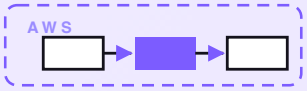
7-PART SERIES · FREE COMPANION



Vendor onboarder

A serverless onboarder that gets a new supplier set up cleanly. It collects the details and documents you need from a new vendor — bank details, tax form, insurance certificate — checks each document is present and not expired, chases anything missing, and hands a complete, verified vendor file to the owner to approve before the vendor is added. Nothing is approved automatically; a human signs off. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Vendor onboarder

PART 1 OF 7

MAY 29, 2026 PART 1 OF 7 · VENDOR ONBOARDER SERIES ~5 MIN READ

A vendor onboarder on AWS for a few dollars a month

A new supplier always owes you the same boring pile of paperwork before you can pay them. Their bank details so the payment goes to the right account. A tax form so your books are clean at year-end. An insurance certificate so you're covered if something goes wrong on a job. A signed agreement. Half the time one piece is missing, one is the wrong year, and the person who chased it left for another job. Setting up a vendor is dull, easy to half-finish, and the gap nobody notices until a payment bounces or an audit finds a vendor with no tax form on file. This post walks through the design of a small onboarder that collects everything a new vendor owes you, checks each document is actually there and not expired, chases what's missing — and then hands a complete, verified file to a human to approve before the vendor goes anywhere near your accounting system.

KEY TAKEAWAYS

- Three ways to start a vendor: a quick form, a forwarded email from the supplier, and a Drive sheet row.
- Every vendor file ends in one of four states on each tick: collecting, ready-to-approve, approved, or rejected.
- Each required document is checked two ways: is it present, and is it still in date?
- A daily tick chases only what's still missing — nudge at day 3, follow-up at day 7, escalate at day 14.
- Nothing is added automatically. A human approves the finished file. Designed on AWS for about $2.20/month.

The whole system on one page

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

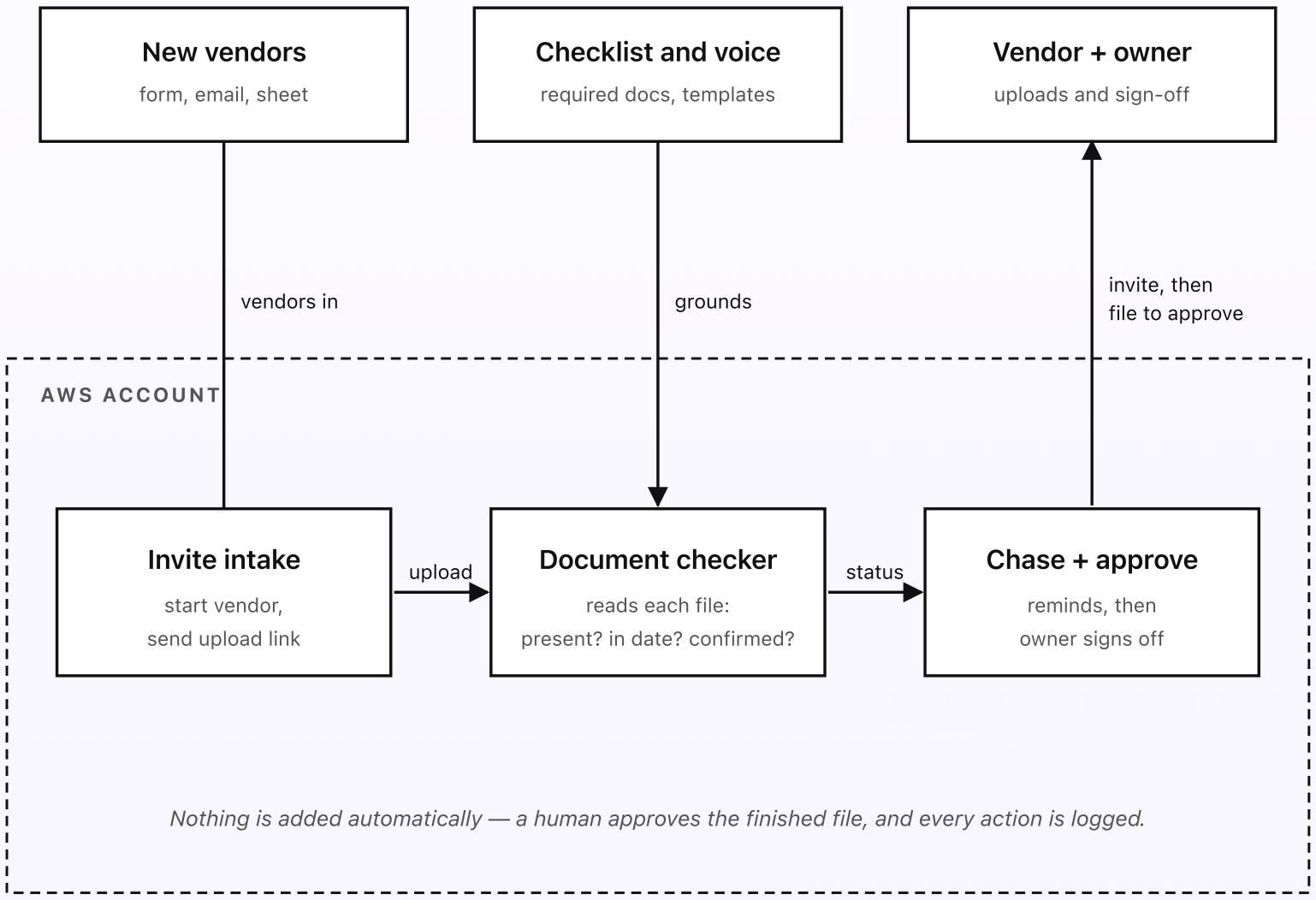

New vendors
form, email, sheet
Checklist and voice
required docs, templates
Vendor + owner
uploads and sign-off
vendors in
grounds
invite, then
file to approve
AWS ACCOUNT
Invite intake
start vendor,
send upload link
upload
Document checker
reads each file:
present? in date? confirmed?
status
Chase + approve
reminds, then
owner signs off
Nothing is added automatically — a human approves the finished file, and every action is logged.

Fig 1. Three inputs outside, three pieces inside AWS. Vendors start from a form, a forwarded email, or a Drive sheet. The Document checker reads each uploaded file and decides present-or-missing and in-date-or-expired. The Chase and approve piece reminds the vendor about only what is left, then sends the complete file to a human to sign off.

What you set up once (the outside)

- **New vendors.** A vendor is started in one of three ways, all covered in Part 2: an owner fills a short web form (vendor name, contact email, vendor type), a salesperson forwards the supplier's first email to a dedicated address, or somebody adds a row to a Google Sheet in a Drive folder. Whichever lane it comes from, the result is the same: a vendor folder is created and the supplier gets a private link where they upload their documents.
- **A checklist folder.** Two short Google Docs in a Drive folder. The *checklist* doc says which documents each vendor type owes — a contractor owes bank details, a tax form, and an insurance certificate; a software supplier might owe only bank details and a tax form. It also holds the rules for what counts as in-date (an insurance certificate must not be past its expiry date; a tax form must be the current year's version). The *voice* doc holds the invite message and the reminder messages, so what the vendor reads stays in your tone.
- **Vendor and owner.** The supplier contact uploads documents through the private link and receives any reminders. The internal owner gets the complete, verified file at the end with a single Approve button. The file they see lists every required document, whether it's present, whether it's in date, and a link to view each one.

What runs on every step (the inside)

- **The invite intake.** Takes a new vendor from any of the three lanes, creates a folder in S3 for the documents and a row in DynamoDB for the checklist status, and sends the private upload link by email. The link opens a small upload page (a Lambda Function URL, not a heavy web app) where the vendor drops their files one at a time.
- **The document checker.** When a vendor uploads a file, it reads it once. Textract (a managed service that reads text out of PDFs and images) pulls the words off the page, and Bedrock Haiku 4.5 reads those words and proposes what kind of document it is and the few fields that matter — the tax-ID on a tax form, the expiry date on an insurance certificate. Plain Python then decides two things: is this required document now present, and is it still in date? A person confirms the read before the item counts as done, because a misread expiry date is worse than a missing one.
- **The chase and approve piece.** Runs once a day. For each vendor still collecting, it looks at what's outstanding and sends a reminder that lists only the missing items. The cadence is gentle — a nudge at day 3, a follow-up at day 7, an escalation to your internal contact at day 14. When every required item is present, in date, and confirmed, the vendor file flips to *ready-to-approve* and the owner gets one message with the whole file and an Approve button. Nothing is written to your system of record until the owner taps it.

In plain words

You agree to start using Bright Spark Electrical for site work. You open the form, type their name and the foreman's email, pick "contractor," and submit. Bright Spark gets an email: "To get set up as a supplier, please upload four things — your

bank details, your W-9, your certificate of insurance, and the signed agreement." They upload the bank details and the W-9 that afternoon. The checker reads both, you confirm them, and two items go green. The insurance certificate and the signed agreement are still missing, so three days later Bright Spark gets a friendly reminder that lists only those two. The certificate arrives; the checker reads its expiry date and notices it lapsed last month, so that item stays red with a note. A day-7 follow-up asks for a current certificate and the agreement. Both arrive, both check out, and the file flips to ready-to-approve. You get one message: the whole Bright Spark file, every item green, with an Approve button. You tap it, and only then is Bright Spark written into your accounting system as a payable vendor.

The cost of running this is about $2.20 a month at SMB volume. The cost of *not* running it is the payment that goes to a stale bank account, the year-end scramble for a tax form nobody collected, and the uninsured contractor who was already on a roof before anyone checked.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every vendor owes a known list. The checklist doc, not code, says what each vendor type must provide.
- Two checks per document, always: is it present, and is it still in date? A stale certificate fails the second one.
- A human confirms every read. A misread tax-ID or expiry date is caught before it counts as done.
- Reminders list only what's left. A vendor who has done most of it is never asked for everything again.
- Nothing is added automatically. The owner approves the finished file; only then is the vendor written to the system of record.
- Every action is logged. Audit a vendor next year and you can see who approved it, when, and on what evidence.

Why this shape

Most small teams onboard vendors in one of three ways: a person who emails the supplier back and forth until the paperwork is in, a shared folder that's missing half its files, or not really at all — the vendor gets paid and the documents get collected "later," which means never. The person works until they're busy, and the weeks they're busiest are exactly when new vendors pile up. The folder is storage, not a system: it holds what arrived but says nothing about what's missing or out of date. And "not at all" is how a business ends up paying a vendor whose insurance lapsed and whose tax form was never on file.

The setup above keeps the simple parts simple — a checklist in a doc, files in a folder — but adds a small system that *watches* each vendor file and acts only when something is missing or expired. Invites go out the moment a vendor is started. Reminders are polite and list only what's left. Each document is checked for being present and for being in date. And the finished file always lands in front of a human before anything is added. The onboarder is invisible once a vendor is set up; it only shows up while a supplier still owes you paperwork.

The next four posts walk through each piece in turn: how a vendor gets invited, how a vendor document gets checked, how a vendor gets chased for missing items, and how a vendor file gets approved. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 29, 2026 PART 2 OF 7 · VENDOR ONBOARDER SERIES ~4 MIN READ

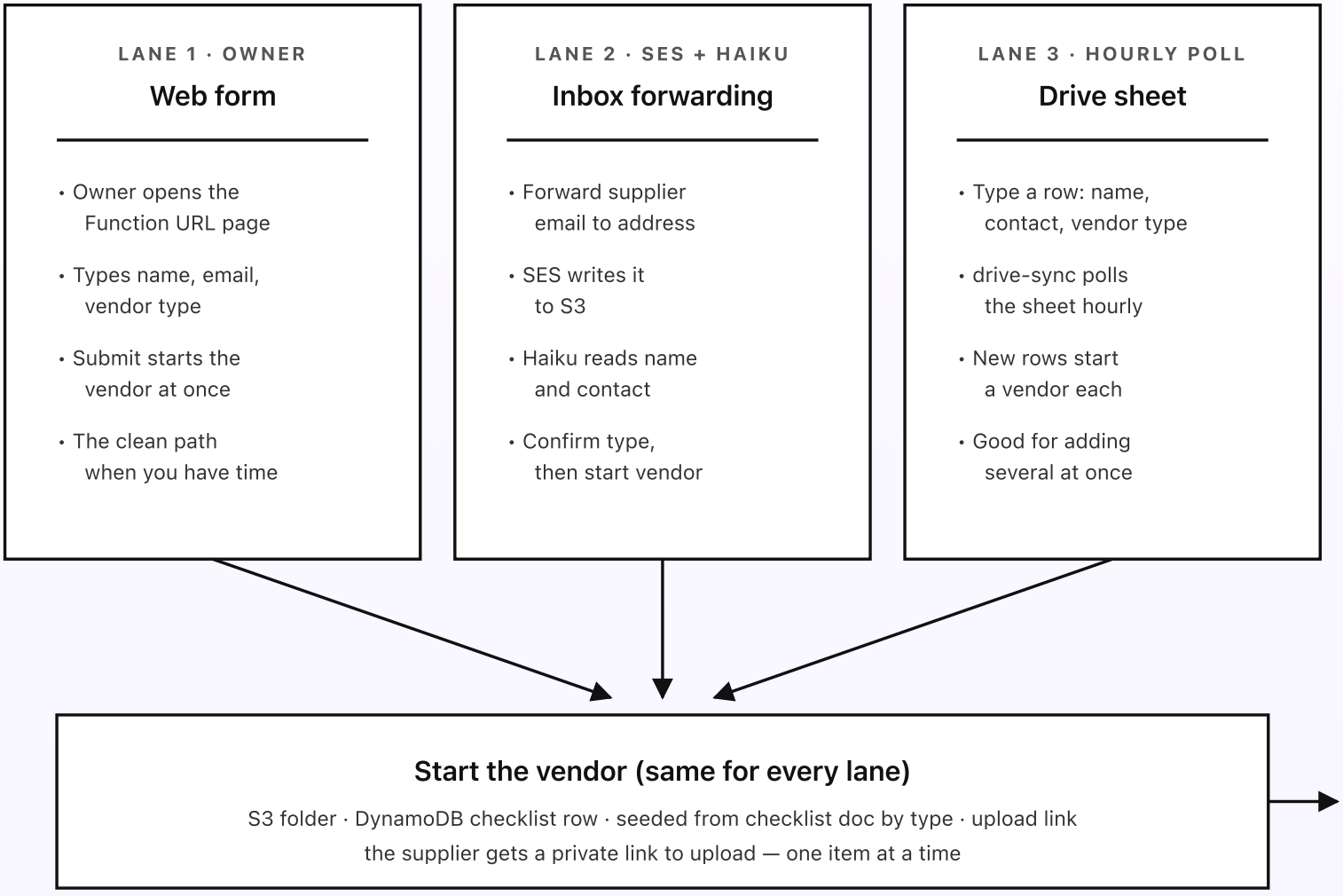
How a vendor gets invited

The onboarder can only chase a vendor it knows about. So the first job is starting the vendor — creating its folder, its checklist, and sending the supplier a link where they can upload. There are three ways a vendor gets started: an owner fills a short form, a salesperson forwards the supplier's first email, or somebody adds a row in the Drive sheet. The first one is the clean path. The other two exist because in real life people don't open a form for a supplier they just agreed to use over the phone.

KEY TAKEAWAYS

- Three intake lanes start one vendor file: a web form, an inbox-forwarding lane, and a Drive sheet row.
- The forwarded-email lane uses Bedrock Haiku 4.5 to read the supplier's name and contact from the message.
- Every started vendor gets a folder in S3, a checklist row in DynamoDB, and a private upload link.
- The vendor type (contractor, software, supplier) decides which documents the checklist asks for.
- The upload link is a Lambda Function URL — a small page, not a heavy web app.

Three lanes into one vendor file



Whichever lane starts a vendor, the result is identical — one folder, one checklist, one upload link.

Fig 2. Three lanes converge on one action: start the vendor. Whether it came from the form, a forwarded email, or a sheet row, the onboarder creates the folder, seeds the checklist for that vendor type, and sends the supplier a private upload link.

Lane 1: the web form

The simplest lane. The owner opens a small page — a Lambda Function URL, so it's a single function serving one form, not a hosted web app — and types three things: the vendor's name, the contact email, and the vendor type (contractor, software supplier, goods supplier, and so on). On submit, the `intake` Lambda creates a folder in S3 for the vendor's documents, writes a checklist row in DynamoDB seeded from the checklist doc for that type, and emails the supplier their private upload link.

The vendor type is the only field that changes what happens next. A contractor's checklist asks for bank details, a tax form, and an insurance certificate. A software supplier's might ask only for bank details and a tax form. The checklist doc holds those lists in plain prose, so adding a new type is an edit, not a deploy.

Lane 2: inbox forwarding (the lane most teams actually use)

Set up a dedicated inbound address — something like `vendors@your-company.com` — via Amazon SES. When a salesperson agrees a deal over email and the supplier replies with their details, the salesperson just forwards that email to the address. SES writes the raw message to `s3://vo-raw-mime/`. The S3 write triggers the `intake` Lambda, which calls Bedrock Haiku 4.5 to read the supplier's company name and the best contact email out of the message.

The model prompt is short: “Read this email. Return JSON with the supplier company name and the contact email. Do not invent details that aren’t there.” The result goes to a small message to the owner with the proposed vendor and a one-tap choice of vendor type. Only on that confirmation is the vendor file created. The reason a human picks the type is that the type decides the whole checklist, and guessing it wrong means asking the supplier for the wrong documents.

Lane 3: the Drive sheet

Some teams prefer to line up several vendors at once — after a trade show, or when switching suppliers across a category. For them, the registry is a Google Sheet with one row per vendor: name, contact email, vendor type. A small `drive-sync` Lambda runs hourly, reads the sheet via the Sheets API, and starts a vendor for any new row it hasn’t seen before. The same create-folder, seed-checklist, send-link steps run for each one.

The sheet doubles as the owner-facing view later: the same `drive-sync` Lambda writes each vendor’s checklist status back into the sheet, so a manager who lives in spreadsheets can see at a glance which vendors are still collecting and which are ready to approve.

What the vendor sees

The supplier gets one email: a short, friendly note in your voice that says what you need and why, and a button that opens their private upload page. The page is theirs alone — the link carries a long random token, so nobody can guess another vendor’s page — and it lists each required document with a place to drop a file. They can upload one item now and the rest later; the page remembers what’s

already in. There's no account to create and no password to set, which is exactly what a busy supplier wants.

Funneling all three lanes into one start action means there is exactly one vendor file per supplier, no matter how it began. The next post follows a single uploaded document through the checker: how it gets read, how the system decides present-or-missing and in-date-or-expired, and why a human confirms before any item counts as done.

PART 3 OF 7

MAY 29, 2026 PART 3 OF 7 · VENDOR ONBOARDER SERIES ~5 MIN READ

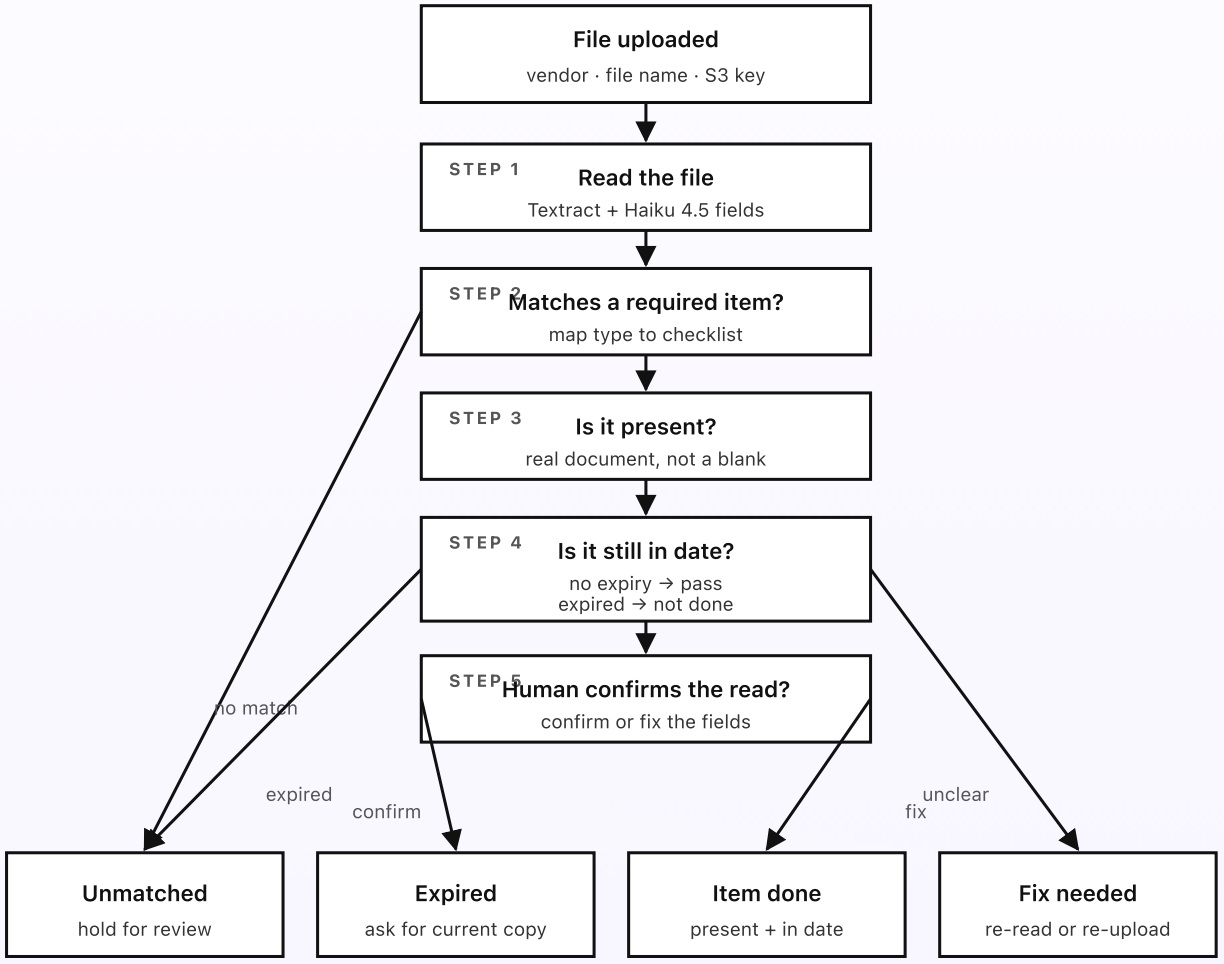
How a vendor document gets checked

A vendor drops a PDF on their upload page. Now the onboarder has to work out what it is, whether it fills a gap in the checklist, and whether it's any good — a certificate of insurance that expired last month is present but useless. The reading uses a model; the deciding is plain Python; and a person confirms before the item turns green. This post follows one uploaded file from the moment it lands to the moment it counts.

KEY TAKEAWAYS

- An upload triggers the checker once — Textract reads the page, Haiku 4.5 reads the fields.
- Plain Python matches the file to a required item and checks both present and in-date.
- An expired insurance certificate is present but fails the date check — it stays not-done.
- A person confirms the read before the item counts; a misread date is caught here, not later.
- The checker calls a model only on upload — never on the daily chase tick.

The check flow, per uploaded file



The model only reads — plain Python decides present and in-date, and a human confirms before it counts.

Fig 3. The checker's flow for one uploaded file. Read it once, match it to a required item, check present and in-date, and let a human confirm. Only then does the item turn green.

Reading the file: a model, used narrowly

When a file lands in the vendor's folder, the S3 write triggers the `checker` Lambda. It runs Amazon Textract on the file — Textract is a managed service that reads the text and tables out of PDFs and images, so a scanned certificate works as well as a clean PDF. The extracted text then goes to one Bedrock Haiku 4.5 call with a tight prompt: "Here is the text of a document a vendor uploaded. Tell me which of these it is — bank details, tax form, insurance certificate, signed agreement, or unknown — and pull these fields if present: the tax-ID, the policy expiry date, the bank account name. Return JSON only. Do not invent a date that isn't in the text."

That's the only place a model is involved. It reads; it doesn't decide anything. Everything after this is plain Python comparing what the model pulled against the checklist rules.

Two checks: present, and in date

A document can fail in two different ways, and the checker tests for both.

Present. Does this file actually fill a gap on the checklist? The Python maps the model's proposed type to a required item for this vendor. If the vendor owes an insurance certificate and the upload reads as one, that item is now a candidate to

mark present. If the file reads as something the vendor doesn't owe — or as "unknown" — it's held as *unmatched* for a person to look at, rather than silently dropped.

In date. Some documents have an expiry; a certificate of insurance is the clearest case, and some tax forms are only valid for the current year. The checklist doc says which items carry a date rule. For those, the Python compares the expiry date the model pulled against today. An insurance certificate that expired last month is *present* but fails the date check, so the item stays not-done with a note: "Insurance certificate received, but it expired on 2026-04-30 — please upload a current one." This is the check that catches the most common real-world problem: a supplier who genuinely sent a certificate, just last year's.

Why a human confirms every read

Even a good model misreads a smudged date or a tax-ID with an extra digit now and then. For a vendor file, a wrong read is dangerous in a quiet way: it can mark an item done when it isn't, and the system would then stop chasing it. So every read goes to a one-tap confirmation. The owner (or whoever set up the vendor) sees the proposed item, the pulled fields, and a thumbnail, with two buttons: *confirm* and *fix*. On *confirm*, the checklist row in DynamoDB marks the item present, in date, and confirmed. On *fix*, they correct the field — usually a date — and the corrected value is what's stored.

The confirmation is deliberately lightweight, because most reads are right and you don't want to make the common case slow. But the principle holds: a document counts as done only when a person has agreed it's the right document and the

read is correct. The cost of a wrong “done” here is a vendor approved on paperwork that wasn’t really valid.

State that makes the checklist trustworthy

The vendor’s checklist lives in one DynamoDB row per vendor, with a small map for each required item: `(item, status, present, in_date, expiry, confirmed_by, confirmed_at)`. Every upload updates exactly one item’s entry. Because the state is explicit, the chase tick in the next post never has to re-read a document — it just looks at which items are still not *done*. Re-uploading a replacement (a current certificate over an expired one) overwrites that one item’s entry and leaves the rest alone.

Why the daily tick uses no model

The checker calls Textract and Bedrock only when a file is uploaded — a handful of times per vendor, total. The daily chase tick in Part 4 reads the same DynamoDB row and calls no model at all; it just compares the checklist state against the calendar. That split keeps the cost down (you pay to read a document once, not every day) and keeps the part that runs every day completely predictable.

Next post: how a vendor gets chased for the items still missing — the daily tick, the four moves, and how a reminder lists only what’s left.

PART 4 OF 7

MAY 29, 2026 PART 4 OF 7 · VENDOR ONBOARDER SERIES ~5 MIN READ

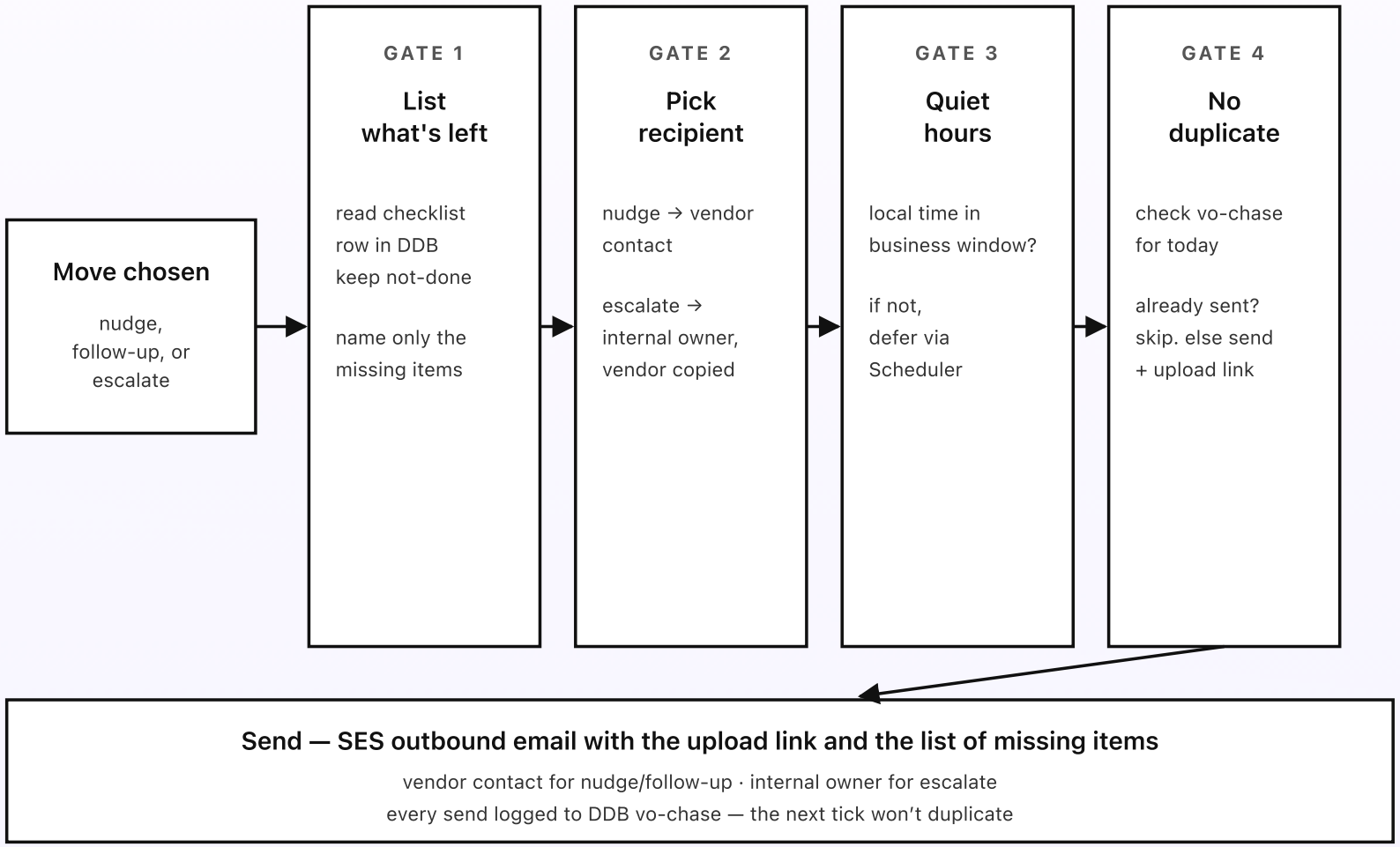
How a vendor gets chased for missing items

Most vendors don’t upload everything in one go. They send the easy items first and forget the rest, and the rest is what slips. Once a day, the onboarder looks at every vendor still collecting, works out which documents are still missing, and decides whether to do nothing, send a gentle nudge, send a firmer follow-up, or escalate to your own team. The decision is plain Python. Four small guardrails sit between “this vendor owes something” and the reminder actually landing — so the chase is polite, well-timed, lists only what’s left, and never doubles up.

KEY TAKEAWAYS

- The chase runs once a day via EventBridge Scheduler at 9am local time.
- Four moves per vendor, every tick: done, nudge, follow-up, escalate — based on days since the invite.
- The cadence lives in the checklist doc — nudge at day 3, follow-up at day 7, escalate at day 14.
- A reminder lists only the outstanding items, so a vendor isn't asked for things already in.
- Quiet hours and a no-duplicate check keep the chase from being noise. No model on the tick.

Four guardrails on every chase



Every gate is a deterministic check — no model calls, and a vendor is never asked for what's already in.

Fig 4. Four guardrails between the move and the sent reminder. List only what's left. Pick the right recipient. Honor quiet hours. Don't duplicate. Then send via email and log it so the next tick doesn't repeat the message.

Four moves, always

Once a day, an EventBridge Scheduler rule fires the `chase` Lambda. It reads every vendor still in the *collecting* state, and for each one computes the days since the invite went out and which items are still not done. Then it lands in exactly one of four buckets.

- **Done.** Every required item is present, in date, and confirmed. The vendor moves to *ready-to-approve* and the chase stops. Part 5 picks up here. No reminder is sent.
- **Nudge.** The first cadence step has arrived (day 3 by default) and items are still missing. Send a friendly reminder that lists only the outstanding documents and re-sends the upload link.
- **Follow-up.** A later step arrived (day 7) with items still missing. Send a firmer reminder that mentions the first one went out, so the vendor doesn't feel it's the first time you've asked.
- **Escalate.** The final step arrived (day 14) and the vendor still hasn't finished. Tell your own internal contact — the person who wanted this vendor — so a human can pick up the phone. The vendor's contact is copied; the escalation isn't a substitute for asking them, it's a flag for your side.

The cadence numbers (3, 7, 14) live in the checklist doc, not the code, so a rep can make a slow category gentler or a rush vendor faster without a deploy.

Gate 1: list only what's left

The single thing that makes a chase feel respectful instead of robotic is that it asks for the right things. Gate 1 reads the vendor's checklist row and keeps only the items that aren't done — missing entirely, or present-but-expired. The reminder names those and nothing else. A vendor who has already sent their bank details and tax form gets a reminder about the insurance certificate alone, with a line like "Thanks for the two you've sent — we just need your certificate of insurance to finish." An expired item gets its own honest note: present, but out of date, please send a current copy.

Gate 2: pick the right recipient

A nudge and a follow-up go to the vendor's contact — the person who's actually holding the documents. An escalate is different: it goes to your internal owner, the person who started the vendor, because at day 14 the useful move is a human-to-human conversation, not another automated email. The vendor is copied so nothing happens behind their back. The escalate message carries the full history — what's in, what's still missing, and the dates of the earlier reminders — so your colleague has everything they need before they reach out.

Gate 3: quiet hours

The tick runs at 9am local, so most reminders land in business hours already. But deferred sends and re-runs can fall outside the window. Gate 3 reads the quiet-hours setting (default 6pm to 8am) and, if the current local time is in the quiet window, creates a one-off EventBridge Scheduler rule that fires at the next

business-hour minute and exits without sending. A reminder is a finite kind of goodwill; landing one at 11pm spends it for nothing.

Gate 4: no duplicate

The last gate reads the `vo-chase` DynamoDB table to confirm a reminder for this vendor and this move hasn't already gone out today. If it has, the send is skipped. This is what makes the tick safe to re-run: if the Lambda retries, or someone triggers it manually, no vendor gets the same message twice. After a successful send, a row is written to `vo-chase` with the vendor, the move, the date, and the list of items that were outstanding — which is also what the weekly digest and the audit trail read later.

Why the chase uses no model

The chase could call a model to write a warmer message each time. It doesn't. The cadence and the recipient choice are simple rules, and a model in that loop would add cost and variance for no real gain — the reminder templates in the voice doc already sound like you. The only place a model reads anything is the document checker in Part 3, on upload. The daily chase is plain Python comparing a checklist against a calendar.

Next post: what happens once a vendor finishes — how the complete file reaches the owner, and the three things the owner can do before the vendor is ever added.

PART 5 OF 7

MAY 29, 2026 PART 5 OF 7 · VENDOR ONBOARDER SERIES ~5 MIN READ

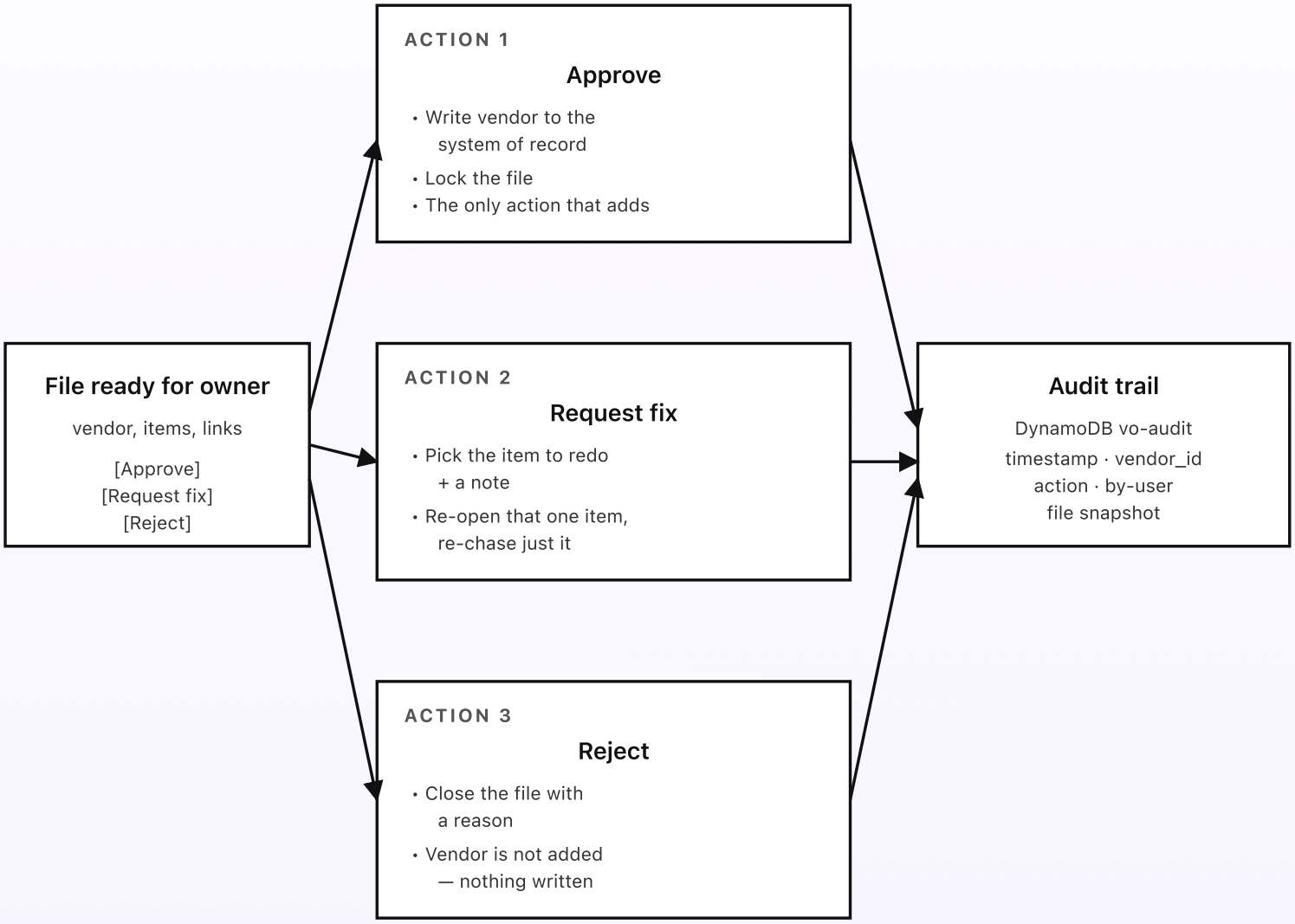
How a vendor file gets approved

Bright Spark Electrical just uploaded their last document. Every required item is present, in date, and confirmed, so the file flips to ready-to-approve and a message lands in the owner's inbox: the whole Bright Spark file, every item green, with an Approve button. What happens when they tap it? And what if they don't? This post walks through the three things the owner can do with a finished file — approve, request a fix, reject — and how the system of record, the file state, and the audit trail all stay in sync. This is the human-in-the-loop step the whole system is built around: nothing is added without it.

KEY TAKEAWAYS

- Three actions on the finished file: *approve* (write to the system of record), *request a fix* (send back one item), *reject*.
- Approve is the only action that adds the vendor — and it only ever happens on a human tap.
- Request-a-fix re-opens one item and re-starts the chase for just that document.
- Every action writes an audit row with who acted, when, and the file they saw.
- The Approve button is backed by a Function URL; the click is signed and logged.

Three actions on the finished file



Only Approve adds the vendor — and it never happens without a human tap. Every action is logged.

Fig 5. Three actions on the finished file, three different effects. Approve writes the vendor to your system of record and locks the file. Request-a-fix re-opens one item. Reject closes it. Every action writes to the audit trail.

Action 1: approve (the common one)

The owner opens the message, sees every item green, clicks through to spot-check the certificate and the bank details, and taps *Approve*. That tap submits to a Function URL Lambda, and three things happen, in order. First, the vendor is written to your system of record — your accounting tool — via its API: the name, the bank details, the tax-ID, and a link back to the document file. Second, the vendor folder is locked so the approved documents can't be quietly swapped later. Third, an `action: approved` row is written to `vo-audit` with the user, the timestamp, and a snapshot of the file exactly as the owner saw it.

This is the one action that adds the vendor, and it is the whole reason the system exists in its careful shape. Everything before it — the checking, the chasing, the confirmations — is in service of putting a complete, verified file in front of a person so that this tap is a quick yes, not a leap of faith. The system never makes this decision itself.

Action 2: request a fix (the small redo)

Sometimes the file is complete on paper but something looks off to the human eye — the bank account name doesn't match the company name, the certificate is for the wrong type of cover, the agreement is signed by the wrong person. The owner

taps *Request fix*, picks the one item that needs redoing from a short list, and adds a note explaining what's wrong.

The Function URL Lambda re-opens just that one item on the checklist — marking it not-done again — and sends the vendor a message: "Almost there. We just need to redo one item: [the note]." The chase from Part 4 re-starts for that single document, on a fresh cadence, while every other item stays exactly as it was. The vendor doesn't have to re-send anything that was already fine. When the new copy arrives and the checker clears it, the file returns to ready-to-approve and the owner gets the message again. Request-a-fix is the escape hatch that keeps "the documents are all here but one is wrong" from forcing a full restart.

Action 3: reject (the clean no)

Occasionally a vendor shouldn't be added at all — the deal fell through, the supplier turned out not to be a fit, or a check failed in a way that can't be fixed. The owner taps *Reject* and gives a reason. The Function URL Lambda closes the file, marks it rejected, and stops all chasing. Nothing is written to the system of record — the vendor simply never gets added. The documents stay in S3 for the retention period so there's a record of what was collected and why it was turned down, and an `action: rejected` row goes to `vo-audit` with the reason.

Reject is deliberately as easy as approve. A system that makes it hard to say no quietly pushes people toward yes, which is the opposite of what a human-in-the-loop gate is for.

Every action is logged, every approval is provable

The `vo-audit` table records every approve, fix-request, and reject with the user who acted, the timestamp, and a snapshot of the file as it stood at that moment — which documents were present, their expiry dates, who confirmed each read. That snapshot is the thing that matters at audit time. When somebody asks, a year from now, “who approved Bright Spark and what did they see?”, the answer is one row: the person, the date, and the exact file they signed off on. The documents themselves live in versioned S3, so the certificate that was valid at approval time is still there even if a newer one replaced it since.

This is the difference between a folder of files and a vendor record you can stand behind. The folder tells you what you have now. The audit trail tells you what was true when the decision was made, and who made it.

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 most of the bill is reading documents, not chasing them.

PART 6 OF 7

MAY 29, 2026 PART 6 OF 7 · VENDOR ONBOARDER SERIES ~3 MIN READ

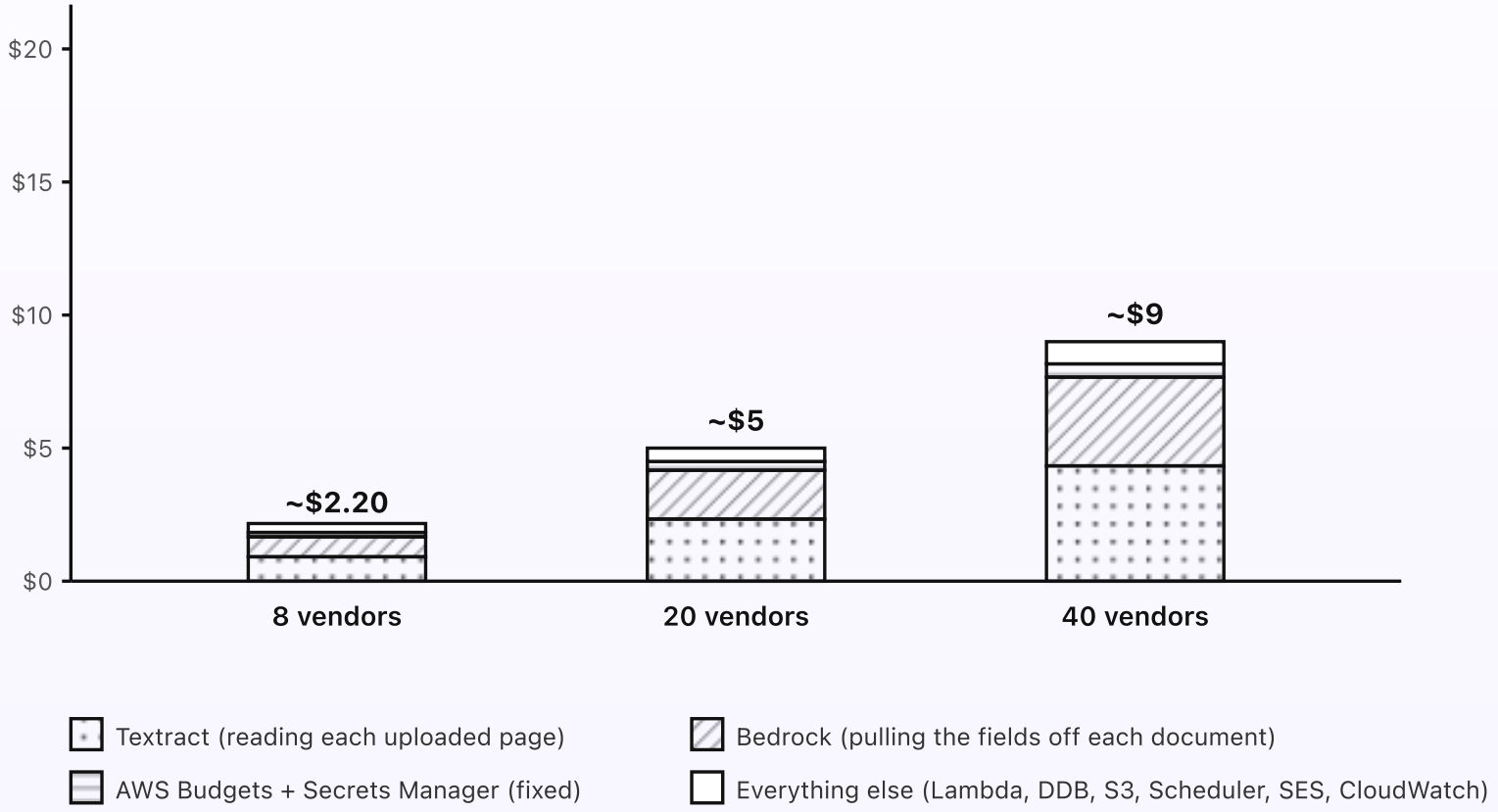
What the vendor onboarder costs

The onboarder is one of the cheaper systems in this whole series. The daily chase tick reads a small table, does some date arithmetic, and sends a handful of emails. It calls no models. The one part that costs real money is reading the documents a vendor uploads — Textract reads each page and Bedrock pulls the fields — and that fires only a few times per vendor, total. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.20/month at typical SMB volume (around 8 new vendors a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The daily chase tick costs pennies — no model calls.
- Textract and Bedrock fire only when a vendor uploads a document — a few times per vendor.
- At 20 new vendors a month the bill is around $5. At 40 it’s around $9.

Cost at three volumes



Reading documents is the dominant cost — and even that is a few cents per vendor.

Fig 6. Monthly cost at three new-vendor volumes. Textract and Bedrock are the biggest slices because reading each uploaded document is the only real per-vendor cost. The chase tick, the emails, and the table reads barely register.

Where the dollars actually go

Textract (the bulk). Every uploaded document is read once. Textract is priced per page, and a typical vendor sends a few short documents — bank details, a tax form, a certificate, an agreement — so call it ten to twenty pages per vendor. At 8 new vendors a month that's a dollar or so; at 40 vendors it scales to a few dollars. This is the single largest line, and it only happens when a vendor actually uploads something.

Bedrock (the second slice). Each uploaded document gets one Haiku 4.5 call to read its type and fields: a few thousand input tokens (the Textract output) and a few hundred output tokens (the proposed fields as JSON), so a fraction of a cent per document. Across a few documents per vendor, it's cents per vendor. Like Textract, it's tied to uploads, not to the calendar.

Lambda runtime. The chase tick runs once a day and iterates the vendors still collecting — a few milliseconds each. Add the intake Lambda, the checker Lambda (which fires per upload), the Function URL Lambdas for the upload page and the approve button, and the hourly drive-sync — the Lambda total still lands well under a dollar at all three volumes.

DynamoDB on-demand. Three small tables: `vo-vendors` (one row per vendor, the checklist), `vo-chase` (one row per reminder), `vo-audit` (one row per action). Reads and writes are a handful per vendor. Pennies a month at any of these volumes.

S3 + Storage. The vendor documents themselves plus the raw inbound MIME from forwarded emails. A few megabytes per vendor at most. Effectively free.

SES. Inbound for the forwarding lane: $0.10 per thousand received messages. Outbound for invites and reminders: $0.10 per thousand sent. A few emails per vendor means a couple of cents a month even at 40 vendors.

EventBridge Scheduler. The daily chase rule, the hourly sync, and the occasional deferred-send one-off. A few invocations a day. Pennies.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the upload page and the approve button.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The chase tick sleeps 23.99 hours a day.
- **A Knowledge Base.** The checklist is structured rows, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the tick.** The daily chase is plain Python. Textract and Bedrock fire only when a vendor uploads a document.

How the cost scales

The bill tracks the number of new vendors, not the number of vendors you already have. Once a vendor is approved, the system stops reading and chasing — it costs nothing to keep an approved vendor on file. So the cost is driven entirely by intake: at 80 new vendors a month the bill is around $18; at 160 it's around $36,

still almost all of it Textract and Bedrock reading uploads. A business onboarding that many suppliers a month is unusual for an SMB, and even then the cost is a rounding error against a single mis-paid invoice or a lapsed-insurance claim.

Set an AWS Budgets alarm at $20/month so anything unusual — a vendor uploading a 200-page PDF, a retry loop — pages you before the bill matters. The normal-volume bill stays well under that ceiling.

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 EventBridge Scheduler config.

PART 7 OF 7

MAY 29, 2026 PART 7 OF 7 · VENDOR ONBOARDER SERIES ~8 MIN READ

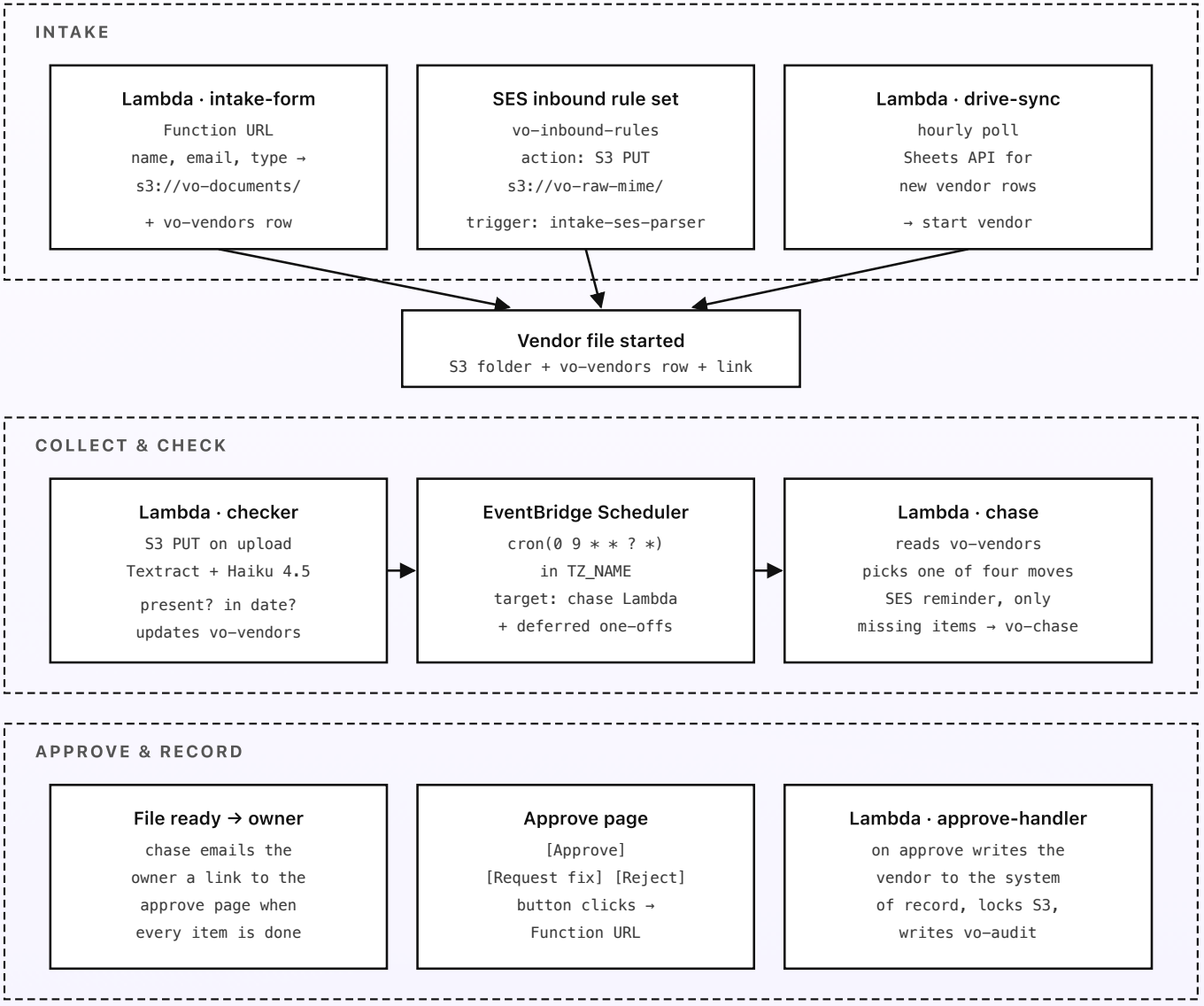
Engineering reference: the vendor onboarder architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, EventBridge Scheduler config, the DynamoDB schemas, and the Function-URL upload and approve 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 cross-Region inference, Textract, and EventBridge Scheduler are all available 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 vendor onboarded a day late, not a regional outage. One AWS account dedicated to the onboarder (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Nothing is added without a human Approve — and every interaction is logged to vo-audit.

Fig 7. AWS topology, in three regions of the diagram: intake (three lanes start a vendor file), collect and check (uploads read on arrival, a daily chase for the rest), approve and record (the owner signs off and the vendor is written to the system of record). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

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

- **`intake-form`** — Lambda Function URL, public with `AuthType: NONE`. Serves the owner-facing start form (GET) and handles the submit (POST). On submit, creates the vendor folder under `s3://vo-documents/<vendor-id>/`, writes a `vo-vendors` row seeded from the checklist doc for the chosen type, and triggers the invite email via SES. The upload page for the vendor is served by the same function under a token-scoped path. Memory: 256 MB. Timeout: 15 s.
- **`intake-ses-parser`** — S3 PUT trigger on `s3://vo-raw-mime/`. Parses the forwarded MIME, calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to read the supplier name and contact, and posts an owner confirmation message with a one-tap vendor-type choice. On confirm, starts the vendor the same way `intake-form` does. Memory: 256 MB. Timeout: 30 s.
- **`drive-sync`** — EventBridge Scheduler target, hourly. Uses the Google Sheets API (service-account credentials in Secrets Manager under `vo/drive/sa`) to

read new vendor rows from the start sheet and start a vendor for each. Also writes each vendor's checklist status back to the sheet for the owner-facing view. Memory: 256 MB. Timeout: 30 s.

- **`checker`** — S3 PUT trigger on `s3://vo-documents/`. Runs Textract via `StartDocumentTextDetection` + `StartDocumentAnalysis` (asynchronously to handle multi-page documents). On Textract completion (via SNS notification), calls Bedrock Haiku 4.5 to read the document type and fields, then plain Python decides present-and-in-date against the checklist rules. Posts a one-tap confirmation to the owner; on confirm, marks the item done in `vo-vendors`. For DOCX uploads (Textract doesn't accept them), falls back to `python-docx`; XLSX uses `openpyxl`. Both packages are stable and widely used in 2026 though lightly maintained — acceptable for a path that runs a few times per vendor; if precision becomes a concern, the active fork `python-docx-oss` is a drop-in alternative. Memory: 512 MB. Timeout: 60 s.
- **`chase`** — EventBridge Scheduler target, daily at 9am local time (the schedule expression runs in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). Reads `vo-vendors`, and for each vendor in the *collecting* state computes days-since-invite and the outstanding items, picks one of four moves (done/nudge/follow-up/escalate), and sends a reminder via SES `SendRawEmail` listing only the missing items. On quiet-hours defer, creates a one-off EventBridge Scheduler rule. Writes a row to `vo-chase` after each send. When a vendor reaches *done*, emails the owner the approve-page link. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`approve-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a signed token on the request. Serves the owner's approve page (GET) and handles Approve/Request-fix/Reject (POST). On *approve*, writes the vendor

to the system of record via its API, sets an S3 Object Lock-style write-protection on the vendor prefix, marks the vendor approved, and writes `vo-audit`. On *request-fix*, re-opens one checklist item and re-arms the chase for it. On *reject*, closes the file with a reason. Memory: 256 MB. Timeout: 15 s.

- **`digest`** — EventBridge Scheduler target, weekly Monday 9am. Reads `vo-vendors` and `vo-chase`; sends the owner a summary of vendors collecting, ready-to-approve, and approved that week. No Bedrock; a plain summary table. Memory: 256 MB.

Storage

- **DynamoDB** · **`vo-vendors`** — one row per vendor. PK `vendor_id`; attributes: `name`, `contact_email`, `type`, `internal_owner`, `state` (collecting/ready/approved/rejected), `invited_at`, and a `checklist` map of `item → {status, present, in_date, expiry, confirmed_by, confirmed_at}`. On-demand.
- **DynamoDB** · **`vo-chase`** — one row per reminder sent. PK `(vendor_id, sent_date)`; attributes: `move` (nudge/follow-up/escalate), `recipient`, `missing_items`. On-demand.
- **DynamoDB** · **`vo-audit`** — one row per write action of any kind. PK `(vendor_id, ts)`; attributes: `action` (approved/request_fix/rejected/item_confirmed), `by_user`, `snapshot`. On-demand. No TTL — this is the long-term audit trail.
- **S3** · **`vo-documents`** — one prefix per vendor holding the uploaded documents. Versioning enabled; write-protection applied to a vendor prefix on approval. Lifecycle to Glacier at 90 days; expiry at 7 years.

- **S3 · `vo-rules-source`** — mirrored checklist and voice docs as plain text. Versioning enabled.
- **S3 · `vo-raw-mime`** — raw inbound MIME from forwarded supplier emails. Lifecycle to Glacier at 30 days; expiry at 7 years.

Bedrock and Textract

- **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: `checker` for reading uploaded documents, and `intake-ses-parser` for reading the supplier name and contact off a forwarded email. A heavier `summary`-style narrative isn't needed here; if you later add a quarterly vendor-base report, route that one call to `anthropic.claude-sonnet-4-6-20250930-v1:0` for the longer reasoning.
- **Textract.** `StartDocumentTextDetection` for plain document text and `StartDocumentAnalysis` with the FORMS feature for key-value pairs (tax-ID, account number, expiry date). Async, with completion via SNS to the `checker` Lambda.
- **Embeddings.** Not used. The checklist is structured rows; deterministic lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors. (If you ever add "find similar past vendors," that's when Amazon Titan Text Embeddings V2 at 1024-dim plus S3 Vectors would earn its place — not before.)

EventBridge Scheduler config

- **`vo-daily-chase`** — `cron(0 9 * * ? *)` in the SMB's timezone. Target: `chase` Lambda.
- **`vo-drive-sync`** — `rate(1 hour)`. Target: `drive-sync` Lambda.
- **`vo-weekly-digest`** — `cron(0 9 ? * MON *)` in TZ. Target: `digest` Lambda.
- **One-off rules** — created on the fly by `chase` when a quiet-hours defer is needed. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `vendors.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `vo-inbound-rules`: one rule with recipient `vendors@your-company.com` → spam scan → S3 PUT to `s3://vo-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for invites, reminders, and the approve-page email: verify a sender identity at `onboarding@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:

- **chase role:** `dynamodb:Query` + `GetItem` on `vo-vendors`; `dynamodb:PutItem` on `vo-chase`; `ses:SendRawEmail` from the verified sender identity; `scheduler:CreateSchedule` for the deferred-send one-offs. *No* `bedrock:*`.

- **checker role:** `s3:GetObject` on `vo-documents`; `textract:StartDocumentTextDetection` + `StartDocumentAnalysis` + `GetDocument*`; `bedrock:InvokeModel` on the Haiku ARN; `dynamodb:UpdateItem` on `vo-vendors`; `dynamodb:PutItem` on `vo-audit`.
- **approve-handler role:** `dynamodb:UpdateItem` on `vo-vendors`; `dynamodb:PutItem` on `vo-audit`; `secretsmanager:GetSecretValue` on the accounting-tool API secret; `s3:PutObjectRetention` on the vendor prefix; outbound network access to the accounting tool's API host.
- **intake-form and intake-ses-parser roles:** `s3:PutObject` on `vo-documents` and read on `vo-raw-mime`; `dynamodb:PutItem` on `vo-vendors`; `ses:SendRawEmail`; `bedrock:InvokeModel` (parser only).
- **drive-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `dynamodb:Query` + `PutItem` on `vo-vendors`; outbound network to `sheets.googleapis.com`.

Upload and approve flow

Both vendor-facing and owner-facing surfaces are Lambda Function URLs, never API Gateway. The vendor's upload page is served by `intake-form` under a path carrying a long random per-vendor token (stored on the `vo-vendors` row); the token scopes the page to one vendor so nobody can reach another's documents. Uploads use a short-lived S3 presigned PUT URL minted by the Function URL, so the file goes straight to `vo-documents` and the S3 PUT triggers `checker`. The owner's approve page is served by `approve-handler` under a separate signed token with a short TTL; the Approve/Request-fix/Reject actions POST back to the

same function, which verifies the token before doing anything. There is no account and no password on either side.

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:** chase Lambda failures > 0 in a day (the daily tick has to run); checker failure rate > 1% in 24h; approve-handler token-verification failures > 5/hour (might mean a leaked or stale link).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $20/month threshold, alarm at 80% and 100%, posts to SNS topic `vo-cost-alarm` subscribed to the on-call admin's email.

Config and secrets

Service-account credentials for the Sheets API live in Secrets Manager under `vo/drive/sa`. The accounting-tool API key (for writing approved vendors to the system of record) lives under `vo/system-of-record/api`. The configured timezone, quiet-hours window, chase cadence defaults, and the per-type checklist reference all live in Parameter Store under `/vo/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 `vo-documents` and `vo-rules-source` so a bad upload or doc edit can be rolled back, and version the EventBridge Scheduler timezone setting so you don't accidentally start running the daily chase in UTC after a CI rotation. Total deployable surface: around seven Lambdas, three DDB tables, three S3 buckets, 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.