7-PART SERIES · FREE COMPANION

Supplier bill matcher

A serverless matcher that reads every supplier bill as it arrives, finds the matching purchase order and goods-received note, and checks them line by line — right item, right quantity received, right price. Clean three-way matches clear for a manager's one-tap approval; anything off is flagged with the exact reason. It never pays a bill on its own. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Supplier bill matcher

PART 1 OF 7

JUNE 19, 2026 PART 1 OF 7 · SUPPLIER BILL MATCHER SERIES ~5 MIN READ

A supplier bill matcher on AWS for a few dollars a month

A small business pays more supplier bills than anyone checks carefully. The bill that quietly charges $4.20 a unit when the purchase order said $3.80. The one for 100 boxes when only 84 turned up at the dock. The duplicate that arrives twice and gets paid twice because two people each thought the other had it. The one with no purchase order at all that somebody pays just to make the supplier stop calling. Checking every bill against what you ordered and what actually arrived is slow, dull, and the first thing that slips when the week gets busy. This post walks through the design of a small matcher that reads every supplier bill, lines it up against the purchase order and the goods-received note, checks it line by line, and never pays anything on its own.

KEY TAKEAWAYS

- Three sources for bills: an emailed-PDF lane, a supplier-portal poll, and a manual upload.
- Every bill ends in one of four outcomes: matched, price variance, quantity variance, or no PO.
- The match is a three-way check: the bill against the purchase order and the goods-received note.
- Clean matches clear for one-tap approval; anything off is flagged with the exact line and gap.
- Designed on AWS for about $3.20/month at typical small-business volume. It never pays a bill on its own.

The whole system on one page

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

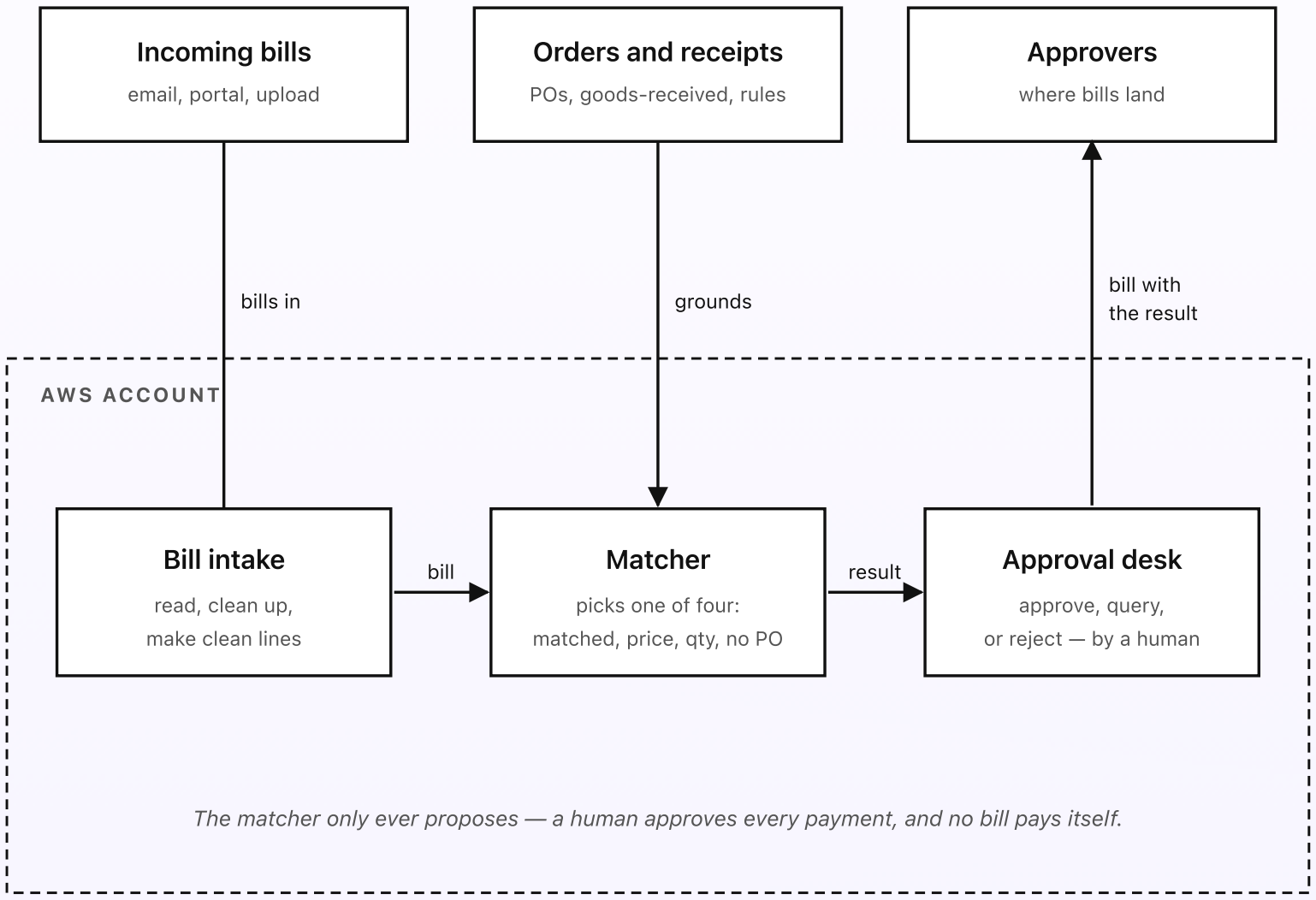

Incoming bills
email, portal, upload
Orders and receipts
POs, goods-received, rules
Approvers
where bills land
bills in
grounds
bill with
the result
AWS ACCOUNT
Bill intake
read, clean up,
make clean lines
bill
Matcher
picks one of four:
matched, price, qty, no PO
result
Approval desk
approve, query,
or reject — by a human
The matcher only ever proposes — a human approves every payment, and no bill pays itself.

Fig 1. Three sources outside, three pieces inside AWS. Bills flow in from email, a supplier portal, and manual upload. The Matcher runs as each bill lands and picks one of four outcomes. The Approval desk sends the bill to the right person, who approves, queries, or rejects.

What you set up once (the outside)

- **Orders and receipts.** A Google Sheet in a Drive folder with two tabs. The *purchase orders* tab has one row per ordered line: PO number, supplier, item code, item name, ordered quantity, agreed unit price, and an open/closed flag. The *goods-received* tab has one row per delivery line: PO number, item code, quantity actually received, and the date it arrived at the dock. You already produce both of these in the normal course of buying things; this just keeps them where the matcher can read them. New bills enter via three lanes covered in Part 2 — an emailed-PDF lane (forward a supplier bill to a dedicated address), a supplier-portal lane (the matcher polls portals that publish bills), and a manual-upload lane.
- **A rules doc.** One short Google Doc in the same Drive folder. It holds the match *tolerances* — how much a price or quantity is allowed to drift before the matcher flags it. A common set: price may be within 2% or $5 (whichever is larger) before it's a price variance; quantity must match the goods-received note exactly, with a small allowance for agreed part-deliveries. The doc also names the *approver* per supplier (or per spending category), the spend threshold above which a second approver is required, and the quiet hours for notifications.
- **Approvers.** The people who actually sign off on paying suppliers — usually a manager or a buyer. Each approver has an email address. Bills land with the

supplier name, the bill total, the per-line result against the PO and the goods-received note, the exact gap on any flagged line, and three buttons: *Approve*, *Query*, and *Reject*. A button click never pays the bill directly — it records a decision and, on approve, marks the bill ready for your normal payment run.

What runs on every bill (the inside)

- **The bill intake.** Three sources feed the matcher. Forward a supplier bill PDF to `bills@your-company.com` and SES writes the raw email to S3. A small portal-poll Lambda checks supplier portals on a schedule and pulls down any new bills. A manual-upload lane lets someone drop a PDF straight into a folder. Whatever the source, the intake runs Textract to read the PDF and one Bedrock Haiku 4.5 call to turn the read text into clean, structured lines — item, quantity, unit price, line total — that the matcher can compare.
- **The matcher.** Runs as soon as a bill is read. It finds the purchase order the bill refers to (by PO number on the bill, or by supplier and item if the number is missing), pulls the matching goods-received lines, and walks the bill line by line. For each line it checks three things: is this the right item, does the billed quantity match what the dock received, and does the billed price match what the PO agreed — each within the tolerance from the rules doc. It then picks one of four outcomes. *Matched*: every line agrees inside tolerance. *Price variance*: items and quantities are fine but a unit price is too high. *Quantity variance*: a billed quantity doesn't match the goods received. *No PO*: there's no purchase order to match against. The match itself is plain Python — no model decides whether to pay.
- **The approval desk.** Takes the outcome and sends the bill to the right approver. A clean match goes out as "ready to approve" with a single tap. A flagged bill

goes out with the exact problem spelled out: “Line 3, item BOLT-12: billed 100 units, dock received 84.” Both carry Approve, Query, and Reject. Every decision writes a row to DynamoDB so the trail is auditable. A daily sweep re-surfaces any bill sitting unapproved past its due date so nothing quietly ages out. A monthly summary writes a short narrative: bills matched clean, value caught in variances, top suppliers by mismatch.

In plain words

A bill arrives from your packaging supplier for $4,200: 1,000 boxes at $4.20 each. The matcher finds purchase order PO-1182, which ordered 1,000 boxes at $3.80 each, and the goods-received note that says 1,000 boxes actually arrived. The quantity is fine. The price is not — the PO agreed $3.80 and the bill charges $4.20, a $400 overcharge across the line. The matcher marks the bill *price variance* and emails the buyer, Dan: “Packaging Co bill #5567 — price variance on boxes: billed $4.20/unit, PO agreed $3.80/unit, +$400 on the line. *[bill PDF] [PO-1182]*” with Approve, Query, and Reject. Dan taps *Query*; the matcher sends the supplier a templated note asking them to confirm the agreed price or reissue. The supplier reissues at $3.80. The corrected bill comes back in, matches clean, and Dan approves it with one tap. It joins the next payment run. The overcharge never got paid.

The cost of running this is about $3.20 a month at SMB volume. The cost of *not* running it is the steady drip of small overcharges nobody catches, the occasional duplicate paid twice, and the bill for goods that never fully arrived — each one small, all of them adding up.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every bill is checked three ways — against the purchase order and against what the dock actually received. Never one-sided.
- Four outcomes, always. Matched, price variance, quantity variance, no PO. There is no fifth.
- The match is plain Python against tolerances in a doc. No model decides whether a bill is right.
- It never pays a bill on its own. A human approves every payment, and the bill carries the exact gap.
- Tolerances and approvers live in a doc. Changing a price tolerance or a sign-off doesn't need a deploy.
- Every decision is logged. Audit a payment next year and you can see who approved what, and why.

Why this shape

Most small teams check supplier bills in one of three ways: somebody eyeballs the total and pays it, somebody compares the bill to the purchase order but not to what arrived, or nobody checks at all and the bookkeeper pays whatever lands. Eyeballing the total misses a wrong unit price that still produces a plausible-looking number. Checking only the PO misses the 16 boxes that never showed up. And paying whatever lands is how a supplier's "oops, our system double-billed you" becomes your money, permanently.

The setup above keeps the two documents you already produce — the purchase order and the goods-received note — as the source of truth, and adds a small system that *reads* every incoming bill and lines it up against both. Clean bills clear in one tap, so the matcher saves time on the 90% that are fine. The few that are off get caught with the exact line and the exact gap, while there's still time to query the supplier. And nothing is ever paid automatically — the matcher proposes, a human decides.

The next four posts walk through each piece in turn: how a supplier bill gets read, how a bill gets matched three ways, how a mismatch reaches the right person, and how a bill gets approved for payment. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 19, 2026 PART 2 OF 7 · SUPPLIER BILL MATCHER SERIES ~4 MIN READ

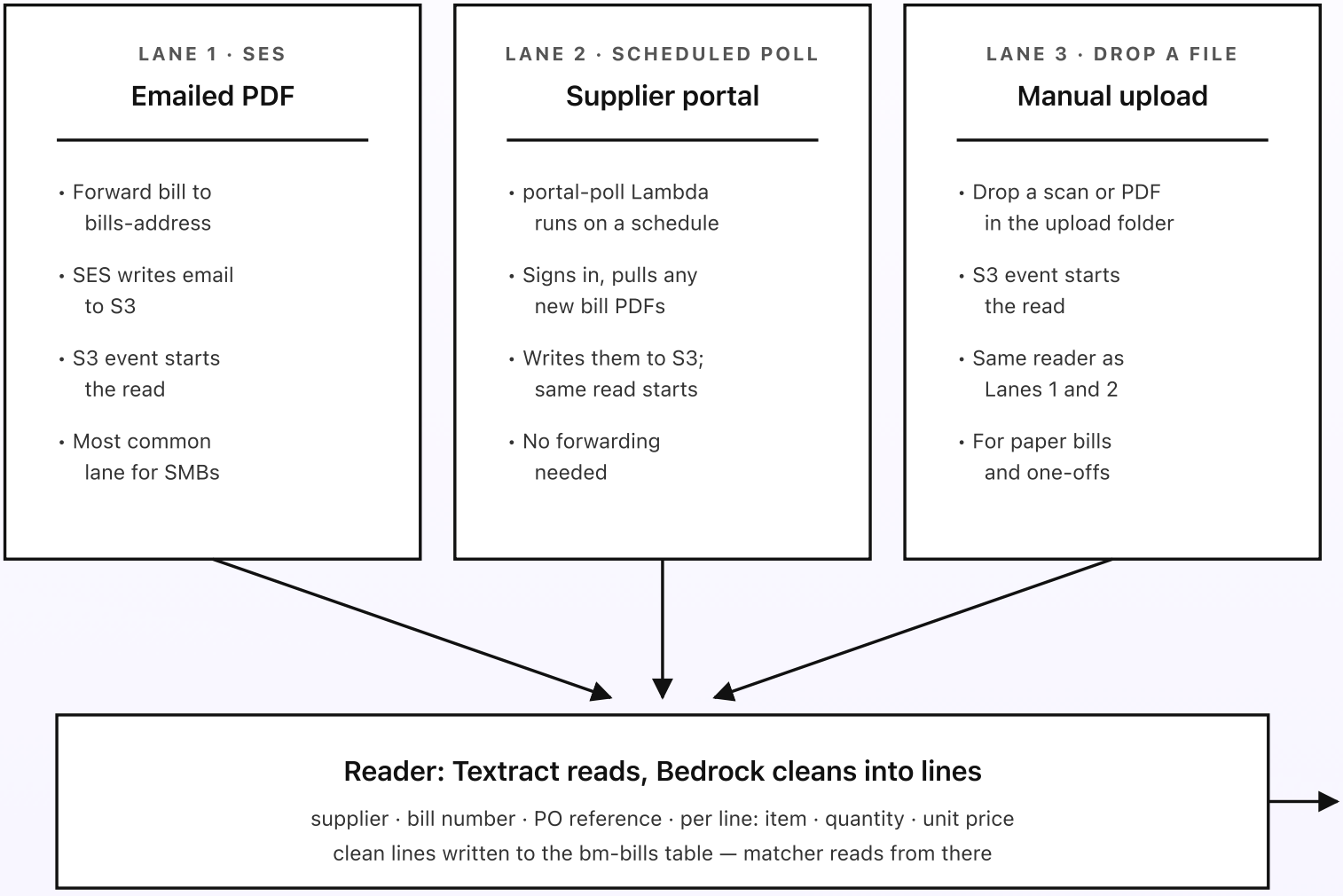
How a supplier bill gets read

The matcher can only check what it can read. So the first job is turning a supplier bill — which arrives as a PDF, a portal entry, or a scan somebody dropped in a folder — into clean, structured lines: this item, this many units, this unit price. There are three ways a bill gets in, and they all end the same way: Textract reads the page, and one short Bedrock call tidies the read into lines the matcher can compare. No bill skips the read step, because a bill the matcher can’t read is a bill that can’t be checked.

KEY TAKEAWAYS

- Three intake lanes feed one reader: an emailed-PDF lane, a supplier-portal poll, and a manual upload.
- Every bill is read by Textract; Bedrock Haiku 4.5 turns the read text into clean structured lines.
- The reader extracts the supplier, bill number, PO reference, and each line's item, quantity, and price.
- Low-confidence reads are held for a human to confirm before the bill goes to the matcher.
- The read step is where AI earns its place — the match itself, in Part 3, uses no model.

Three lanes into one reader



Every bill is read the same way — and a low-confidence read is held for a human to confirm first.

Fig 2. Three lanes converge on one reader. However a bill arrives, it is read by Textract and cleaned into structured lines by a single Bedrock call. The clean lines land in the bm-bills table, which the matcher reads in the next post.

Lane 1: emailed PDF (the lane most teams actually use)

Set up a dedicated inbound address — something like `bills@your-company.com` — via Amazon SES. Suppliers email their bills there directly, or anyone on the team forwards a bill they received. SES writes the raw email to `s3://bm-raw-mime/`. The S3 PUT triggers a reader Lambda that walks the email to the PDF attachment and starts the read.

This is the lane most small businesses live in. Suppliers already email bills; pointing those emails at one address is the whole setup. If a bill arrives as a link instead of an attachment, the reader follows the link, downloads the PDF, and proceeds the same way.

Lane 2: supplier portal

Some suppliers don't email bills at all — they post them to a portal and expect you to log in and fetch them. A small `portal-poll` Lambda runs on a schedule (a few times a day), signs in to each configured portal using credentials stored in Secrets Manager, and downloads any new bill PDFs to `s3://bm-uploads/`. From there the same S3 event starts the same read. The team never has to remember to log in and check; the poll does it.

Portals change their sign-in flow now and then, so this lane is the most maintenance-prone of the three. It's worth it only for suppliers that won't email — for everyone else, Lane 1 is simpler and more reliable.

Lane 3: manual upload

Paper still happens. A bill arrives in the post, or as a photo in a text message, or as a one-off a supplier handed over at delivery. For those, someone scans or saves the file and drops it straight into an upload folder that maps to `s3://bm-uploads/`. The S3 event starts the read exactly as the other lanes do. This lane is the catch-all: anything that didn't come by email or portal still gets read and checked the same way.

The read step: Textract, then one Bedrock call

Whatever lane a bill came in through, the reader does the same two things. First, Amazon Textract reads the PDF — it handles PDF, PNG, JPEG, and TIFF, including scans and photos, and returns the text plus any tables it found on the page. Bills are table-heavy (a list of line items with quantities and prices), so Textract's table reading does most of the heavy lifting here.

Second, one Bedrock Haiku 4.5 call turns that raw read into clean, structured lines. Suppliers lay bills out a hundred different ways — the quantity column might be labelled "Qty" or "Units" or nothing at all; the unit price might be before tax or after; a single line might wrap across two rows. The model's job is narrow and well-defined: take the messy table and emit a tidy list — supplier, bill number, PO reference if present, and per line the item code, item name, quantity, and unit price. The prompt is short: "Return JSON only. Copy numbers exactly as printed. Mark each field with a confidence score. Do not invent a value that isn't on the page."

This is the one place in the whole system where a model earns its place. Reading wildly varied real-world bill layouts is exactly what a model is good at, and exactly

what a pile of brittle parsing rules is bad at. But the model only *reads* — it never decides whether the bill is correct. That decision, covered in Part 3, is plain Python.

Low-confidence reads are held, not guessed

Each field the reader emits carries a confidence score. If a key field — the bill total, a line quantity, a unit price — comes back low-confidence (a smudged scan, an unusual layout), the bill is held in a *needs-review* state and a person is asked to confirm the read before it goes to the matcher. The reason is the same one that runs through this whole system: a bill the matcher checked against a misread quantity is worse than a bill that waited two minutes for a human to glance at it. The misread one passes a check it should have failed.

Next post: how the matcher takes those clean lines, lines them up against the purchase order and the goods-received note, and picks one of four outcomes — with no model anywhere in the decision.

PART 3 OF 7

JUNE 19, 2026 PART 3 OF 7 · SUPPLIER BILL MATCHER SERIES ~5 MIN READ

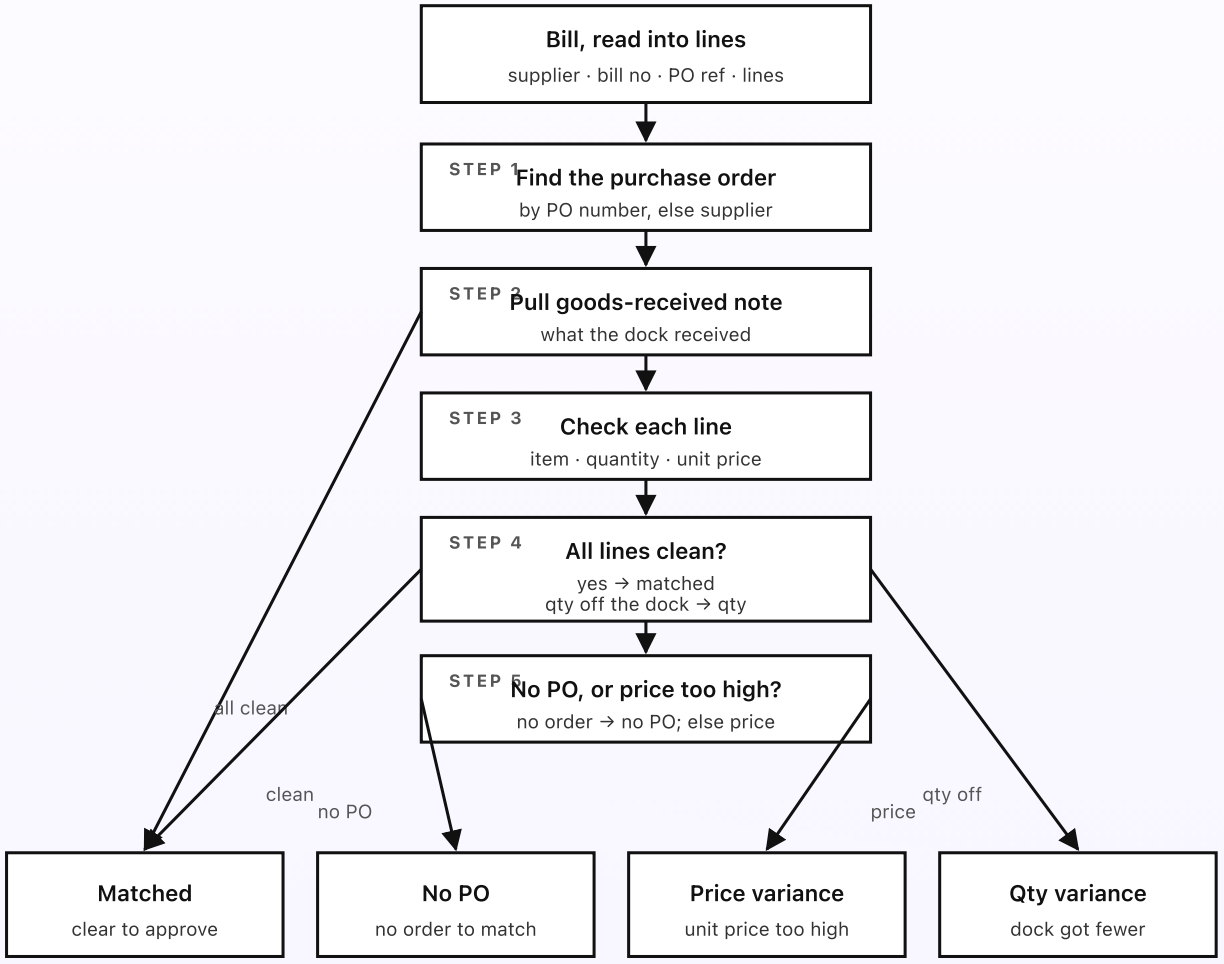
How a bill gets matched three ways

A bill has been read into clean lines. Now the matcher Lambda has to decide whether it's right. It pulls the purchase order the bill refers to and the goods-received note for that order, walks the bill one line at a time, and checks three things on each: right item, right quantity received, right price — each within the tolerance from the rules doc. The whole decision is plain Python. No model. No vector lookup. Every tolerance lives in the rules doc, where a buyer can edit it without a deploy.

KEY TAKEAWAYS

- The match is three-way: the bill against the purchase order and against the goods-received note.
- Each line is checked for item, quantity received, and unit price — each against a tolerance.
- Tolerances live in the rules doc — e.g. price within 2% or $5; quantity must match the dock.
- Four outcomes per bill: matched, price variance, quantity variance, no PO.
- The matcher never calls a model. The decision is entirely deterministic.

The decision flow, per bill



The rules doc holds every tolerance — change one and the next bill uses the new value.

Fig 3. The matcher's decision tree, per bill. Five steps decide which of four outcomes applies. The rules doc holds every tolerance; the matcher only enforces them.

Three-way, not two-way: why the dock matters

A two-way match compares the bill to the purchase order: did they bill what we ordered, at the price we agreed? That catches a wrong price. It does not catch a wrong delivery. If you ordered 100 boxes and the supplier billed 100 but only 84 turned up at the dock, a two-way match pays for 100. The 16 missing boxes are your money, gone.

The third leg is the goods-received note — the dock's record of what actually arrived. The matcher checks the bill against *both* the order and the receipt. Quantity is judged against what the dock received, not against what was ordered, because the thing you should pay for is the thing you actually got. Price is judged against the PO, because that's the number you agreed. Item is checked against both. Three documents, one decision.

Tolerances: a little drift is fine, a lot isn't

Real bills don't match to the penny. A supplier rounds, a freight surcharge nudges a line, an exchange rate moves. If the matcher flagged every one-cent difference, every bill would be a variance and the whole thing would be noise. So the rules doc sets a tolerance per check, in plain prose: "Price: allow the larger of 2% or $5 per line before flagging. Quantity: must match the goods-received note exactly,

except where the PO is marked part-delivery, in which case allow the agreed installment. Item: must be the exact item code on the PO."

Tolerances are per-supplier-overridable. A supplier you trust and whose prices float with a published index can get a looser price tolerance; a new supplier can get a tighter one. The override is a column in the rules doc, so a buyer changes it without anyone touching code.

Four outcomes, always

Every bill, every time, lands in exactly one of four outcomes. The names are simple on purpose.

- **Matched.** Every line agrees with the PO and the goods-received note inside tolerance. The bill is cleared for a manager's one-tap approval. Most bills, most days, are matched.
- **Price variance.** The right items arrived in the right quantity, but a unit price on the bill is higher than the PO agreed, beyond tolerance. The matcher records the exact line, the billed price, the PO price, and the dollar gap across the line.
- **Quantity variance.** A billed quantity doesn't match what the dock received. Usually that means the dock got fewer than billed (you're being charged for goods that didn't fully arrive), but it can also mean a duplicate or a split delivery the bill didn't account for. The matcher records the billed quantity and the received quantity side by side.
- **No PO.** The bill has no purchase order to match against — no PO number on it, and no open order that fits the supplier and items. This is the riskiest outcome,

because there's nothing to check the bill against. It always goes to a human; it never clears automatically.

State that makes the decision deterministic

The matcher reads the clean bill lines from the `bm-bills` table and the mirrored PO and goods-received data from S3, and writes the outcome to `bm-results`: `(bill_id, outcome, failing_lines, po_number, checked_at)`. With the bill, the PO, the receipt, and the tolerances all fixed, the outcome is fully determined. Re-running the match on the same inputs produces the same outcome — there's no randomness, no model, nothing that drifts. That matters when a finance manager asks "why was this flagged?" six weeks later: the answer is a line of arithmetic they can read, not a model's opinion.

Why the match uses no model

The matcher could ask a model "does this bill look right?" It doesn't. Two reasons. First, deciding whether to pay a supplier is exactly the kind of decision that must be predictable and explainable — if the PO says $3.80 and the bill says $4.20, that's a variance, full stop, and no amount of model judgment should soften it. Second, the match runs on every bill, and a model on that hot path would add cost and variance to the one part of the system that should have neither.

The model already did its job upstream, in Part 2, turning a messy PDF into clean lines. From there, comparing numbers against tolerances is plain Python and should stay that way. Next post: how a flagged bill reaches the right person, with the exact line and the exact gap, through four small guardrails.

PART 4 OF 7

JUNE 19, 2026 PART 4 OF 7 · SUPPLIER BILL MATCHER SERIES ~5 MIN READ

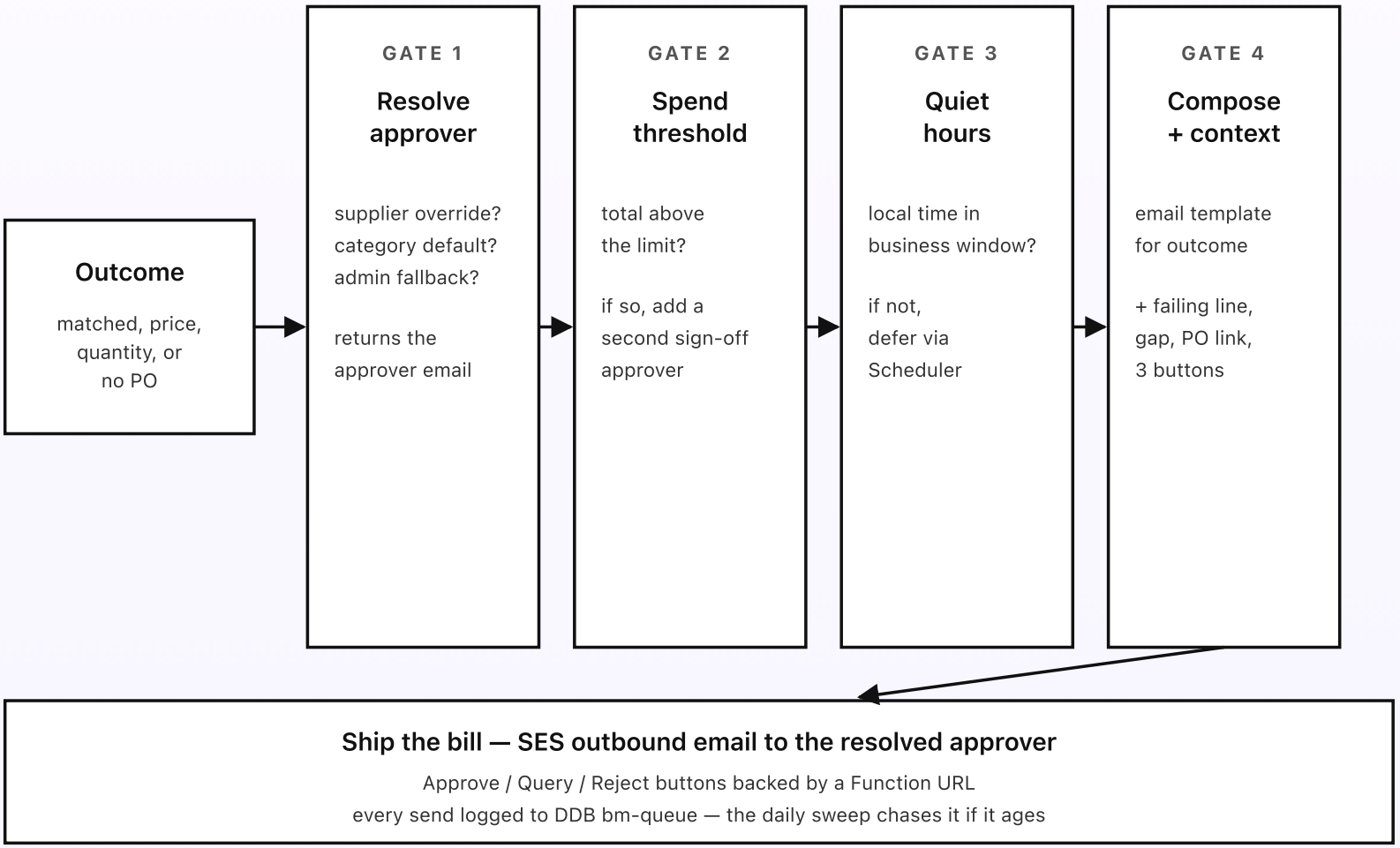
How a mismatch reaches the right person

The matcher picked an outcome — matched, price variance, quantity variance, or no PO. Now the approval-desk Lambda has to get the bill in front of the right person, at a sensible time, with enough detail that they can decide in seconds. Get any of that wrong and the message is worse than none: a bill sent to someone who left, a flag with no explanation, a sign-off request that skips the second approver a big payment needs. Four small guardrails sit between the outcome and the bill landing with a person.

KEY TAKEAWAYS

- Approver resolution: per-supplier override beats per-category default beats fallback to the finance admin.
- Big-spend bills require a second approver; the rules doc holds the threshold.
- Quiet hours defer non-urgent notifications to the next business hour.
- Every bill ships with the exact failing line, the billed-vs-agreed numbers, and Approve, Query, Reject.
- A no-PO bill always goes to a human and never clears on its own.

Four guardrails on every bill



Every gate is a deterministic check — no model calls, no surprise behavior on a Tuesday in April.

Fig 4. Four guardrails between the outcome and the bill landing with a person. Resolve the approver. Add a second sign-off for big spend. Honor quiet hours. Compose with the exact line and gap. Then ship via email and log it so nothing ages out unseen.

Gate 1: resolve the approver

Three places the approval desk looks for the right person, in order. First, the rules doc's per-supplier approver — if a supplier is assigned to a specific buyer, that buyer owns its bills regardless of category. Second, the per-category default ("all packaging bills go to the operations manager"). Third, the finance admin fallback — whoever set the matcher up and catches anything unrouted. The fallback should rarely fire in steady state; when it does, the monthly summary names every bill that hit it so the rules doc can be tidied.

Approvers are reached by email. The matcher deliberately uses email rather than a chat tool here, because a supplier-payment decision belongs in a place with a durable record and a clear from-address — and because the people who approve supplier bills (a finance manager, an owner) live in their inbox, not a chat channel.

Gate 2: a second sign-off for big spend

Not every bill should be one person's decision. A $200 stationery bill is fine for one approver; a $40,000 equipment bill is not. The rules doc sets a spend threshold (say $10,000). Above it, Gate 2 adds a second approver, and the bill isn't cleared for payment until *both* have approved. This is the one place the matcher quietly enforces a control that's easy for a busy team to skip: large

payments get two sets of eyes, automatically, without anyone having to remember the policy.

The second approver is named per category in the rules doc — usually the owner or the finance lead. If a bill needs two sign-offs and only one comes back, the bill stays in a waiting state and the daily sweep keeps it visible until the second arrives.

Gate 3: quiet hours

Most bills aren't urgent. A clean matched bill can wait until morning; a small price variance can too. Gate 3 reads the quiet-hours setting from the rules doc (default 6pm to 8am) and, if a non-urgent bill would land outside business hours, defers it with a one-off EventBridge Scheduler rule that re-sends at the next business minute. The exceptions are bills the rules mark urgent — a no-PO bill, or a variance above a large-dollar line — which go out immediately so a genuine problem isn't sitting unseen overnight.

Gate 4: compose with the exact gap, then ship

The template doc has one email layout per outcome. A matched bill gets a short "ready to approve" note with the supplier, total, and PO. A flagged bill gets the specifics: the exact line, the billed value, the agreed value, and the gap — "Line 3, item BOLT-12: billed 100 units at $0.40, dock received 84; you're being charged for 16 units that didn't arrive ($6.40)." The point is that the approver never has to open three documents and do the arithmetic themselves; the matcher already did it and shows its work.

Each email carries three buttons — Approve, Query, Reject — that hit the `ack-handler` Function URL when clicked. The bill PDF and the matching purchase order are linked inline so the approver can check the source in one click if they want to. Then the bill ships via SES outbound, and a row is written to `bm-queue` in DynamoDB marking it as waiting on this approver. The next post covers what each of those three buttons actually does.

Why the guardrails exist

None of these gates are exotic. They're the care a thoughtful finance person would take by hand — send it to whoever owns this supplier, get a second pair of eyes on the big ones, don't email at 11pm unless it's actually urgent, and spell out the problem so the approver can decide without digging. Putting them in code as four small sequential gates makes them part of the design, not something you're trusting a busy week to remember.

Next post: the three things an approver can do — approve, query, reject — and how each one updates the bill, the supplier, and the audit trail without ever paying a bill on its own.

PART 5 OF 7

JUNE 19, 2026 PART 5 OF 7 · SUPPLIER BILL MATCHER SERIES ~5 MIN READ

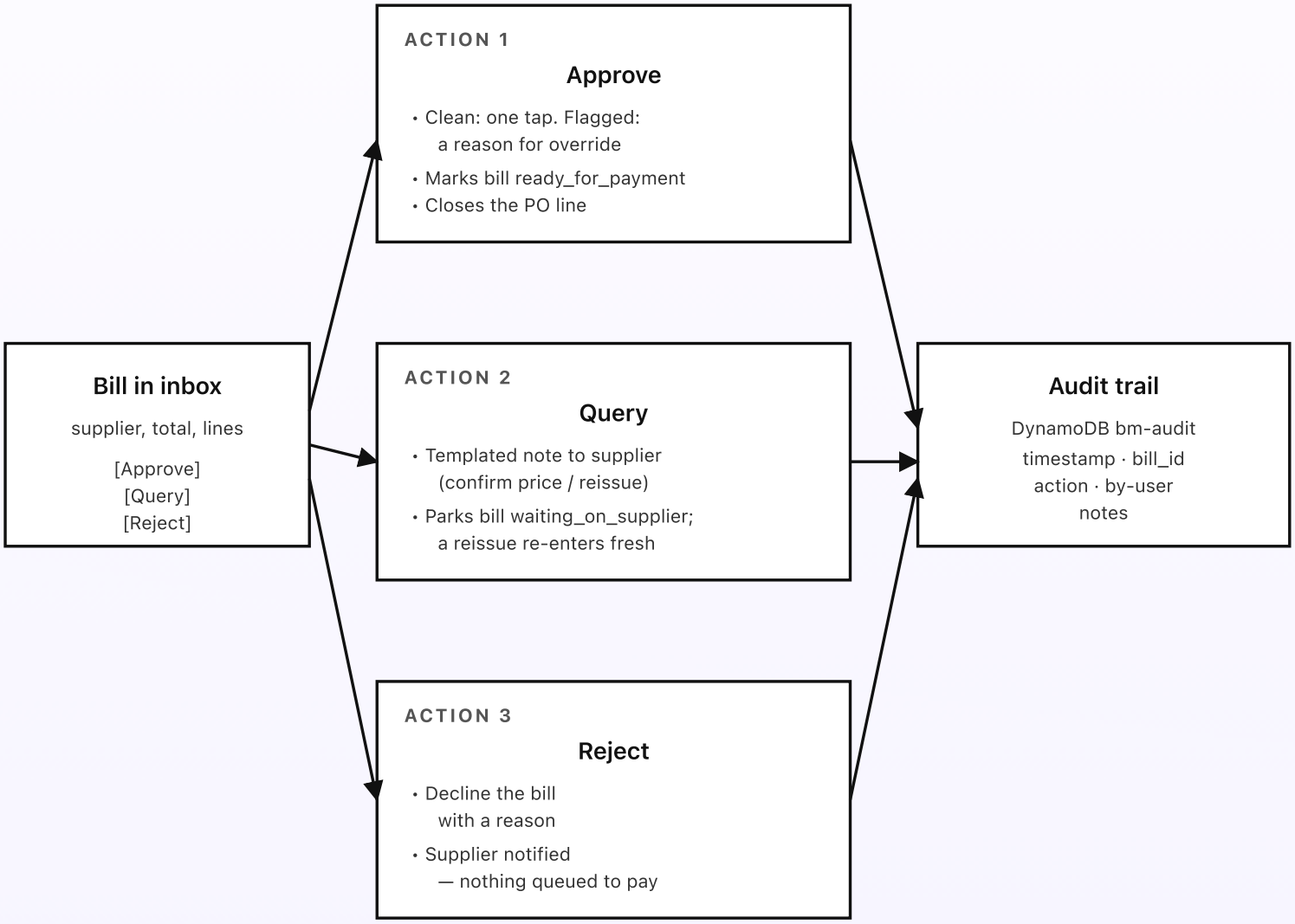
How a bill gets approved for payment

A bill lands in Dan’s inbox at 8:03am. Packaging Co, $4,200, flagged price variance. There are three buttons: Approve, Query, Reject. What happens when he taps one? The honest answer is “it depends which one.” This post walks through the three things an approver can do — approve, query, reject — and how the bill, the supplier, and the audit trail all stay in sync. The one thing none of them does is pay the bill: that’s always your normal payment run, after a human has said yes.

KEY TAKEAWAYS

- Three actions per bill: *approve* (clear for the payment run), *query* (ask the supplier), *reject* (decline).
- Approving a flagged bill requires a reason, captured in the audit trail.
- Query sends a templated note to the supplier; a reissued bill re-enters the matcher fresh.
- Approve only marks the bill ready — payment is still your normal run, never the matcher.
- The Approve button is an email action backed by a Function URL.

Three actions on a bill



Approve only marks a bill ready — payment is always your normal run, after a human says yes.

Fig 5. Three actions per bill, three different effects. Approve clears it for the payment run. Query asks the supplier and parks the bill. Reject declines it. Every action writes to the audit trail, and none of them moves money.

Action 1: approve (the most common)

Most bills are clean, so most of the time Approve is a single tap. On a matched bill, the approver glances at the supplier and total, taps *Approve*, and they're done. On a flagged bill, Approve is slightly heavier on purpose: a small form opens asking for a reason for overriding the flag — "agreed the higher price with the supplier by phone" or "short delivery, but we'll accept and they'll credit next month." The reason is required, because an override with no explanation is exactly the thing an auditor (or a future you) will want to understand later.

The button submits to the `ack-handler` Function URL. Three things happen, in order. First, the bill is marked `ready_for_payment` in the `bm-bills` table and the matched purchase-order line is closed so it can't be billed against twice. Second, if the bill needed two sign-offs (Part 4), the bill stays in a half-approved state until the second approver also taps Approve. Third, an `action: approved` row is written to `bm-audit` with the user, timestamp, outcome, and any override reason.

Marking a bill `ready_for_payment` does not pay it. The matcher hands a clean list of approved bills to your normal payment process — an export to your accounting tool, or a file for the bank run — and a person runs that process. The matcher never touches money. That boundary is the whole point: the system removes the tedious checking, not the human judgment about whether to actually pay.

Action 2: query (the “ask the supplier”)

When a bill is flagged and the approver doesn’t want to either approve or reject it — they want the supplier to explain or fix it — they tap *Query*. A small form opens with a templated message tailored to the outcome: for a price variance, “Your bill charges $4.20/unit; our PO agreed $3.80/unit. Please confirm the price or reissue.” For a quantity variance, “Your bill is for 100 units; our dock received 84. Please reissue for the received quantity or advise on the balance.”

On send, the `ack-handler` emails the supplier via SES outbound and parks the bill in a `waiting_on_supplier` state. The bill stops chasing the approver — the ball is in the supplier’s court. If the supplier reissues, the corrected bill comes back in through the normal intake lanes, gets read fresh, and is matched again from scratch. A corrected bill that now matches clean clears for a one-tap approval, and the loop closes. The original queried bill is linked to the reissue in the audit trail so the history is one connected story, not two unrelated bills.

Action 3: reject (the decline)

Sometimes the right answer is no. A duplicate of a bill already paid. A bill for an order that was cancelled. A no-PO bill for goods nobody ordered. The approver taps *Reject*, gives a reason, and the `ack-handler` marks the bill `rejected`, notifies the supplier with the reason, and closes it out. Nothing is queued for payment, and the rejected bill stays in the record so the same bill arriving again can be spotted as a re-send rather than treated as new.

Reject is deliberately final but not destructive. The bill isn’t deleted — it’s kept, marked rejected, with the reason and the person attached. If a reject was a

mistake (the duplicate turned out to be a legitimate second order), an admin can reopen it through a small command that reads the audit snapshot and puts the bill back in the queue. The reopen is itself an audit row.

Every action is logged, every action is reversible

The `bm-audit` table records every approve, query, and reject with the user who acted, the timestamp, the outcome the matcher had assigned, any override reason, and a snapshot of the bill's state before and after. If a bill was approved in error — wrong supplier, wrong amount — an admin can run an "undo last action" that restores the previous state from the snapshot, as long as the bill hasn't already gone out in a payment run. The undo is itself an audit row, so the trail of edits stays clean.

This kind of reversibility matters most for the decisions made quickly on a busy morning. Six months later, when an auditor or the owner asks "why did we pay this one over the PO price?", the answer is right there: who approved it, when, and the reason they typed. The audit trail is the system's memory.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the match itself is essentially free.

PART 6 OF 7

JUNE 19, 2026 PART 6 OF 7 · SUPPLIER BILL MATCHER SERIES ~3 MIN READ

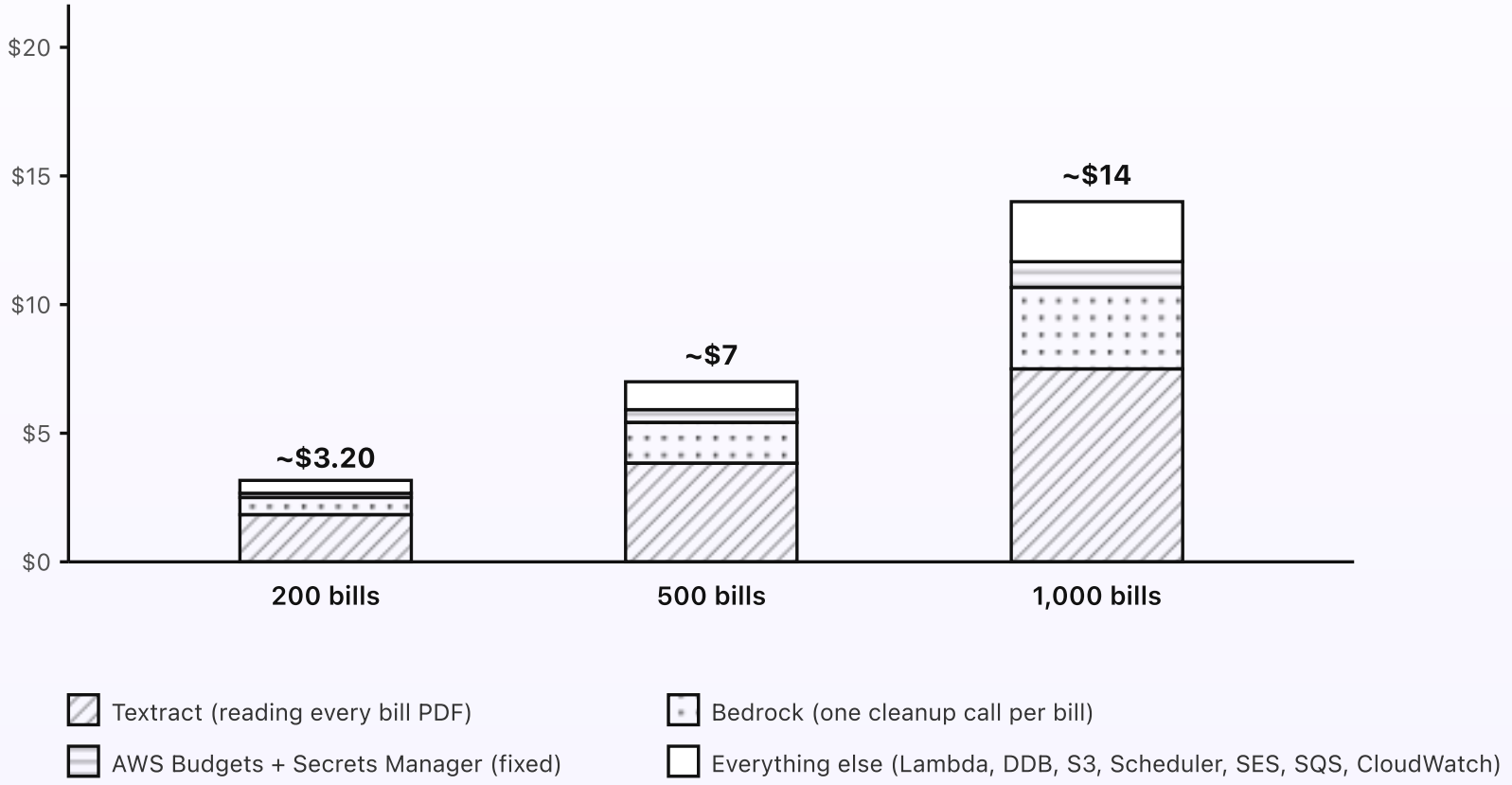
What the bill matcher costs

The matcher is one of the cheaper systems in this series, and the part that does the actual work — the three-way match — is essentially free, because it's plain Python doing arithmetic. The cost is almost entirely in reading the bill: Textract reads each PDF, and one small Bedrock call per bill turns that read into clean lines. At typical SMB volume the bill is a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $3.20/month at typical SMB volume (around 200 bills a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The three-way match costs almost nothing — it's plain Python.
- The cost is Textract reading each bill and one Bedrock call per bill to clean the lines.
- At 500 bills the bill is around $7. At 1,000 bills it's around $14.

Cost at three volumes



Reading the bill is the dominant cost — the three-way match itself is essentially free.

Fig 6. Monthly cost at three bill volumes. Textract and Bedrock dominate because each bill is read and cleaned once. The match itself adds almost nothing: it's plain Python comparing numbers against tolerances.

Where the dollars actually go

Textract (the bulk). Every bill is read once. Textract is priced per page, and a typical supplier bill is one to three pages. At 200 bills a month, that's a few hundred pages, which lands around a couple of dollars. Textract is the single biggest line because reading is the unavoidable first step — you can't check a bill you can't read. It grows roughly linearly with bill count.

Bedrock (one call per bill). After Textract reads the page, one Bedrock Haiku 4.5 call turns the raw text into clean lines: a few thousand input tokens (the Textract output) and a few hundred output tokens (the structured JSON), so a fraction of a cent per bill. Across 200 bills it's well under a dollar. The monthly summary adds one larger call — a board narrative of what matched, what was caught, and the top suppliers by mismatch — for a couple of cents.

Lambda runtime. The reader, the matcher, the approval desk, the `ack-handler`, the portal poll, the daily sweep, and the drive-sync all run as small short functions. The matcher itself is the cheapest of all — it reads a few rows, does arithmetic, writes an outcome. The Lambda total lands well under a dollar at all three volumes.

DynamoDB on-demand. A handful of small tables: `bm-bills`, `bm-results`, `bm-queue`, `bm-audit`. One write per bill read, one per match, one per approval action, plus the audit rows. Pennies a month at any of these volumes.

S3 + storage. The mirrored PO and goods-received CSVs, the raw inbound email, and the bill PDFs. A few hundred MB at SMB volume. Effectively free.

SES and SQS. Inbound for the emailed-bill lane and outbound for approver emails and supplier queries: $0.10 per thousand messages each way, so cents a month. SQS buffers the read-and-match work with a dead-letter queue for anything that fails twice; negligible at this scale.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the approve/query/reject endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Everything is event- or schedule-driven.
- **A Knowledge Base.** POs and goods-received notes are structured rows — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the match.** The three-way match is plain Python. Bedrock fires only to clean up the read and write the monthly summary.

How the cost scales

Textract and Bedrock grow linearly with bill count, because each bill is read and cleaned once. Lambda and DynamoDB grow linearly too, but from a much smaller base. So the bill at 2,500 bills a month is around $35; at 5,000 it's around $70. Past those volumes you'd look at Textract's bulk pricing and at caching repeat-layout reads from the same supplier — but those are optimizations for high-

volume accounts payable, not redesigns. The match itself never becomes the cost.

Set an AWS Budgets alarm at $25/month so anything unusual pages you before the bill matters. The matcher's normal-volume bill stays well under that ceiling — and a single caught overcharge usually pays for a year of running it.

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

JUNE 19, 2026 PART 7 OF 7 · SUPPLIER BILL MATCHER SERIES ~8 MIN READ

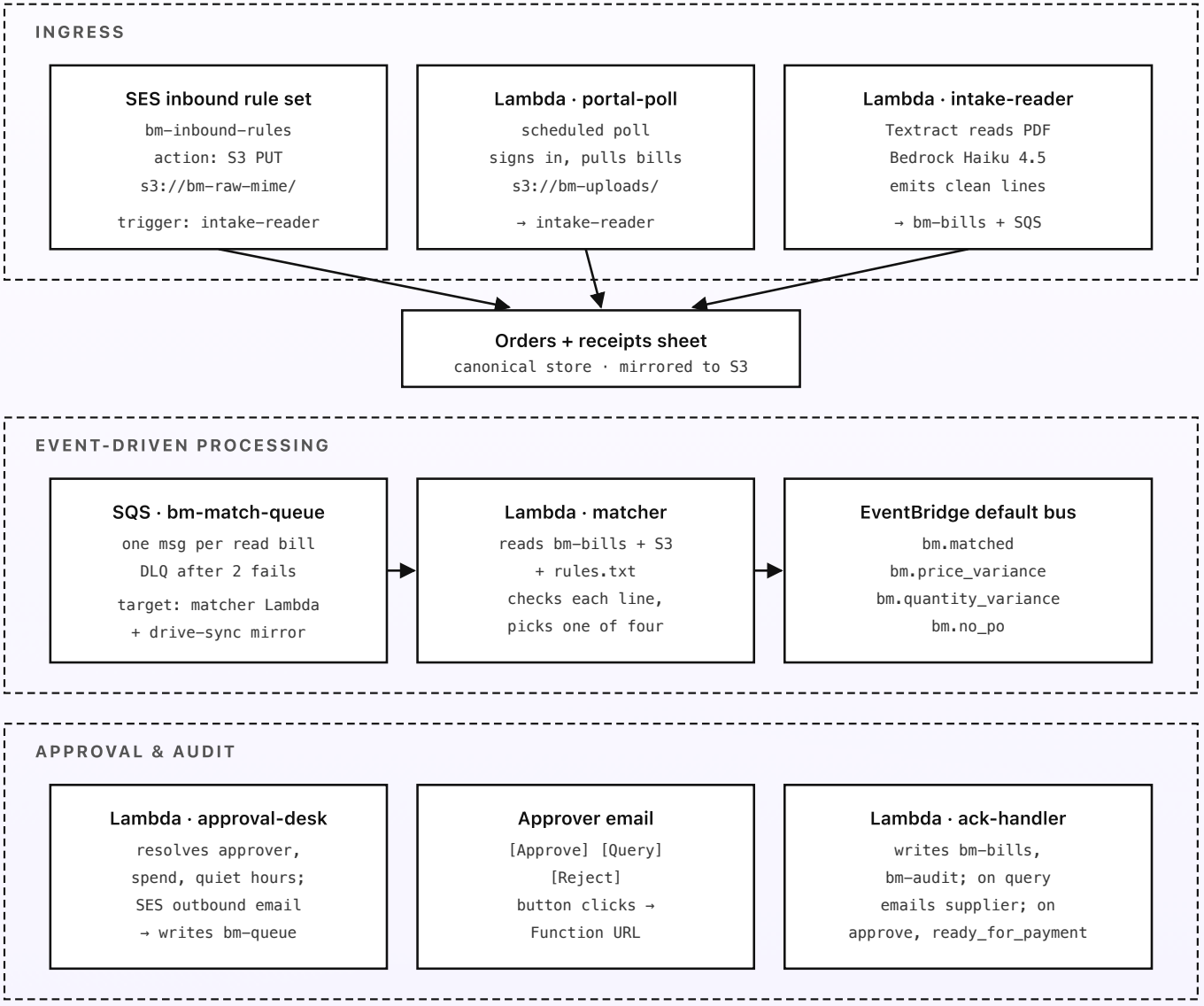
Engineering reference: the bill matcher architecture

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 email approval 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 cross-Region inference, and EventBridge Scheduler 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 bill sitting unmatched for an hour, not a regional outage, and the matcher never moves money so there's no in-flight payment to protect. One AWS account dedicated to the matcher (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



The matcher only proposes — every payment is approved by a human, logged to bm-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the reader), event-driven processing (the matcher consuming a queue and emitting one of four outcome events), approval and audit (the bill ships to an approver and their decision is recorded). 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-reader`** — S3 PUT trigger on `s3://bm-raw-mime/` and `s3://bm-uploads/`. Parses MIME (for the email lane), extracts the bill PDF, runs Textract via `StartDocumentAnalysis` with the `TABLES` feature (asynchronously, to handle multi-page bills). On Textract completion (via SNS notification), reads the structured tables and calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to emit clean lines (supplier, bill number, PO reference, per-line item/quantity/unit price) with a confidence score per field. Writes the bill to `bm-bills` and enqueues a match message to `bm-match-queue`; low-confidence reads are written in a `needs_review` state and skip the queue until a human confirms. For DOCX or XLSX bills (rare), falls back to `python-docx` / `openpyxl`. Memory: 512 MB. Timeout: 60 s.
- **`portal-poll`** — EventBridge Scheduler target, three times a day. Signs in to each configured supplier portal (credentials in Secrets Manager under `bm/portals/*`), downloads any new bill PDFs, and writes them to `s3://bm-`

`uploads/` where `intake-reader` picks them up. Portal sign-in flows are brittle; each portal is its own small adapter, and a failure on one portal is isolated from the others. Memory: 512 MB. Timeout: 120 s.

- **`matcher`** — SQS trigger on `bm-match-queue`. For each bill, reads the clean lines from `bm-bills`, pulls the matching purchase order and goods-received lines from `s3://bm-orders-source/` (mirrored from the Drive sheet), and the tolerances from `s3://bm-rules-source/rules.txt`. Resolves the PO by PO number, falling back to a supplier-plus-item match. Checks each line for item, received-quantity, and unit-price against tolerance; picks one of four outcomes; writes `bm-results` and emits `bm.matched`, `bm.price_variance`, `bm.quantity_variance`, or `bm.no_po` with the bill and failing-line context. Memory: 512 MB. Timeout: 30 s. *No Bedrock calls.*
- **`approval-desk`** — EventBridge rule on the four outcome events. Resolves the approver (per-supplier → per-category → admin fallback), checks the spend threshold (adds a second approver above it), checks quiet hours, formats the email from the template for the outcome, and ships via SES `SendRawEmail` with Approve/Query/Reject links to the `ack-handler` Function URL. On a quiet-hours defer, creates a one-off EventBridge Scheduler rule that re-invokes `approval-desk` at the next business minute. Writes a row to `bm-queue`. Memory: 256 MB. Timeout: 30 s.
- **`ack-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a signed, single-use token embedded in the button link (HMAC keyed from a secret) so a leaked email link can't be replayed. Triggered by Approve/Query/Reject clicks. On approve: marks the bill `ready_for_payment` in `bm-bills`, closes the matched PO line, and (if dual sign-off) waits for the second approver. On query: emails the supplier via SES and sets

`waiting_on_supplier`. On reject: marks `rejected` and notifies the supplier. Always writes `bm-audit`. Memory: 256 MB. Timeout: 15 s.

- **`drive-sync`** — EventBridge Scheduler target, every 15 minutes. Uses the Google Drive + Sheets APIs (service-account credentials in Secrets Manager under `bm/drive/sa`) to export the orders-and-receipts sheet as CSV to `s3://bm-orders-source/` and the rules and templates docs to `s3://bm-rules-source/`, only if changed since the last sync. Memory: 256 MB. Timeout: 30 s.
- **`sweep`** — EventBridge Scheduler target, daily at 9am local. Reads `bm-queue` for bills still unapproved past their due date (or waiting on a second sign-off) and re-surfaces them to the approver; also re-pings `waiting_on_supplier` bills that have gone quiet. No Bedrock; a plain summary. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `bm-results` and `bm-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph narrative (bills matched clean, value caught in variances, top suppliers by mismatch); emails it via SES to the stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB · `bm-bills`** — one row per bill. PK `bill_id`; attributes: `supplier`, `bill_number`, `po_ref`, `lines` (item/qty/unit_price), `state` (needs_review/queued/flagged/ready_for_payment/waiting_on_supplier/rejected), `total`. On-demand.
- **DynamoDB · `bm-results`** — one row per match. PK `bill_id`; attributes: `outcome` (matched/price_variance/quantity_variance/no_po), `failing_lines`,

`po_number`, `checked_at`. On-demand.

- **DynamoDB · `bm-queue`** — one row per pending approval. PK `bill_id`; sort key `approver`; attributes: `sent_at`, `due_date`, `second_approver` (if dual sign-off), `status`. On-demand.
- **DynamoDB · `bm-audit`** — one row per write action of any kind. PK `(bill_id, ts)`; attributes: `action` (approved/queried/rejected/override), `by_user`, `reason`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **S3 · `bm-orders-source`** — mirrored CSV of the purchase-order and goods-received tabs. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3 · `bm-rules-source`** — mirrored rules and email templates as plain text. Versioning enabled.
- **S3 · `bm-raw-mime`** — raw inbound MIME from emailed bills. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3 · `bm-uploads`** — bill PDFs from the portal poll and manual upload, plus the parsed source bills kept for reference by the registry row.

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: `intake-reader` for turning the Textract read into clean lines, and `summary` for the monthly narrative. The `matcher` never calls Bedrock; the three-way match is deterministic Python.

- **Heavier model.** `anthropic.claude-sonnet-4-6-...` is configured but off by default. It's only worth switching the reader to Sonnet for suppliers with pathological multi-page layouts where Haiku's line extraction needs a confidence bump; the per-supplier config flag lets you route just those.
- **Embeddings.** Not used. POs and goods-received notes are structured rows; deterministic lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors.

EventBridge Scheduler config

- **`bm-drive-sync`** — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- **`bm-portal-poll`** — `cron(0 8,12,16 * * ? *)` in TZ. Target: `portal-poll` Lambda.
- **`bm-daily-sweep`** — `cron(0 9 * * ? *)` in TZ. Target: `sweep` Lambda.
- **`bm-monthly-summary`** — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `approval-desk` 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. `bills.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.

- SES inbound rule set `bm-inbound-rules`: one rule with recipient `bills@your-company.com` → spam scan → S3 PUT to `s3://bm-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-reader`.
- SES outbound for approver emails and supplier queries: verify a sender identity at `ap@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:

- **matcher role:** `s3:GetObject` on the orders and rules keys; `dynamodb:GetItem` on `bm-bills`; `dynamodb:PutItem` on `bm-results`; `events:PutEvents` on the default bus; `sqs:ReceiveMessage` + `DeleteMessage` on `bm-match-queue`. *No* `bedrock:*`.
- **intake-reader role:** `s3:GetObject` on `bm-raw-mime` and `bm-uploads`; `textract:StartDocumentAnalysis` + `GetDocumentAnalysis`; `bedrock:InvokeModel` on the Haiku ARN; `dynamodb:PutItem` on `bm-bills`; `sqs:SendMessage` on `bm-match-queue`.
- **approval-desk role:** `events:CreateSchedule` for the deferred one-offs; `ses:SendRawEmail` from the verified sender; `dynamodb:PutItem` on `bm-queue`; `secretsmanager:GetSecretValue` on the link-signing secret.
- **ack-handler role:** `dynamodb:PutItem` / `UpdateItem` on `bm-bills` and `bm-audit`; `ses:SendRawEmail` for supplier query/reject notices; `secretsmanager:GetSecretValue` on the link-signing secret; `dynamodb:Query` for dual-sign-off state.

- **drive-sync and portal-poll roles:** `secretsmanager:GetSecretValue` on the relevant Google or portal secret; `s3:PutObject` on the orders, rules, and uploads buckets; outbound network to `www.googleapis.com` and the configured portal hosts.

Email approval flow

The approval email is a small HTML email with three buttons rendered as links to the `ack-handler` Function URL. Each link carries a signed, single-use token: an HMAC over `(bill_id, action, approver, nonce, expiry)` keyed from `bm/links/secret` in Secrets Manager. `ack-handler` verifies the HMAC, checks the nonce hasn't been spent (a conditional write to a small `bm-nonce` attribute), and checks the token hasn't expired before acting. Approve on a flagged bill and Query both render a follow-up form (the override reason, or the supplier-query text) posted back to the same Function URL; Approve on a clean match and Reject are one click plus a confirm.

This keeps the whole approval surface to one public Function URL with no API Gateway, while staying safe against a forwarded or leaked email link — the token is single-use and time-boxed, so the worst case from a leaked link is a no-op on an already-decided bill.

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.

- **SQS DLQ:** `bm-match-queue` has a dead-letter queue; a message that fails the matcher twice lands there with an alarm, so a bad read never silently blocks a bill.
- **Alarms:** matcher failures > 0 in an hour; DLQ depth > 0; ack-handler token-verification failures > 5/hour (might mean the signing secret rotated); Textract throttles.
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `bm-cost-alarm` subscribed to the finance admin's email.

Config and secrets

Service-account credentials for Drive and Sheets live in Secrets Manager under `bm/drive/sa`. Supplier-portal credentials under `bm/portals/*`. The email-link signing secret under `bm/links/secret`. The configured timezone, quiet-hours window, spend threshold, tolerance defaults, and admin fallback approver live in Parameter Store under `/bm/config/` (with per-supplier overrides kept in the rules doc so a buyer can change them without a deploy). Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) and AWS SAM. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for `bm-orders-source` and `bm-rules-source` so a bad Drive edit rolls back in one click, and give the matcher an

SQS source with a DLQ so a single unreadable bill never wedges the pipeline. Total deployable surface: around eight Lambdas, four DDB tables, four S3 buckets, one SQS queue with a DLQ, one EventBridge rule on the default bus (plus the Scheduler rules), one SES rule set, and one Budgets alarm.

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