" category list, the chart of accounts location, and the admin owner all live in Parameter Store under `/ro/config/`; per-device upload tokens under `/ro/upload-tokens/`. 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) running 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 `ro-receipts` and `ro-rules-source` so a bad edit can be rolled back in one click, and give the `ro-intake`

queue a real DLQ from day one so a malformed image never silently vanishes. Total deployable surface: around eight Lambdas, three DDB tables (plus the dup GSI), four S3 buckets, one SQS queue with a DLQ, one EventBridge rule pair on the default bus (plus the Scheduler rules for digest and summary), 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.