7-PART SERIES · FREE COMPANION

Stock count reconciler

A physical stock count almost never matches the system of record, and the gap is where money quietly leaks — shrinkage nobody logged, a sale that never rang through, a delivery counted twice at the dock. This is a small serverless system that takes a physical count, compares each counted line against the system quantity, works out the variance, and asks a model for the likely cause — then queues every material gap for a manager's sign-off before a single adjustment is posted. The arithmetic is plain code; the model only ever suggests a cause, it never moves stock. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

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

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/stock-count-reconciler

CONTENTS

Stock count reconciler

PART 1 OF 7

JUNE 27, 2026 PART 1 OF 7 · STOCK COUNT RECONCILER SERIES ~10 MIN READ

A stock count reconciler on AWS for a few dollars a month

Every business that holds stock runs a count now and then, and every count tells the same uncomfortable story: the shelf and the system disagree. There are 432 bolts on the rack and the system swears there are 480. A tin of paint the system has never heard of is sitting in aisle three. Forty-eight units gone and nobody can say whether it walked out the door, was sold without ringing through, or was never delivered in the first place. Reconciling that by hand — line by line, working out each gap and chasing each cause — is slow, dull, and the first thing that gets skipped when the count runs late. This post walks through the design of a small system that takes a physical count, compares it against the system of record, flags every variance with a likely cause, and queues the adjustments for sign-off. It never moves stock on its own.

KEY TAKEAWAYS

- Two ways to submit a count: a scanner export or a filled-in sheet. Both land as clean counted lines.
- Every counted line lands in one of three states: in agreement, a minor variance that's logged, or a material variance that's queued.
- Each material gap is labelled with one of four likely causes: shrinkage, miscount, unrecorded sale, or receiving error.
- The variance arithmetic is plain code. The model only suggests the cause — it never moves stock.
- Designed on AWS for about $2.20/month at typical small-business volume. No adjustment posts without a manager's sign-off.

The whole system on one page

Before any code, here's the shape of what we're designing. A stock count reconciler is not a reorder bot — it never buys anything. It takes a physical count, lines it up against the system of record, and tells you where the two disagree and why. The buying decision stays with you.

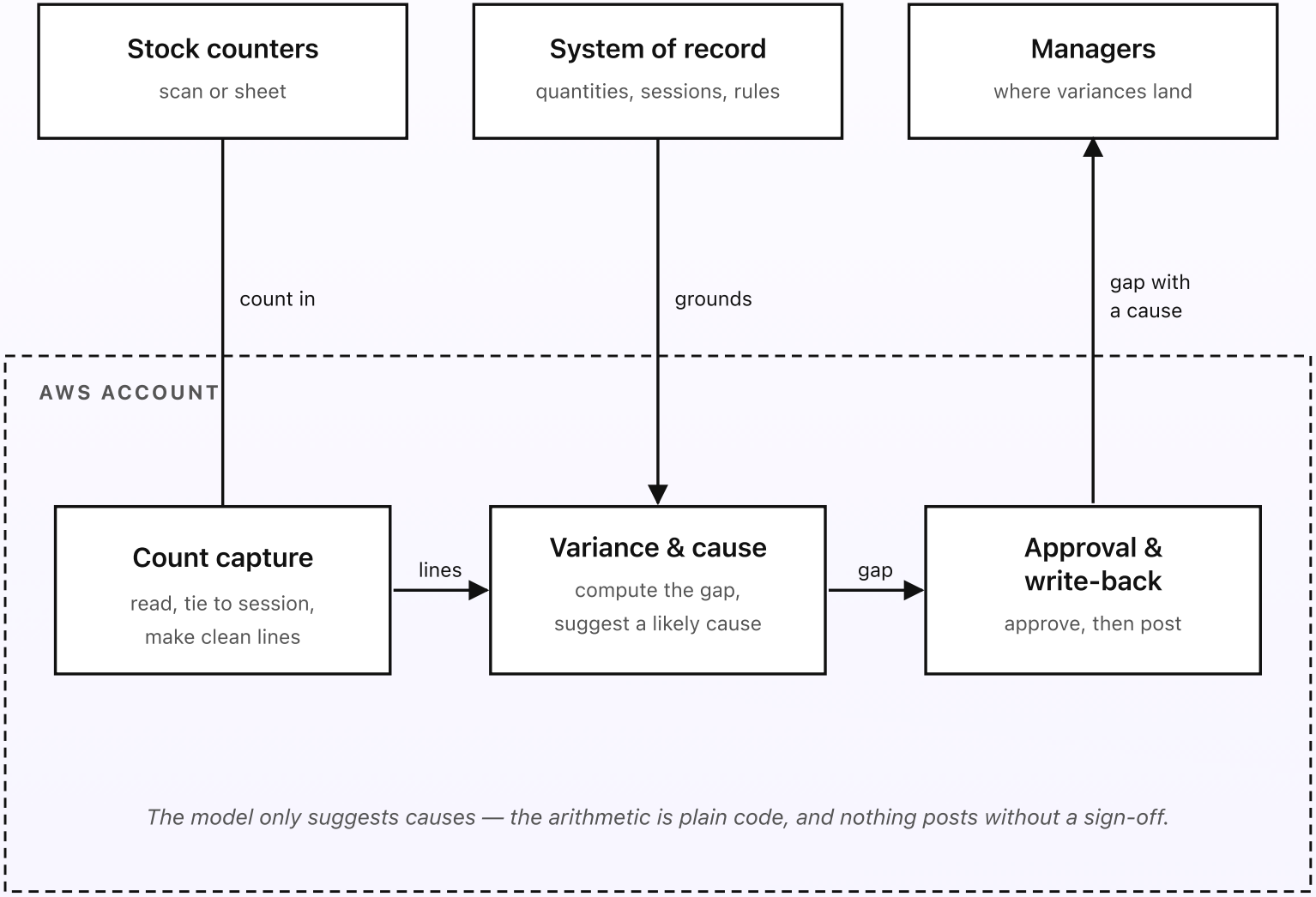

Stock counters
scan or sheet
System of record
quantities, sessions, rules
Managers
where variances land
count in
grounds
gap with
a cause
AWS ACCOUNT
Count capture
read, tie to session,
make clean lines
lines
Variance & cause
compute the gap,
suggest a likely cause
gap
Approval &
write-back
approve, then post
The model only suggests causes — the arithmetic is plain code, and nothing posts without a sign-off.

Fig 1. Three sources outside, three pieces inside AWS. A count flows in as a scan or a sheet. The engine computes every variance and labels the material ones with a likely cause. The approval piece queues each one for a manager, who approves or rejects before any adjustment posts.

What you set up once (the outside)

- **The system of record.** Wherever your current inventory quantities already live — a stock spreadsheet, the export from your point-of-sale or warehouse system. One row per stocked line: SKU, item name, location, the system quantity, and a unit cost. You already keep this; the reconciler just needs to read it. A small sync mirrors it into a DynamoDB table, `stkr-inventory`, so every comparison runs against a stable snapshot rather than a live system mid-edit.
- **A rules doc.** One short doc holding the *materiality tolerance* — how big a gap has to be before it's worth a manager's time. A common set: a variance is material if it exceeds 2 units *and* 1% of the system quantity, or if the value of the gap exceeds £20, whichever trips first. Anything smaller is logged but posts nothing. The doc also names the *approver* per location and the quiet hours for notifications.
- **Managers.** The people who sign off adjustments — usually a store or warehouse manager. Each has an email address. Material variances land with the SKU, the counted quantity, the system quantity, the gap in units and value, the suggested cause with its one-line rationale, and two buttons: *Approve* and *Reject*. A button click never moves stock by itself — *Approve* posts the adjustment and records who did it; *Reject* discards it and the count line stands.

What runs on every count (the inside)

- **The count capture.** A count starts a *session* — a row in `stkr-counts` recording the location, who's counting, and when. Staff submit either a scanner export or a filled-in sheet; both land in S3, which fires a count-submitted event. The capture Lambda reads the file, ties it to the open session, and normalises it: de-duplicates repeat scans, drops blank rows, maps each entry to a known SKU, and converts cases to eaches where needed. Out comes a clean list of counted lines — SKU, location, counted quantity — with nothing reconciled yet. Part 2 covers this.
- **The variance and cause engine.** Runs as soon as a session is captured. For each counted line it looks up the system quantity, subtracts, and computes the gap three ways: units, percent, and value. It applies the materiality tolerance to sort each line into *in agreement*, a *minor variance* (logged, no adjustment), or a *material variance* (queued). For each material gap it builds a small evidence bundle — the gap, recent sales, the last receiving, the last count date — and asks one Bedrock Haiku 4.5 call to pick a likely cause from a fixed list: *shrinkage*, *miscount*, *unrecorded sale*, or *receiving error*. The arithmetic is plain Python; the model only labels. Parts 3 and 4 cover this.
- **The approval and write-back.** Takes each material variance and queues it for the right manager. The email carries the full picture and the suggested cause, with *Approve* and *Reject*. On approve, the adjustment is posted to `stkr-inventory` atomically and a row is written to `stkr-adjustments` with a before-and-after snapshot, so the trail is auditable. A daily sweep re-surfaces any variance left unactioned, and a monthly summary writes a short narrative: lines counted, value caught in variances, the split of causes. Part 5 covers this.

In plain words

The Saturday team counts the fixings aisle and submits the scan. The capture turns it into clean lines and the engine reconciles them. Most agree. One does not: SKU BOLT-M6-50, counted 432, system says 480 — a gap of 48 units, 10%, worth £16.80 at £0.35 each. That clears the materiality tolerance, so it's queued. The engine pulls the evidence: no delivery of M6 bolts in the last month, steady sales, last counted eight weeks ago. One Bedrock call returns *likely cause: shrinkage* with the rationale "a sustained shortage with no recent receiving and normal sales points to loss rather than a recording error." The manager, Priya, gets an email: "BOLT-M6-50 — short 48 units (–10%, –£16.80). Likely shrinkage. *[Approve] [Reject]*." She recounts the rack, confirms it, and taps *Approve*. The system posts –48 against the SKU and logs the adjustment. The leak is now visible, dated, and explained — not discovered six months later when the numbers stop adding up.

Running this costs about $2.20 a month at SMB volume. The cost of *not* running it is shrinkage that hides inside "the count was a bit off," receiving errors nobody traces back to the supplier, and a system of record that drifts further from the shelf with every month that passes.

DESIGN RULES THAT SHAPED EVERY DECISION

- The arithmetic is plain Python against a tolerance in a doc. No model decides whether a gap is real or how big it is.
- Three states for every line: in agreement, a minor variance that's logged, or a material variance that's queued. There is no fourth.
- Four likely causes, always: shrinkage, miscount, unrecorded sale, receiving error. The model picks from that list and explains itself in a sentence.
- It never moves stock on its own. A manager approves every adjustment, and the email carries the exact gap.
- It reconciles; it does not reorder. Working out what you have is a different job from deciding what to buy.
- Every action is logged. Audit an adjustment next year and you can see who posted it, when, and why.

Why this shape

Most small teams handle a stock count one of three ways: they count, eyeball the obvious gaps, and overwrite the system with the new numbers; they count and never reconcile, so the sheet goes in a drawer; or they reconcile by hand on a spreadsheet that takes a day and gets abandoned halfway. Overwriting the system loses the variance entirely — you never learn that 48 bolts went missing, only that the number is now 432. Never reconciling means the system and the shelf drift

apart until nobody trusts either. And reconciling by hand is real work that competes with everything else on a manager's desk.

The setup above keeps the system of record as the source of truth, takes the count as a fact, and puts a small system between them that computes every gap, explains the material ones, and changes nothing until a person says so. The gaps inside tolerance get logged and ignored, so the manager only ever looks at the handful that matter. Each of those arrives with a likely cause already attached, so the decision is "recount or accept," not "start investigating." And nothing is ever posted automatically — the reconciler proposes, a human decides.

The next four posts walk through each piece in turn: how a count gets captured, how a variance gets found, how a likely cause gets suggested, and how an adjustment gets approved. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 27, 2026 PART 2 OF 7 · STOCK COUNT RECONCILER SERIES ~8 MIN READ

How a count gets captured

Before anything can be reconciled, the count has to arrive in a shape the system can read. Staff submit a physical count one of two ways: a scanner export, or a spreadsheet they fill in walking the aisles. Both are messy — duplicate scans, blank rows, an item code typed three different ways, a count entered in cases when the system holds eaches. This post is about the first step: how a submitted count lands, how it gets tied to a count session, and how it becomes a clean list of counted lines — SKU, location, counted quantity — with nothing reconciled yet. Getting this step boring and reliable is what makes every step after it trustworthy.

KEY TAKEAWAYS

- A count is a session, not a file. It opens, collects lines, and closes — one row in `stkr-counts`.
- Two submission lanes: a scanner export and a filled-in sheet. Both land in S3 and fire one count-submitted event.
- Capture normalises the mess: de-duplicates scans, drops blanks, maps codes to known SKUs, converts cases to eaches.
- The output is a clean list of counted lines — SKU, location, counted quantity — with nothing reconciled yet.
- Anything that won't map cleanly is parked, not guessed. A bad line never silently becomes a variance.

A count is a session, not a file

The unit of work here is a *count session*, not a spreadsheet. Before anyone scans a thing, the session is opened: a row in `stkr-counts` recording the location being counted, who's counting, the moment it opened, and a status of `open`. Every counted line that comes in is tied to that session. When the count is submitted, the session flips to `captured` and the reconciler takes over. Modelling it this way means a count is auditable from the first scan — you can always say which session a line belonged to, who ran it, and against which snapshot of the system it was reconciled.

It also makes partial counts honest. A team rarely counts the whole shop in one go; they do the fixings aisle on Saturday and the paint aisle on Sunday. Each is its own session against its own location, so a half-finished count never gets mistaken for a full one, and a SKU that simply wasn't in scope this weekend never shows up as a phantom variance.

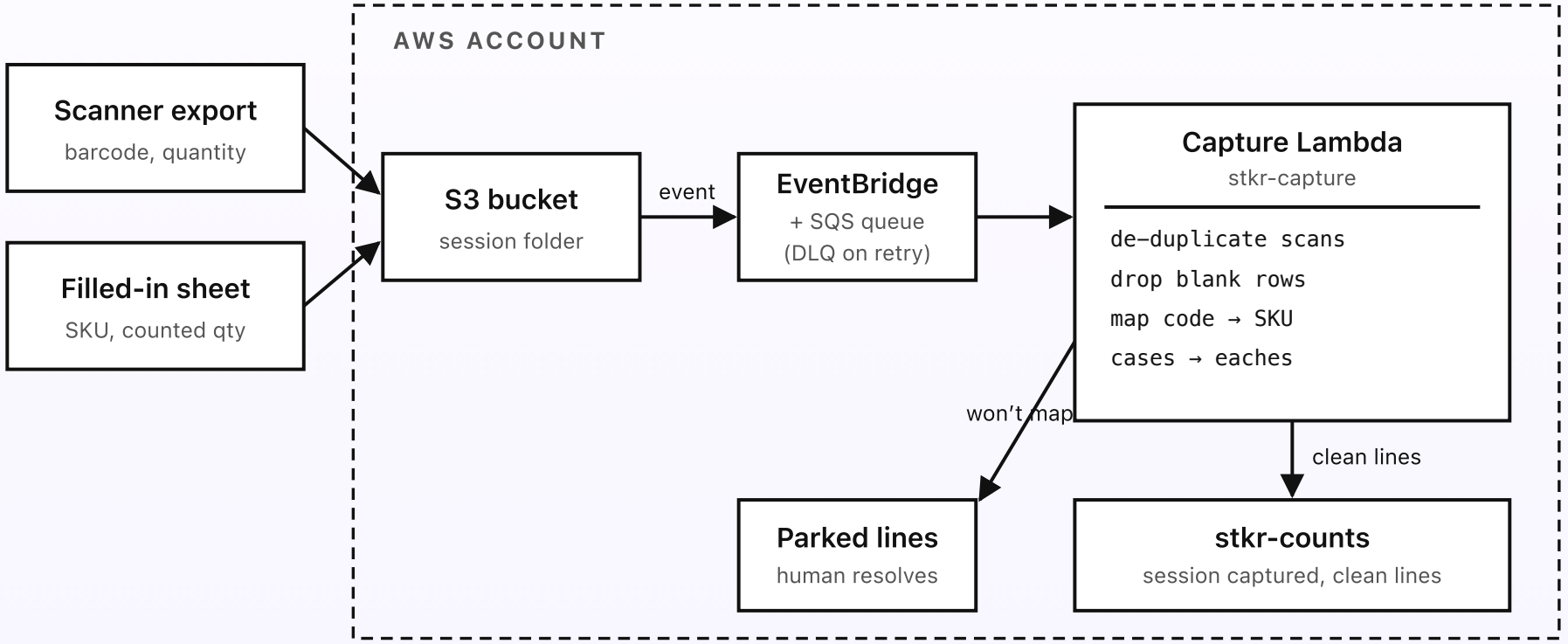
Two lanes in

Staff submit a count one of two ways, and both end in the same place.

The **scanner lane** is for shops with a handheld or a phone scanning app. The team walks the aisle, scans each item, keys the quantity, and the app exports a file — usually a CSV of barcode and quantity. That file is dropped into the session's folder in S3.

The **sheet lane** is for everyone else. The session opens with a pre-filled spreadsheet — one row per SKU expected at that location, a blank column for the counted quantity — and staff fill it in as they walk. When they're done they upload it to the same S3 folder. A scan of a paper tally counts as this lane too: it's saved as a sheet with the numbers typed in.

Either way, the upload to S3 is the trigger. S3 fires a count-submitted event onto EventBridge, which lands on an SQS queue in front of the capture Lambda. The queue matters: it means two teams submitting at once can't trample each other, a capture that fails gets retried, and anything that fails twice goes to a dead-letter queue for a human to look at rather than vanishing.



Capture normalises and ties to a session — it reconciles nothing.
A line it can't map is parked, never guessed.

Fig 2. A count flows from upload to clean lines. Either lane lands in S3, which fires one event onto a queue. The capture Lambda normalises the file, writes clean counted lines to the session, and parks anything it can't map cleanly rather than guessing.

Normalising the mess

Raw counts are never clean, and capture's whole job is to make them so before the engine ever sees them. Four steps, all plain code:

- **De-duplicate scans.** A handheld picks up the same barcode three times as someone re-scans a tricky label. Capture collapses repeat scans of the same SKU at the same location into a single counted quantity — summing where the app records one scan per unit, taking the keyed quantity where it records a count.
- **Drop blank rows.** A pre-filled sheet has a row for every expected SKU, and the team only fills in what they actually counted. A blank counted-quantity cell means "not counted," not "counted zero" — an important difference. Blanks are dropped from the count; a true zero has to be typed as `0`.
- **Map code to SKU.** The same item gets written as a barcode, an internal SKU, or a supplier code depending on who's counting. Capture maps each entry to the canonical SKU in `stkr-inventory` using a known-aliases lookup. An entry that matches nothing is not forced — it's parked.
- **Convert cases to eaches.** Someone counts "4 boxes" of an item the system holds in single units. Capture applies the pack size from the inventory record — 4 boxes of 24 becomes 96 eaches — so the count and the system are always in the same unit before anything is subtracted.

Parked, not guessed

The one rule capture never bends: a line it can't resolve cleanly is *parked*, not guessed. A barcode that maps to no known SKU, a quantity that reads as text, a

pack size the inventory record doesn't have — each of these goes to a small parked-lines list for a human to fix, and the session notes how many are outstanding. None of them is silently dropped, and none is silently turned into a zero or a guess that would later surface as a phantom variance and waste a manager's sign-off.

This is the same instinct that runs through the whole system: the reconciler is only as trustworthy as its inputs, so the cheapest place to stop a bad number is at the door, before it ever becomes arithmetic. A parked line is a known unknown. A guessed line is a lie with a timestamp.

WHY THIS SHAPE

- The session is the unit of work, so a count is auditable from the first scan and a partial count never poses as a full one.
- Both lanes converge on S3, so the rest of the system has exactly one trigger to care about.
- A queue with a dead-letter queue sits in front of capture, so concurrent submissions and transient failures are handled, not lost.
- Normalisation is plain code, run once, before reconciliation — the engine in Part 3 only ever sees clean, single-unit, canonical-SKU lines.
- Unmappable lines are parked for a human. The system would rather ask than guess.

With a clean, session-tied list of counted lines in hand, the next post does the actual reconciliation: how a variance gets found, walked through one real line with

the numbers.

PART 3 OF 7

JUNE 27, 2026 PART 3 OF 7 · STOCK COUNT RECONCILER SERIES ~8 MIN READ

How a variance gets found

Once the count is clean, the actual reconciliation is arithmetic. For each counted line the engine looks up what the system of record says should be there, subtracts one from the other, and works out three numbers: the gap in units, the gap as a percentage of the system quantity, and the value of the gap in pounds. Then it applies a single rule — the materiality tolerance — to decide whether this gap is worth a manager's time or whether it just gets logged and moved past. This post walks one real line through that engine, with the numbers, and shows why every bit of it is plain code with no model anywhere near the decision.

KEY TAKEAWAYS

- Each counted line is joined to the system quantity from the snapshot in `stkr_inventory`.
- The gap is computed three ways: units, percent of the system quantity, and value at unit cost.
- One rule — the materiality tolerance — sorts every line into in agreement, minor variance, or material variance.
- Counted-not-in-system and in-system-not-counted are handled explicitly, not dropped.
- All of it is plain Python. No model is anywhere near the number or the decision.

The join

Reconciliation starts with a join. For each clean counted line from the capture — SKU, location, counted quantity — the engine looks up the matching row in `stkr_inventory` and reads the system quantity and the unit cost. The lookup is against the *snapshot* taken when the count session opened, not the live system, so a sale that rings through mid-count can't move the target underneath the arithmetic. Counting against a moving number is how you get variances that aren't real.

Three things can come back from the join, and each is handled on purpose:

- **Counted, and in the system.** The normal case — both numbers exist, and the engine computes the gap.

- **Counted, but not in the system.** A tin of paint on the shelf that the system has never heard of. There's no system quantity to subtract from, so this is its own kind of variance: an unrecorded item, treated as a material gap of the full counted quantity and flagged for sign-off.
- **In the system, but not counted.** A SKU the snapshot expected at this location that no line came in for. If the session covered that SKU's area, a blank is meaningful — it may be a true zero on the shelf — so it's surfaced for the manager rather than ignored. If the SKU was simply out of this session's scope, it's left alone.

The three numbers

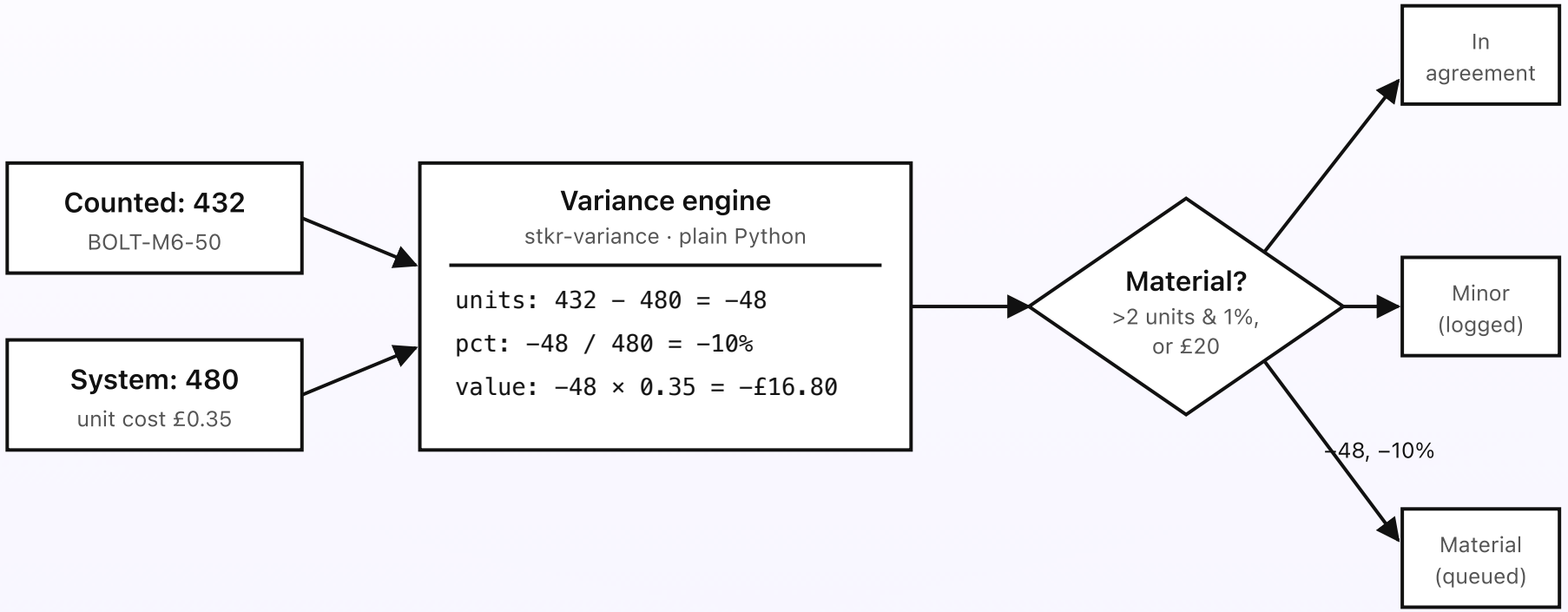
For a line that exists on both sides, the engine computes the gap three ways, because a single number lies in two different directions:

- **Units.** `counted − system`. The raw gap. Negative is a shortage; positive is an overage.
- **Percent.** `(counted − system) / system`. The same gap relative to how much there should be. Twelve units missing from a stock of 20 is a different problem from twelve missing from a stock of 5,000.
- **Value.** `(counted − system) × unit cost`. The gap in money. This is what actually hits the books, and it's what makes a small unit gap on an expensive item matter and a big unit gap on a cheap one not.

All three travel with the variance from here on, because the manager in Part 5 needs all three to judge it and the model in Part 4 needs them to suggest a cause.

A real line, with the numbers

Take SKU `BOLT-M6-50` — M6 bolts, 50 mm, stocked loose in the fixings aisle. The system snapshot says there should be **480**. The Saturday team counted **432**. Unit cost is **£0.35**.



Every number here is plain code — the model isn't consulted until a gap is already proven material.

Fig 3. One line through the engine. Counted 432 against a system 480 gives −48 units, −10%, and −£16.80. That clears both the unit and the percent thresholds, so the line is a material variance and goes on to cause suggestion. A line inside tolerance would be logged and left.

The engine runs the three sums: `432 − 480 = −48` units, which is `−48 / 480 = −10%`, worth `−48 × £0.35 = −£16.80`. Now the tolerance: a gap is material if it exceeds 2 units *and* 1% of the system quantity, or if its value exceeds £20. This gap is 48 units (well past 2) and 10% (well past 1%), so it trips the unit-and-percent test. It's a **material variance**, and it goes on to Part 4 for a likely cause. Had the team counted 479 — a gap of one unit, 0.2%, 35 pence — it would have been *in agreement*: logged, no adjustment, no manager's time spent. Had they counted 476 — four units, 0.8%, £1.40 — it would clear the unit test but not the percent test, so it's a *minor variance*: recorded for the trend, but nothing queued.

The tolerance is the whole game

The materiality tolerance is the one knob that decides how much of a manager's attention this system spends. Set it too tight and every count buries them in £1 variances; set it too loose and real shrinkage hides under the threshold. That's exactly why it lives in the rules doc, not the code: a business can tune it — tighter on high-value lines, looser on bulk consumables — without a deploy. The default of "2 units and 1%, or £20" is a starting point, not a law.

The combined test — units *and* percent — is deliberate. A unit-only rule flags trivial gaps on huge stocks; a percent-only rule flags noise on tiny ones. Requiring both keeps the flag where it belongs: a gap that's big enough to notice *and* big enough to matter relative to what should be there. The value test sits beside them as an override, so a two-unit gap on a £200 item still gets caught even though it fails the percent test.

Why there's no model here

Nothing in this post involves Bedrock, and that's the point. Subtraction, division, and a threshold comparison are things code does perfectly, cheaply, and identically every time. A model asked to "find the variances" would be slower, cost money per line, and — worst of all — might give a different answer on a re-run, which is intolerable for a number that feeds the books. The model earns its place in the next post, where the question stops being "how big is the gap" (arithmetic) and becomes "what most likely caused it" (judgement over messy signals). The reconciliation stays deterministic; the model only ever labels a number the code has already nailed down.

WHY THIS SHAPE

- The join is against the count-time snapshot, so a mid-count sale can't fabricate a variance.
- The gap is computed three ways because units, percent, and value each tell the manager something the others don't.
- One tolerance rule, in a doc, decides materiality — tunable without a deploy.
- Counted-not-in-system and in-system-not-counted are first-class outcomes, not dropped rows.
- It's all plain Python. The number that feeds the books is never produced by a model.

The gap is now proven and sized. The next post answers the harder question: of the four likely causes, which one does the evidence point to — and how the system asks a model that without ever letting it touch the stock.

PART 4 OF 7

JUNE 27, 2026 PART 4 OF 7 · STOCK COUNT RECONCILER SERIES ~8 MIN READ

How a likely cause gets suggested

A number on its own — minus 48 bolts — tells a manager what changed but not why, and the why is what decides what to do about it. Shrinkage means a security conversation. A miscount means recount before you touch anything. An unrecorded sale means a till or process problem. A receiving error means a word with the supplier. This post is about how the system turns a bare variance into a likely cause: the small bundle of evidence it assembles for each gap, the single Bedrock Haiku 4.5 call that picks one cause from a fixed list of four and explains itself in a sentence, and the hard line that keeps the model labelling numbers rather than moving stock.

KEY TAKEAWAYS

- Only material variances reach the model. Lines in agreement and minor variances never cost a call.
- For each gap the system builds a small evidence bundle: the gap, recent sales, last receiving, last count date.
- One Bedrock Haiku 4.5 call returns a cause from a fixed list of four, with a one-line rationale.
- The four causes are shrinkage, miscount, unrecorded sale, and receiving error — and nothing else.
- The model labels a number the code already computed. It cannot move stock, change a gap, or invent a SKU.

A number needs a why

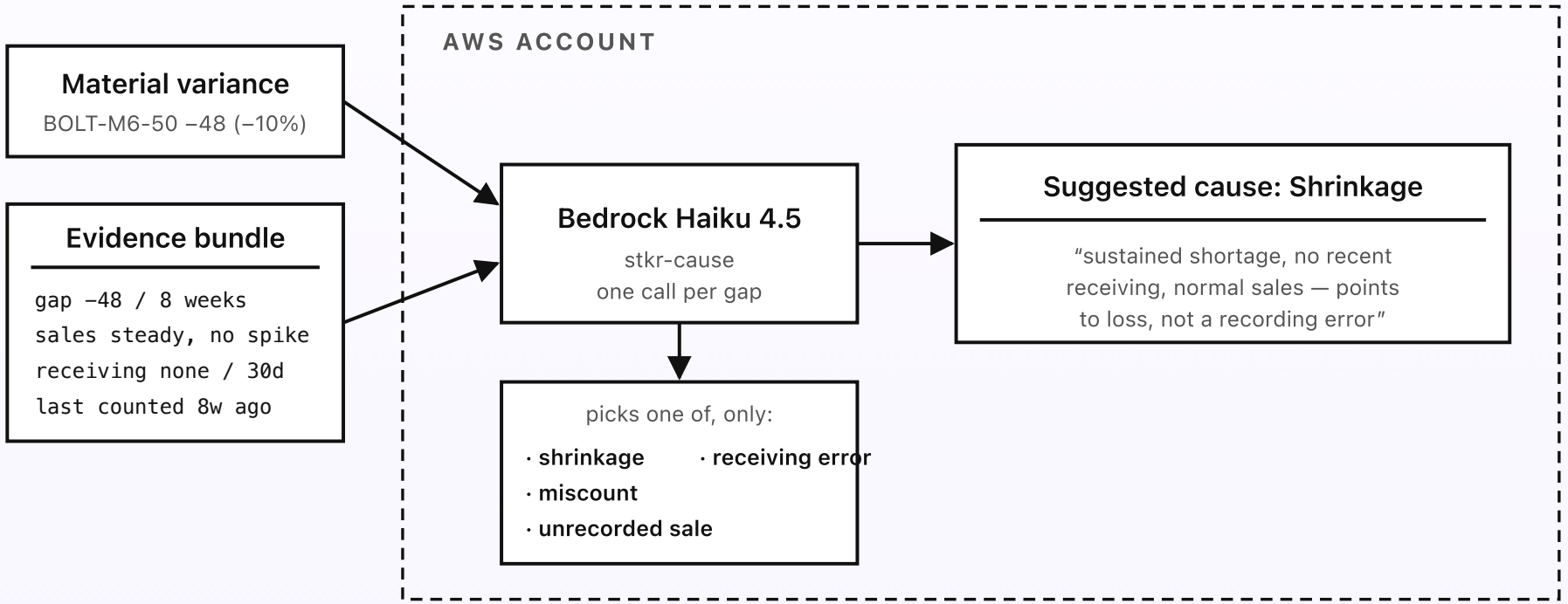
Part 3 left us with a proven, sized gap: `BOLT-M6-50`, short 48 units, −10%, −£16.80. That tells a manager *what* changed, but the action depends entirely on *why*. Shrinkage points to a security or process conversation. A miscount means recount before touching anything. An unrecorded sale points at a till or a workflow that isn't capturing every transaction. A receiving error means a word with the supplier and a check of the delivery paperwork. Same −48 on the screen; four completely different next steps. The job of this piece is to put the most likely *why* next to the number, so the manager starts from a hypothesis rather than a blank page.

The evidence bundle

The model doesn't get the whole database, and it doesn't get a vague prompt. For each material variance the engine assembles a small, structured *evidence bundle* — the handful of facts that actually discriminate between the four causes — all pulled with plain queries before any call is made:

- **The gap itself.** Direction and size in units, percent, and value. A shortage and an overage point at different causes; a 10% shortage and a 200% overage point at different causes again.
- **Recent sales for the SKU.** Units sold since the last count. A shortage that matches a sales spike is unremarkable; a shortage with flat sales is not.
- **The last receiving.** When stock was last booked in against this SKU, and how much. A gap right after a delivery smells like a receiving error; a gap with no recent delivery does not.
- **The last count date.** How long the gap had to accumulate. Forty-eight units lost over eight weeks reads differently from 48 lost since Tuesday.

Keeping the bundle small is deliberate: it makes the call cheap, keeps the model focused on the signals that matter, and means the same gap with the same evidence gets a stable answer. The model is reasoning over four or five numbers and dates, not trawling a warehouse.



The model labels the gap — it never changes the number, moves stock, or invents a SKU.
An off-list or unparseable reply falls back to "cause unknown" and still queues.

Fig 4. A material variance plus its evidence bundle go to one Bedrock call. The model returns a cause from the fixed list of four and a one-sentence rationale. It cannot return anything outside the list, and it never touches the stock.

One call, four causes

The bundle goes to a single Bedrock Haiku 4.5 call — the `stkr-cause` Lambda — via Global cross-Region inference. The prompt is tight: here is the gap, here is the evidence, pick exactly one cause from this list and give one sentence of reasoning. The list is closed:

- **Shrinkage.** Stock that left without being recorded — theft, breakage, spoilage. The signature is a sustained shortage with normal sales and no recent receiving, which is exactly our bolt line.
- **Miscount.** The count itself is wrong — a section skipped, a pack size misread, a digit transposed. The signature is often a gap that's suspiciously round or suspiciously large, or an overage that has no source.
- **Unrecorded sale.** The item left as a legitimate sale that never made it into the system. The signature is a shortage that lines up with foot traffic or a known till issue, with no matching record.
- **Receiving error.** The wrong quantity was booked in — a delivery short-shipped, double-counted at the dock, or keyed against the wrong SKU. The signature is a gap that appears right after a recent receiving.

For `BOLT-M6-50`, the evidence — a steady 10% shortfall, no delivery in 30 days, flat sales, eight weeks since the last count — points away from receiving (no recent delivery) and away from a one-off sale spike, and toward slow, unrecorded loss. The call returns *shrinkage* with the rationale: "a sustained shortage with no recent receiving and normal sales points to loss rather than a recording error." That label and that sentence ride along with the variance into the approval queue.

The guardrails on the model

This is the only place a model touches the system, so the rails are explicit:

- **It picks from a fixed list.** The four causes are the entire vocabulary. A reply that isn't one of them is rejected by the code that reads it.
- **It returns a label and a sentence, never an action.** The response schema is a cause plus a rationale — there is no field in which the model could say "post −48" even if it wanted to. The adjustment is computed by code in Part 3 and posted by a human in Part 5.
- **It fails safe.** If Bedrock errors, times out, or returns something off-list or unparseable, the variance doesn't vanish — it's queued anyway, labelled *cause unknown*, so a manager still sees the gap. A missing suggestion is a smaller problem than a hidden variance.
- **It never reads or writes stock.** The `stkr-cause` Lambda's IAM role can call Bedrock and read the evidence it was handed. It has no permission to write `stkr-inventory`. The model literally cannot move stock.

This is the same division of labour as the bill matcher: code owns every number and every decision that touches the books; the model is allowed to be helpful exactly where being occasionally wrong is cheap — a suggestion a human reads before acting. A wrong cause label costs a manager a moment's thought. A wrong number would cost real money.

WHY THIS SHAPE

- Only material variances are sent, so the model never runs on lines that don't matter — and the cost tracks variances, not SKUs.
- A small, structured evidence bundle keeps the call cheap, focused, and stable across re-runs.
- A closed list of four causes makes the output checkable — off-list replies are rejected, not displayed.
- The response is a label and a sentence, with no field that could express an action.
- Fail-safe: a model failure downgrades to "cause unknown" and still queues. The gap is never lost.
- The cause Lambda can't write inventory. The guardrail is enforced by IAM, not just intention.

The gap is sized and the cause is suggested. The last piece puts both in front of a person: how an adjustment gets approved, and how an approved one is written back to inventory with a trail behind it.

PART 5 OF 7

JUNE 27, 2026 PART 5 OF 7 · STOCK COUNT RECONCILER SERIES ~8 MIN READ

How an adjustment gets approved

A flagged variance with a suggested cause is still just a proposal. Before a single number in the system of record changes, a person has to look at it and decide. This post is about that last step: how material variances are queued, how the right manager gets an email carrying the counted quantity, the system quantity, the gap, and the suggested cause, what the two buttons actually do, and how an approved adjustment gets written back to inventory atomically with a full audit trail behind it. Nothing posts itself — and every action, all the way down to the posted adjustment, is recorded so you can reconstruct exactly who changed what, when, and why.

KEY TAKEAWAYS

- Each material variance is queued and emailed to the approver for its location, with the gap and the suggested cause.
- Two buttons, two Function URLs: *Approve* posts the adjustment; *Reject* discards it and the count line stands.
- An approved adjustment writes to `stkr-inventory` atomically, guarded against a double-tap or a stale snapshot.
- Every action — queued, approved, rejected, posted — lands in `stkr-adjustments` with a before-and-after snapshot.
- A daily sweep re-surfaces anything left unactioned, and a monthly summary reports the split of causes. No stock moves itself.

From a flag to a person

A material variance with a suggested cause is the end of the automated part and the start of the human one. The approval piece takes each one, works out who should decide it, and puts it in front of them with everything they need to decide in one glance — and nothing they don't.

The *who* comes from the rules doc: an approver per location. The fixings aisle belongs to Priya; the paint aisle to someone else. Routing by location rather than blasting every variance to one inbox keeps each manager looking only at their own stock, and keeps the sign-off close to the person who can actually walk over and recount the rack.

The email carries the full picture: SKU and name, location, counted quantity, system quantity, the gap in units, percent, and value, the suggested cause with its one-line rationale, and two buttons — *Approve* and *Reject*. For our bolt line: "`BOLT-M6-50` — M6 bolts 50 mm, fixings aisle. Counted 432, system 480. Short 48 units (–10%, –£16.80). Likely shrinkage: sustained shortage, no recent receiving, normal sales. *[Approve] [Reject]*." Quiet hours from the rules doc hold a notification back until the morning, so a count finished at 9 p.m. doesn't buzz anyone's phone.

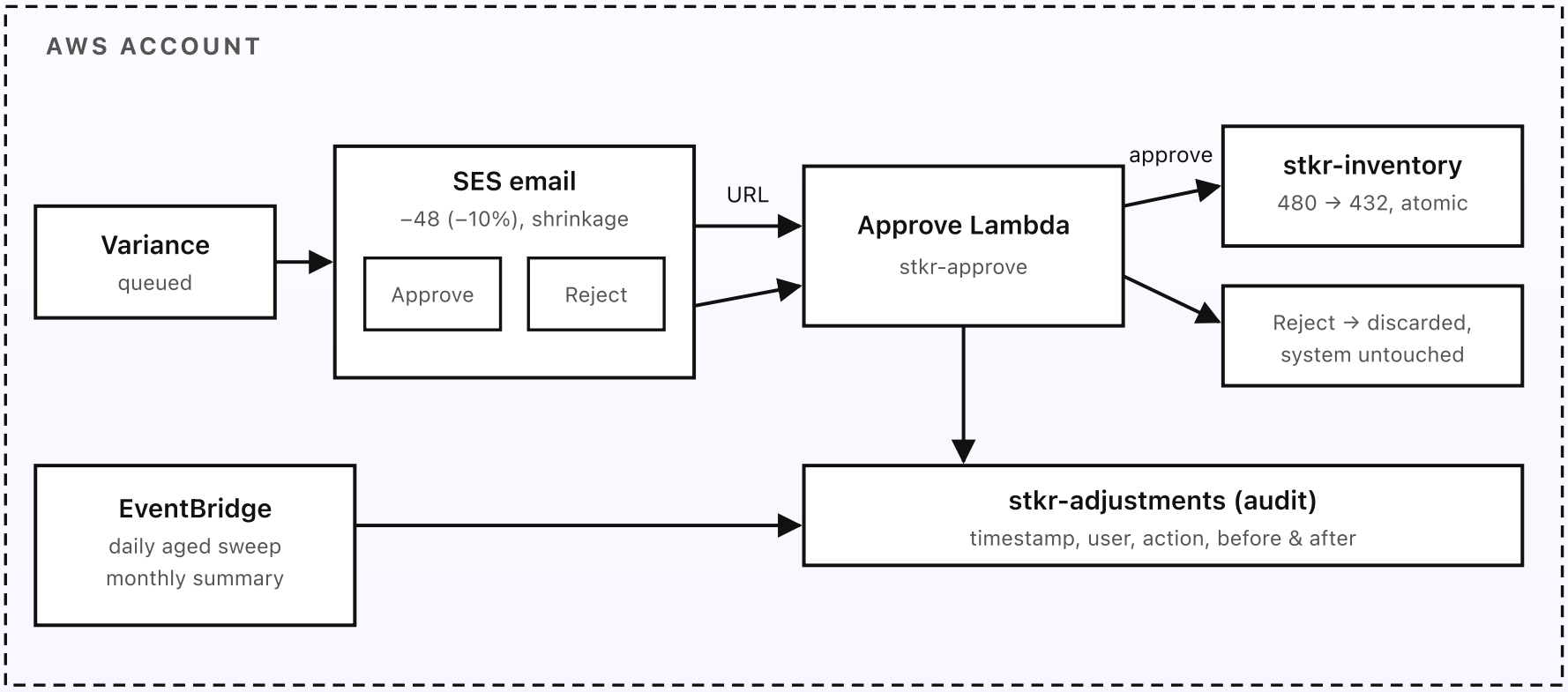
Two buttons, two Function URLs

The buttons are Lambda Function URLs — no API Gateway. Each is a signed, single-purpose link tied to that one variance, and tapping it lands on the `stkr-approve` Lambda.

- **Approve** means "the gap is real, post it." The Lambda writes the adjustment to `stkr-inventory` — in our case bringing the system quantity from 480 to 432 — and records the action. This is the only path that moves stock, and a person is always on the end of it.
- **Reject** means "don't post this." The count line stands as a recorded observation, but the system of record is left untouched. A manager rejects when they suspect a miscount and want a recount first, or when they know the cause and will handle it outside the system.

There's deliberately no third button. The bill matcher had a *Query* action because there was a supplier to write back to; a stock count has no counterparty to query, only a shelf to recount. If a manager wants a recount, they reject and open a fresh

session — which keeps the audit trail honest, because the new count is its own dated session rather than an edit to the old one.



Only Approve moves stock — every action is logged, and a re-tapped link is a no-op.

Fig 5. A queued variance is emailed to the right approver. Approve posts the adjustment to inventory atomically and writes an audit row; Reject discards it and leaves the system untouched. Both are logged. A daily sweep chases stragglers and a monthly summary reports the causes.

Posting it safely

Posting an adjustment is the one moment a number in the system of record actually changes, so it's the most carefully guarded step in the whole system.

- **It's atomic and idempotent.** Approving writes the new quantity to `stkr-inventory` as a single conditional update keyed on the variance staying in its `queued` state. If the manager taps *Approve* twice, or forwards the email and a colleague taps it too, the second write finds the variance already `approved` and does nothing. No adjustment is ever applied twice.
- **It checks the snapshot is still current.** The update also confirms the system quantity hasn't moved since the count snapshot. If stock changed between the count and the sign-off — a delivery booked in, a sale rung through — the post is held and the variance is re-surfaced for a fresh look, rather than stamping a stale number over a live one.
- **It writes the trail before it's done.** Every action writes a row to `stkr-adjustments`: the timestamp, the SKU, the action (`queued`, `approved`, `rejected`, `posted`), the user who did it, the suggested cause, and a before-and-after snapshot of the quantity. Audit a stock figure next year and you can reconstruct exactly who changed it, when, from what to what, and why.

Nothing quietly ages out

Two scheduled jobs keep the queue honest, both driven by EventBridge Scheduler. A **daily aged-variance sweep** re-surfaces any material variance still sitting unactioned past its due window, so a gap doesn't quietly age out because the one manager who could approve it was on leave. EventBridge Scheduler also

fires the **periodic count reminders** — nudging a location that hasn't been counted in too long, since a reconciler only works if counts actually happen. And a **monthly summary** writes a short narrative with one larger Bedrock call: lines counted, value caught in variances, the split across the four causes, and the SKUs that keep coming back short — the view that turns one-off sign-offs into a picture of where stock is actually leaking.

WHY THIS SHAPE

- Routing is per location, so each manager sees only their own stock and the sign-off sits next to the shelf.
- Two buttons, not three — a stock count has a shelf to recount, not a counterparty to query. A recount is a fresh, dated session.
- Posting is atomic and idempotent, so a double-tap or a forwarded link can never apply an adjustment twice.
- A stale snapshot holds the post rather than overwriting a number that moved since the count.
- Every action is logged with a before-and-after snapshot, so any stock figure is auditable long after the fact.
- A daily sweep and scheduled reminders keep both the queue and the counts themselves from quietly lapsing.

That completes the walk through the system. The next post puts a price on all of it: what the stock count reconciler actually costs to run, where the dollars go, and how the bill scales.

PART 6 OF 7

JUNE 27, 2026 PART 6 OF 7 · STOCK COUNT RECONCILER SERIES ~6 MIN READ

What the stock count reconciler costs

This is the post with the numbers. The stock count reconciler costs about $2.20 a month to run at typical small-business volume — around 2,000 SKUs counted each month, of which only a few percent come back as variances worth a model call. Here is exactly where those dollars go, why the reconciliation itself is essentially free, what scales with volume and what stays flat, and what the bill looks like at ten times the count size. The short version: the arithmetic costs nothing, the cause suggestion is a fraction of a cent per gap, and the biggest single line is a Secrets Manager secret.

KEY TAKEAWAYS

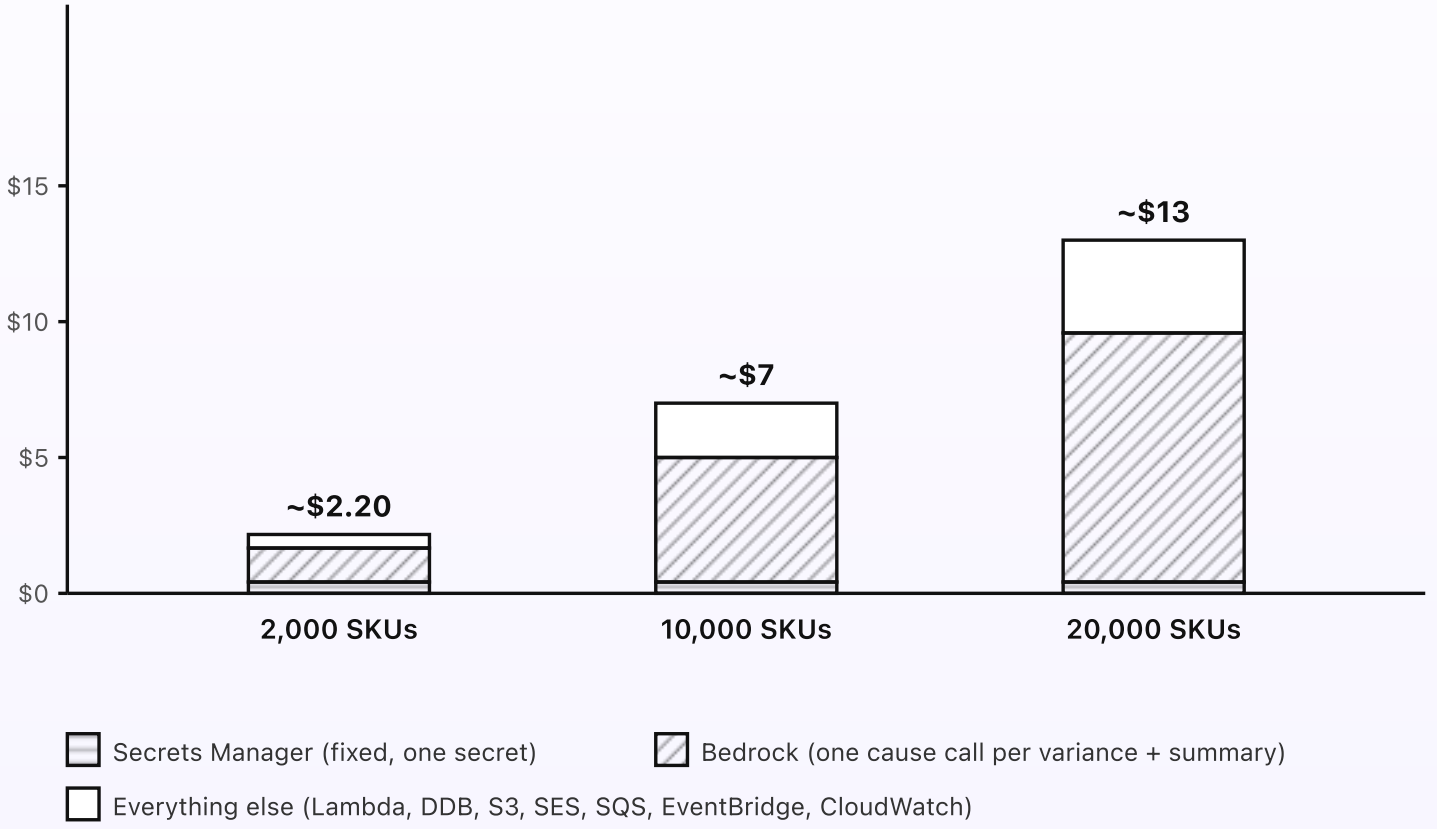
- About $2.20/month at typical SMB volume — around 2,000 SKUs counted, of which a few percent are variances.
- The single biggest line is one Secrets Manager secret at $0.40/month. The reconciliation itself is free.
- Bedrock costs track *variances*, not SKUs — only material gaps get a cause call.
- No always-on compute, no NAT Gateway, no API Gateway. Fixed cost is essentially the one secret.
- At ten times the volume — around 20,000 SKUs a month — the bill lands around $13.

The bill at SMB volume

Take a realistic small business: around 2,000 SKUs counted across the month, in a handful of sessions. Say 6% of counted lines come back as variances and clear the materiality tolerance — about 120 material gaps that each earn one Bedrock call. Here's where the $2.20 goes.

Component	What it does at this volume	~ Monthly
Secrets Manager	One secret (the system-of-record sync credential)	$0.40

Component	What it does at this volume	~ Monthly
Bedrock (Haiku 4.5)	~120 cause calls + one monthly summary	$0.85
Lambda	capture, variance, cause, approve, sweeps — all short, arm64	$0.25
DynamoDB (on-demand)	`stkr-inventory`, `stkr-counts`, `stkr-adjustments`	$0.15
SES	Manager notifications + monthly summary email	$0.05
S3	Uploaded count sheets and scanner exports	$0.05
SQS + DLQ, EventBridge, CloudWatch, Budgets	Queue, events + Scheduler, 7-day logs, the budget alarm	$0.45
Total		**~$2.20**



The variance arithmetic is free — cost rises with the number of variances, not the number of SKUs.

Fig 6. Monthly cost at three count volumes. The fixed Secrets Manager slice never moves; Bedrock grows because cause calls track the number of material variances. The reconciliation itself adds essentially nothing: it's plain Python.

Where the dollars actually go

Secrets Manager (the biggest fixed line). One secret holds the credential the sync uses to read your system of record. At $0.40 a month it is, oddly, the largest single item on the bill at SMB volume — which tells you how little everything else costs.

Bedrock (the variable line). One Haiku 4.5 call per material variance turns a small evidence bundle into a likely cause: a few hundred input tokens, a sentence out, a fraction of a cent each. At ~120 variances that's well under a dollar. The monthly summary adds one larger call — a narrative of what was caught and the split of causes — for a couple of cents. Crucially, this line tracks *variances*, not SKUs: count 2,000 clean lines and the model never fires.

Lambda. The capture, variance, cause, approve, and sweep functions are all short and arm64. The variance engine — the one that does the actual reconciling — is the cheapest of the lot: it reads a few rows, does arithmetic, writes an outcome. Well under a dollar at this volume.

DynamoDB on-demand. Three small tables. A read per counted line, a write per variance, a write per adjustment, plus the audit rows. Pennies a month.

S3, SES, and the plumbing. Count sheets and scanner exports are tiny; a few hundred MB is effectively free. SES is $0.10 per thousand messages, so the manager notifications cost cents. SQS with its dead-letter queue, EventBridge events and Scheduler, 7-day CloudWatch Logs, and the AWS Budgets alarm round out the last bucket — all negligible at this scale.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the approve/reject buttons.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Everything is event- or schedule-driven.
- **A Knowledge Base.** Inventory and counts are structured rows — deterministic lookup beats vector search here. No embeddings, no Knowledge Base.
- **Models on the maths.** The variance arithmetic is plain Python. Bedrock fires only to suggest a cause and to write the monthly summary.

How the cost scales

The only line that really grows is Bedrock, and it grows with variances rather than SKUs. Count ten times as much — 20,000 SKUs — and if the variance rate holds you get ~1,200 cause calls, which is why the bill lands around $13 rather than at twenty-some dollars: the fixed secret, the Lambdas, and the storage barely move. A business with a high variance rate pays a little more (more gaps to explain); one with tidy stock and a sensible tolerance pays less. Either way the reconciliation itself never becomes the cost.

Set an AWS Budgets alarm at $20/month so anything unusual pages you before the bill matters. The reconciler's normal-volume cost stays well under that ceiling — and a single piece of shrinkage caught and dated, instead of written off six months later, usually pays for a year of running it.

PART 7 OF 7

JUNE 27, 2026 PART 7 OF 7 · STOCK COUNT RECONCILER SERIES ~7 MIN READ

Engineering reference: the stock count reconciler architecture

This is the reference. The same stock count reconciler, drawn without the business framing — service names, resource identifiers, the region, the Bedrock model id, the Lambda inventory and their Function URLs, IAM scopes, the EventBridge rules and the Scheduler entries, the SQS and dead-letter queue, and the DynamoDB table and key schemas. If you want to build this, or review someone's build of it, this is the page to read. Everything is prefixed stkr- and lives in one region, eu-west-2.

KEY TAKEAWAYS

- Seven Lambdas (Python 3.14, arm64), one region (eu-west-2), no API Gateway, no VPC, no always-on compute.
- `stkr-approve` carries two Function URLs — approve and reject — signed and tied to a single variance.
- Three DynamoDB tables on-demand: `stkr-inventory`, `stkr-counts`, `stkr-adjustments`.
- One Bedrock model — Claude Haiku 4.5 via Global cross-Region inference — reachable only by `stkr-cause` and `stkr-summary`.
- Every IAM role is scoped to its job; only `stkr-approve` can write a stock quantity.

The architecture, drawn for engineers

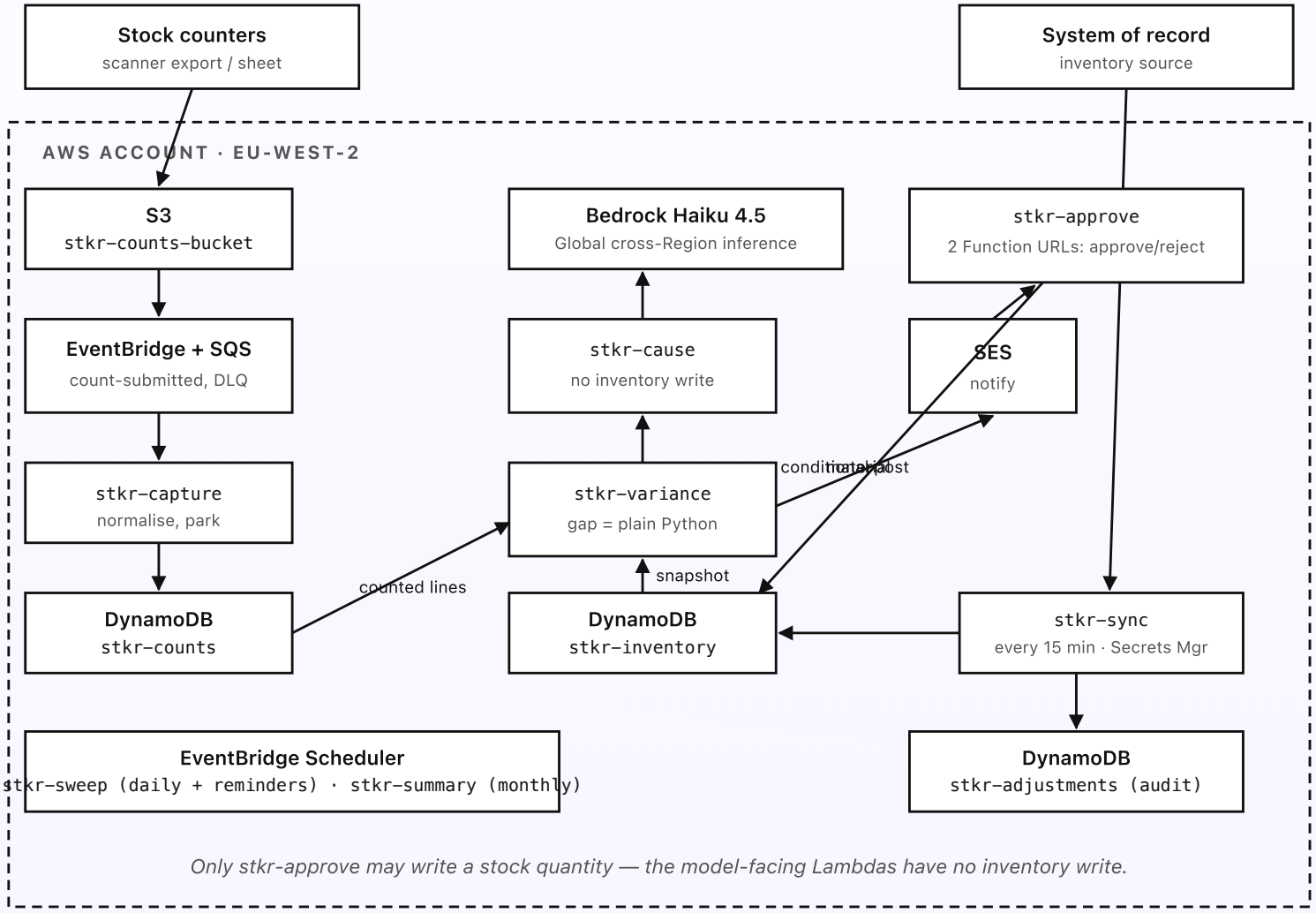

Stock counters
scanner export / sheet
System of record
inventory source
AWS ACCOUNT · EU-WEST-2
S3
stkr-counts-bucket
Bedrock Haiku 4.5
Global cross-Region inference
stkr-approve
2 Function URLs: approve/reject
EventBridge + SQS
count-submitted, DLQ
stkr-cause
no inventory write
SES
notify
stkr-capture
normalise, park
stkr-variance
gap = plain Python
snapshot
counted lines
DynamoDB
stkr-counts
DynamoDB
stkr-inventory
stkr-sync
every 15 min · Secrets Mgr
EventBridge Scheduler
stkr-sweep (daily + reminders) · stkr-summary (monthly)
DynamoDB
stkr-adjustments (audit)
Only stkr-approve may write a stock quantity — the model-facing Lambdas have no inventory write.

Fig 7. The full system in one region. A count lands in S3, an event drives capture, the variance engine reconciles against the synced snapshot, the cause Lambda calls Bedrock, and only the approve Lambda — behind two Function URLs — ever writes a stock quantity. Everything is logged to `stkr-adjustments`.

Lambdas (Python 3.14, arm64)

- `stkr-capture` — triggered by the SQS queue behind the S3 count-submitted event. Normalises the scan/sheet, ties lines to the open session, writes clean counted lines to `stkr-counts`, parks unmappable lines. No Function URL.
- `stkr-variance` — runs per captured session. Joins counted lines to the `stkr-inventory` snapshot, computes gap in units/percent/value, applies the materiality tolerance, writes variances, and invokes `stkr-cause` per material gap. Plain Python; no Bedrock. No Function URL.
- `stkr-cause` — builds the evidence bundle and makes one Bedrock call, returning a cause from the fixed list of four plus a rationale. IAM allows `bedrock:InvokeModel` and reads; **no** write to `stkr-inventory`. No Function URL.
- `stkr-approve` — **two Lambda Function URLs**, one for *approve*, one for *reject*, each a signed link scoped to a single variance. Approve does the conditional post to `stkr-inventory`; both write an audit row to `stkr-adjustments`. The only Lambda permitted to write a stock quantity.
- `stkr-sync` — fired by EventBridge Scheduler every 15 minutes. Reads the system of record using a credential from Secrets Manager and mirrors it into `stkr-inventory`. No Function URL.

- `stkr-sweep` — daily via Scheduler. Re-surfaces aged unactioned variances and issues periodic count reminders for stale locations. No Function URL.
- `stkr-summary` — monthly via Scheduler. One larger Bedrock call for the narrative summary, sent by SES. No Function URL.

EventBridge rules and schedules

- **Rule — count-submitted.** S3 `ObjectCreated` under `sessions/*/` → SQS (`stkr-capture-queue`) with a redrive to `stkr-capture-dlq` after 2 attempts → `stkr-capture`.
- **Schedule — sync.** `rate(15 minutes)` → `stkr-sync`.
- **Schedule — daily sweep + reminders.** `cron(0 7 * * ? *)` → `stkr-sweep` (respecting quiet hours from the rules doc).
- **Schedule — monthly summary.** `cron(0 8 1 * ? *)` → `stkr-summary`.

DynamoDB tables (on-demand) + key schema

- `stkr-inventory` — PK `LOC#<location>`, SK `SKU#<sku>`. Attributes: `name`, `system_qty`, `unit_cost`, `pack_size`, `aliases`, `snapshot_version`. Written only by `stkr-sync` and `stkr-approve`.
- `stkr-counts` — PK `SESSION#<id>`, SK `META` for the session header (`location`, `counter`, `opened_at`, `status`) and SK `LINE#<sku>` per counted line (`counted_qty`, `state`, `gap_units`, `gap_pct`, `gap_value`, `cause`).
- `stkr-adjustments` — PK `SKU#<sku>`, SK `TS#<iso8601>#<variance_id>`. Attributes: `action` (`queued` / `approved` / `rejected` / `posted`), `user`, `cause`,

`qty_before`, `qty_after`, `session_id`. Append-only audit trail.

S3, SES, Secrets Manager

- **S3 — `stkr-counts-bucket`.** Versioning on. Layout `sessions/<session-id>/{scan,sheet}.csv`. Lifecycle: transition raw uploads to Glacier after 90 days.
- **SES.** Outbound only — approver notifications and the monthly summary, through one configuration set for bounce/complaint tracking. No inbound rule set (there is no email lane).
- **Secrets Manager.** One secret, `stkr/sync-credential`, read only by `stkr-sync`.

IAM scopes, Bedrock, region

- **IAM — least privilege per function.** `stkr-variance`: read `stkr-counts` + `stkr-inventory`, write variances, invoke `stkr-cause`. `stkr-cause`: `bedrock:InvokeModel` + read only — *no* inventory write. `stkr-approve`: conditional `UpdateItem` on `stkr-inventory` + write `stkr-adjustments`. `stkr-sync`: read the secret, write `stkr-inventory`. No role has more than its step needs.
- **Bedrock model id.** `global.anthropic.claude-haiku-4-5-20251001-v1:0`, invoked via Global cross-Region inference. Reachable only by `stkr-cause` and `stkr-summary`.
- **CloudWatch Logs.** 7-day retention on every function's log group.
- **AWS Budgets.** One monthly budget at $20 with an alarm action.

- **Region.** Everything in `eu-west-2`. No API Gateway, no VPC, no NAT Gateway, no always-on compute.

THE WHOLE SYSTEM, IN ONE BREATH

- A count lands in S3, an event drives `stkr-capture`, and clean lines go to `stkr-counts`.
- `stkr-variance` reconciles against the `stkr-inventory` snapshot in plain Python and flags material gaps.
- `stkr-cause` labels each gap with one Bedrock call — and cannot touch stock.
- `stkr-approve`, behind two Function URLs, is the only thing that posts an adjustment — after a human taps it.
- Every action is appended to `stkr-adjustments`, so the books are always auditable.

That is the stock count reconciler end to end: a count in, a reconciled and explained set of variances out, and not a single stock quantity changed without a person's sign-off and a row in the audit trail. It reconciles what you have — deciding what to buy is a different system entirely.