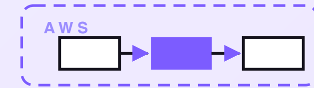
7-PART SERIES · FREE COMPANION



Weekly report builder

A serverless builder that gathers a small business's own numbers — sales, new customers, cash in and out, top items — from the sources you point it at, compares them to last week and last month, writes a short plain-English "here's how the week went, here's what changed" summary grounded in the real figures, and emails it every Monday. It only reports facts from your data, never invents a number, and flags anything that looks off. 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.allannital.dev/w/weekly-report-builder

CONTENTS

Weekly report builder

01 A weekly report builder on AWS for a few dollars a month
02 How the numbers get gathered
03 How the weekly report gets written
04 How the report reaches the owner
05 How a number that looks off gets flagged
06 What the weekly report builder costs
07 Engineering reference: the weekly report builder architecture

PART 1 OF 7

JUNE 18, 2026 PART 1 OF 7 · WEEKLY REPORT BUILDER SERIES ~5 MIN READ

A weekly report builder on AWS for a few dollars a month

Most small-business owners can’t tell you on a Monday morning how last week actually went. The numbers exist — sales are in one place, new customers in another, cash in and out in the bank export, the top-selling items buried in the point-of-sale — but nobody has time to open four tabs, line them up against the week before, and work out what changed. So the question goes unanswered, week after week, until something is badly wrong and it’s obvious to everyone. This post walks through the design of a small builder that gathers all those numbers, compares them, writes a short plain-English summary of how the week went, and emails it every Monday — reporting only what’s really in your data.

KEY TAKEAWAYS

- Three pieces: a gather step, a writer step, and a sender step. The report goes out every Monday.
- Every figure is computed by plain Python from your data before any words are written.
- Each number is compared to last week and last month, so the report says what changed, not just what happened.
- One AI call a week turns the real numbers into a short paragraph; it never sources a number itself.
- Designed on AWS for about $2/month at typical small-business volume.

The whole system on one page

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

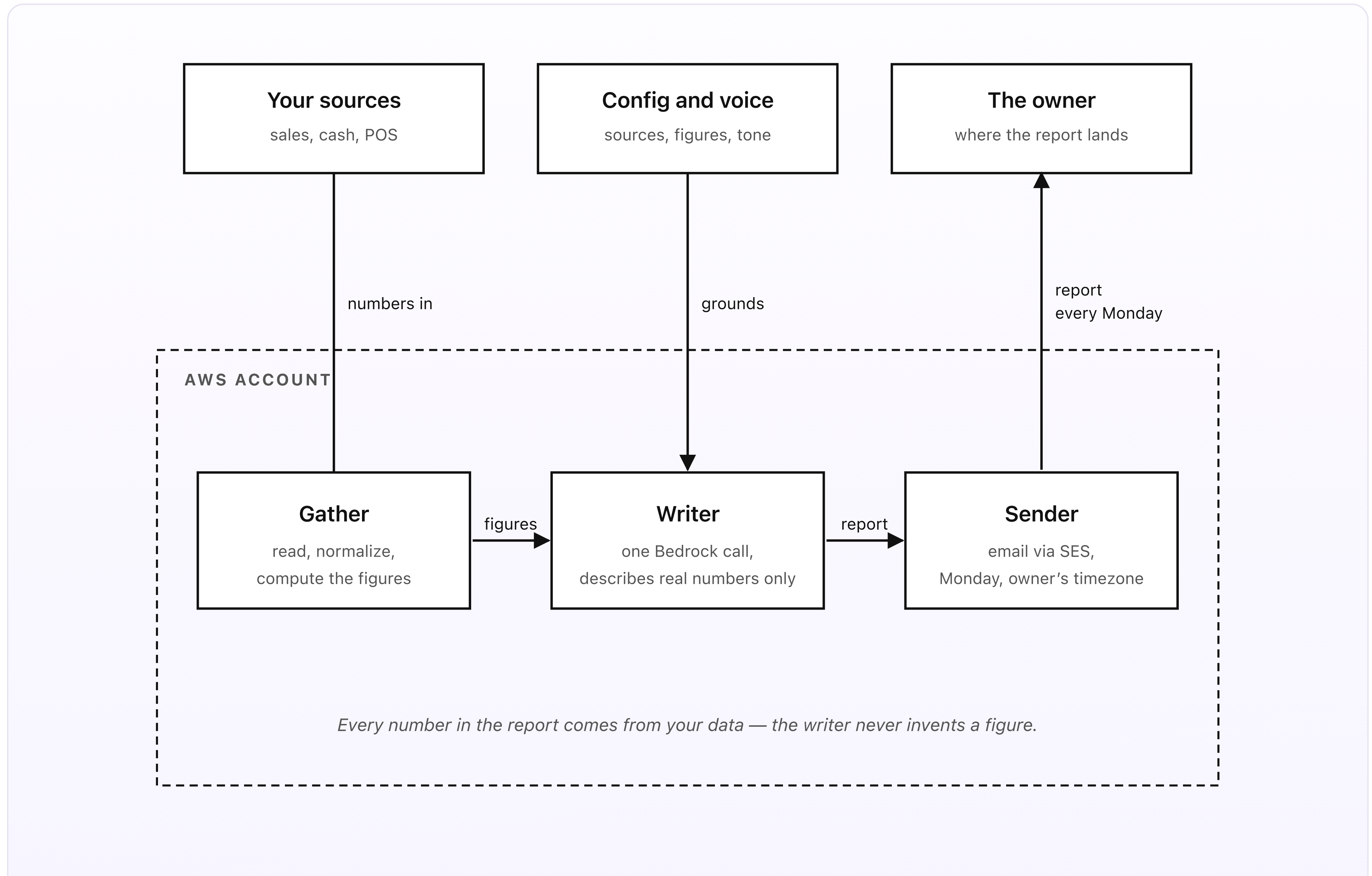

Your sources
sales, cash, POS
Config and voice
sources, figures, tone
The owner
where the report lands
numbers in
grounds
report
every Monday
AWS ACCOUNT
Gather
read, normalize,
compute the figures
figures
Writer
one Bedrock call,
describes real numbers only
report
Sender
email via SES,
Monday, owner's timezone
Every number in the report comes from your data — the writer never invents a figure.

Fig 1. Your sources outside, three pieces inside AWS. Numbers flow in from a sales sheet, a cash export, and a point-of-sale summary. The Gather computes the figures, the Writer turns them into a short paragraph, and the Sender emails the report every Monday.

What you set up once (the outside)

- **Your sources.** The places your real numbers already live. A sales export in a Google Sheet, a Stripe or bank CSV dropped in a Drive folder, a point-of-sale daily summary. You don't move any of this — you just point the builder at it. Each source is listed once in the config doc with where it lives, which columns hold the figures that matter (revenue, order count, new customers, refunds), and how often it updates. Adding a new source later is one new line in the doc.
- **A config and voice folder.** Two short Google Docs in a Drive folder. The *config* doc lists every source and the figures to pull from each, plus the thresholds that decide what counts as a notable change ("flag any week-over-week move bigger than 25%") and what makes a number look off enough to flag. The *voice* doc holds the tone and shape of the summary paragraph — short, plain, owner-facing, lead with the headline number. Changing either is an edit to a doc, never a deploy.
- **The owner.** The person the report is for. They get the email every Monday morning in their own timezone, with the plain-English summary at the top, the numbers table right below it, the comparisons to last week and last month, and a *Needs a look* section for anything the checks flagged. No login, no dashboard to remember — it's just in the inbox.

What runs every Monday (the inside)

- **The gather.** Reads each source. A small `source-sync` Lambda mirrors every source to S3 on a schedule, so the Monday run reads from S3 and never depends on a live API at 7am. The gather step normalizes each source into one clean shape, computes this week's figures (total sales, order count, new customers, cash in and out, top items), and computes the same figures for last week and the four-week average — so every number arrives with its comparison already attached. This step is plain Python. No model.
- **The writer.** Takes the already-computed figures and the comparisons and makes exactly one Bedrock Haiku 4.5 call to write a short paragraph: "here's how the week went, here's what changed." The prompt hands the model the real numbers and tells it to describe only those numbers — no figure may appear in the summary that wasn't computed in the gather step. The model writes prose, not data. Everything it says is checked back against the table before sending.
- **The sender.** Formats the email: the summary paragraph up top, the numbers table below it with this week / last week / four-week average columns, and the *Needs a look* section if anything tripped a check. Sends via SES outbound at the configured Monday time in the owner's timezone. Every send is recorded in DynamoDB so the run is auditable and so next week's comparison has a clean record of what was reported. A link in the footer hits a Function URL if the owner wants the full source table for any figure.

In plain words

It's Monday, 7am. Over the weekend the bank export landed in the Drive folder, the point-of-sale rolled up Saturday's numbers, and the sales sheet got its last few

rows. The builder wakes up, reads all three, and works out the week: $18,400 in sales (up 12% on last week, a touch above the four-week average), 142 orders, 9 new customers (down from 14 — the slowest week in a month), cash out higher than usual because the quarterly insurance premium cleared. It writes one short paragraph saying exactly that, in plain words, and emails it to the owner with the table underneath. The owner reads it over coffee in ninety seconds and knows two things they wouldn't otherwise: sales are fine, but new customers slowed down — worth a look at the ad spend.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the quarter where new-customer growth quietly stalled for six weeks and nobody noticed until the pipeline ran dry.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every figure is computed by plain Python from your data first. The writer only describes numbers that already exist.
- Every number ships with its comparison — last week and the four-week average — so the report says what changed.
- The summary and the table travel together. Any sentence can be checked against the numbers right below it.
- Anything missing, stale, or out of range is flagged, not guessed. A blank is never reported as a real result.
- The config lives in Drive. Adding a source or changing a threshold doesn't need a deploy.
- Every send is logged. Pull up any past Monday and you can see exactly what was reported and why.

Why this shape

Most owners get their numbers in one of three ways: a stack of dashboards nobody opens, a bookkeeper's month-end report that arrives three weeks too late to act on, or a gut feel that's right until it isn't. The dashboards fail because looking at them is a chore you have to remember to do. The month-end report fails because by the time it lands, the week it describes is ancient history. And gut feel fails the first week something moves quietly — new customers, refund rate, average order — in a direction the owner wasn't watching.

The setup above leaves your numbers where they already live, but adds a small system that *reads* them every week and does the lining-up and comparing for you. The report comes early enough in the week to act on. It says what changed, not just what happened. It shows the table next to the words so you can trust any sentence. And it flags what looks off instead of papering over it. The builder is invisible six days a week; visible only on Monday, when it answers the one question the owner never has time to answer themselves.

The next four posts walk through each piece in turn: how the numbers get gathered, how the weekly report gets written, how the report reaches the owner, and how a number that looks off gets flagged. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 18, 2026 PART 2 OF 7 · WEEKLY REPORT BUILDER SERIES ~4 MIN READ

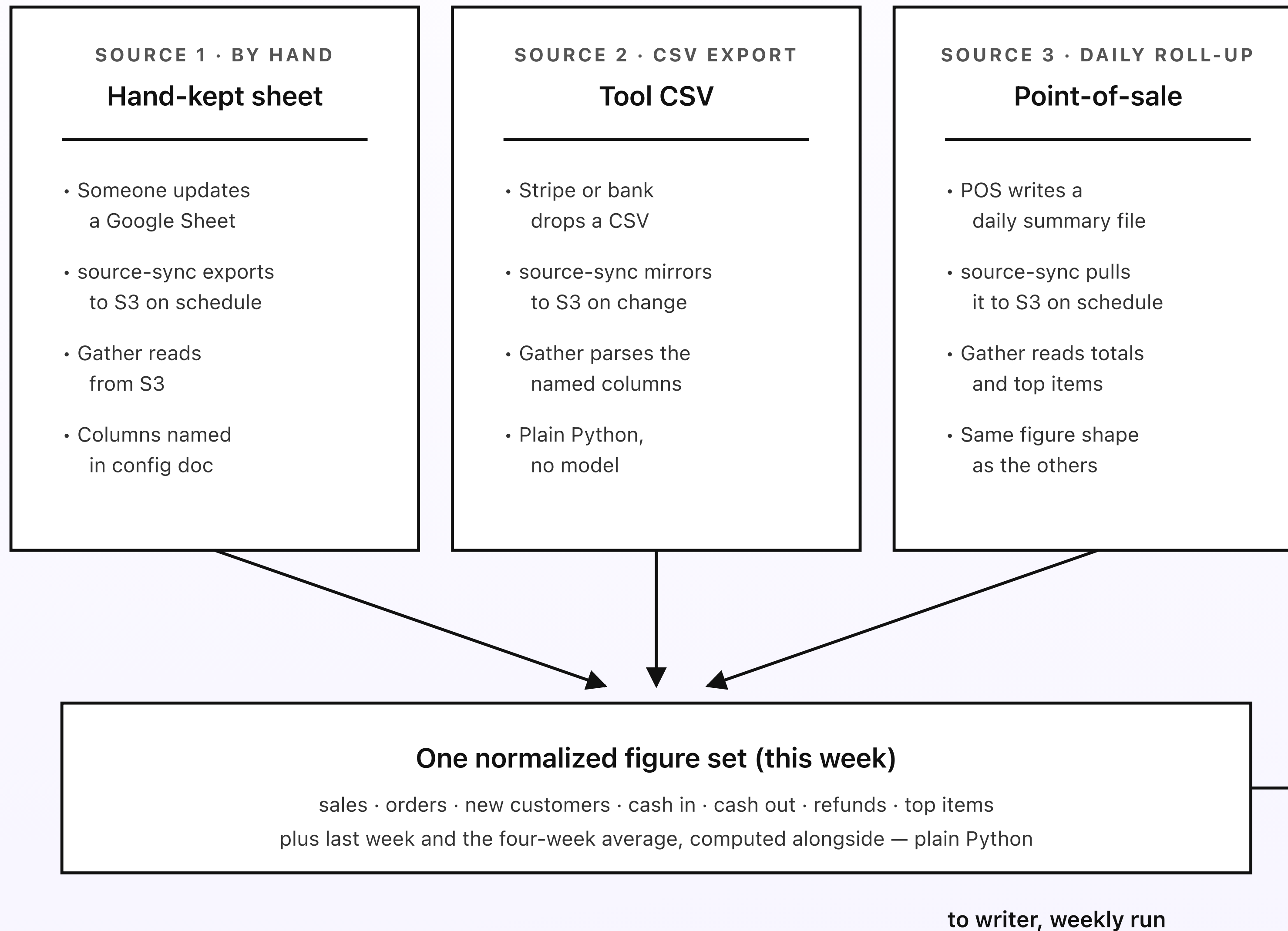
How the numbers get gathered

The report is only as good as the numbers behind it. So the first job is reading every source cleanly and turning it into one set of weekly figures, without ever depending on a live system at 7am on a Monday. There are three kinds of source a small business usually has — a sheet someone keeps by hand, a CSV that a tool like Stripe or the bank produces, and a point-of-sale roll-up. Each gets mirrored to S3 on its own schedule, and then the gather step reads them all from one place.

KEY TAKEAWAYS

- Three kinds of source feed one set of figures: a hand-kept sheet, a tool CSV, and a point-of-sale roll-up.
- A small `source-sync` Lambda mirrors each source to S3 on a schedule, so the Monday run never hits a live API.
- The gather step reads everything from S3 and normalizes it into one clean shape.
- It computes this week, last week, and the four-week average, so each figure arrives with its comparison.
- S3 versioning means a bad source edit can be rolled back in one click.

Three sources into one set of figures



Each source is mirrored to S3 first — the Monday run never depends on a live system being up at 7am.

Fig 2. Three kinds of source converge into one figure set. Each is mirrored to S3 on its own schedule; the gather step reads everything from S3 and normalizes it into one shape, with last week and the four-week average computed alongside this week.

Source 1: the hand-kept sheet

The most common source in a small business is a sheet somebody keeps by hand — the sales log, the new-customer list, the running tally of jobs done. The config doc names the sheet, the tab, and which columns hold which figure: which column is the amount, which is the date, which marks a new customer. A small Lambda — `source-sync` — runs on a schedule, exports the sheet as plain CSV via the Google Sheets API, and writes it to `s3://wr-source-data/sales-sheet.csv` only if the sheet has changed since the last sync. The gather step reads from S3, not Drive directly. That keeps Sheets API calls predictable and gives you S3 versioning for free, so a bad bulk-edit on Friday can be rolled back in one click on Monday.

This covers the figures a person types in as the week goes on. The builder never edits the sheet — it only reads a copy.

Source 2: the tool CSV (the one most owners already have)

Almost every business has at least one tool that already produces a clean export: Stripe for card payments, the bank for the account statement, the invoicing app for what was billed and what was paid. These arrive as CSV files. You set up the tool (or a tiny scheduled export) to drop the file in a Drive folder, and `source-sync` mirrors it to `s3://wr-source-data/` whenever it changes. The gather step parses the columns named in the config doc — amount, date, customer, status — and pulls only the figures the report needs. No model reads the CSV; it's plain

column-by-column Python, which is exactly what you want for money figures: predictable, checkable, and the same every run.

Because these exports are the system of record for cash, the builder treats them as authoritative for the cash-in and cash-out figures. If the sales sheet and the Stripe export disagree on a total, that disagreement is itself something the checks in Part 5 will surface — rather than the builder silently picking one.

Source 3: the point-of-sale roll-up

A shop or a café has a point-of-sale system that already knows the daily totals and the best-selling items. Most of them can write a daily summary — a small file with the day's sales, order count, and top products. The config doc points `source-sync` at wherever that file lands, and it's pulled to S3 on a schedule like the others. The gather step reads the daily totals for the week and the top-item list, so the report can say not just "sales were up" but "the lunch special drove most of it."

Point-of-sale is the most optional of the three. A business without one loses nothing; a shop that has one gets the top-items line in the report, which is often the most useful sentence in it.

Why everything funnels into one figure set

Three sources in, but only one set of figures the writer ever sees. That's deliberate. If the writer had to reach into three different files in three different shapes, every "where did this number come from?" question would mean checking three places and three formats. Normalizing everything into one figure

set — with a fixed list of fields and a fixed shape — means there is exactly one place the report's numbers come from, and exactly one place to look when a figure seems wrong. The gather step does the messy reading; everything downstream works from one clean set.

Next post: how the weekly report gets written — how plain Python computes every comparison first, and how exactly one Bedrock call turns those real numbers into a short paragraph.

PART 3 OF 7

JUNE 18, 2026 PART 3 OF 7 · WEEKLY REPORT BUILDER SERIES ~5 MIN READ

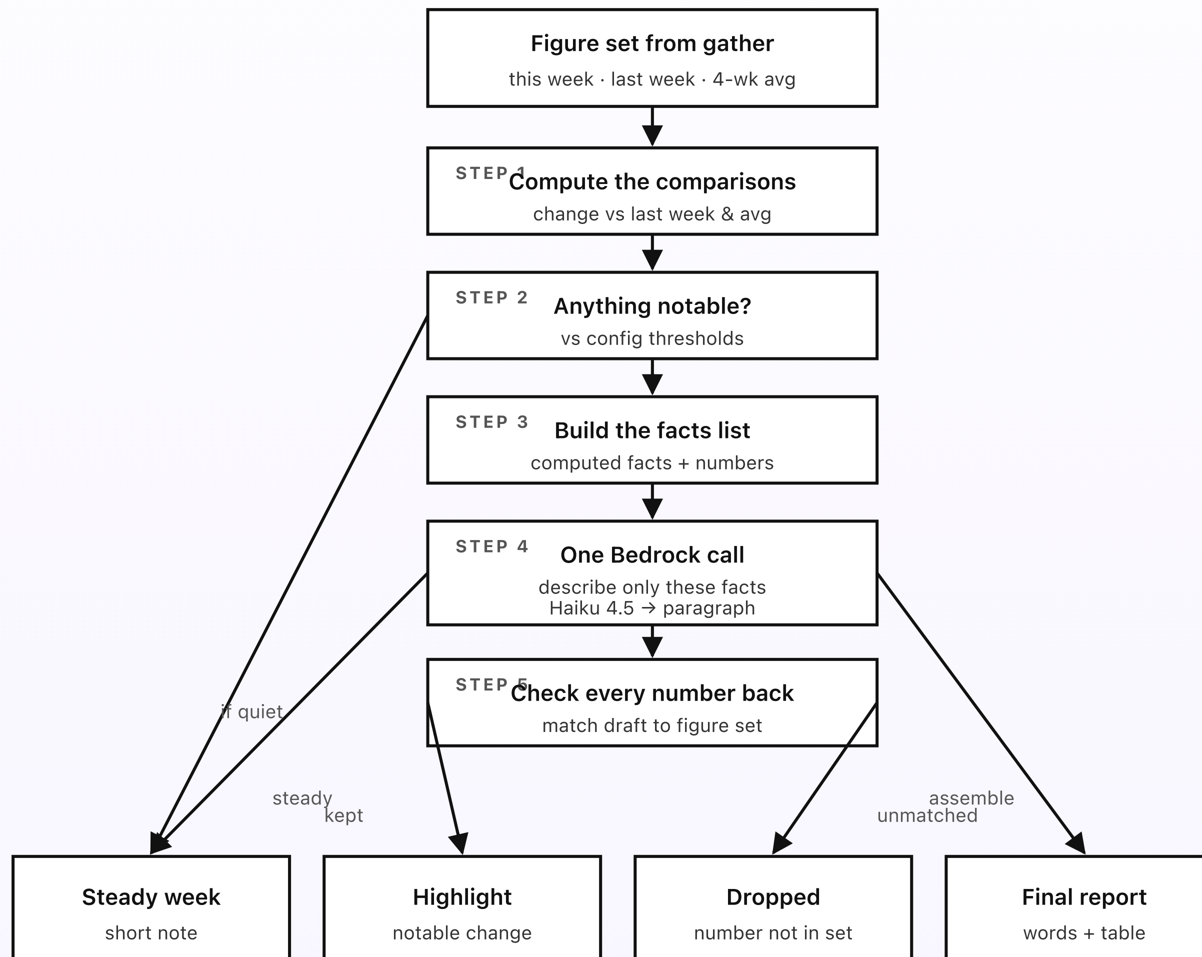
How the weekly report gets written

The gather step handed over one clean figure set. Now the report has to be written — and this is the step where it would be tempting to just feed everything to a model and let it talk. The builder doesn't. Every figure and every comparison is computed in plain Python first. The model is the last step, not the first, and it only ever writes prose about numbers that already exist. This post walks through that order, one step at a time.

KEY TAKEAWAYS

- Plain Python computes every figure and every comparison before the model is touched.
- What counts as a “notable change” is a threshold in the config doc, not the model’s opinion.
- The one Bedrock call is handed the real numbers and told to describe only those.
- After writing, every figure in the paragraph is checked back against the computed set.
- If a sentence references a number that isn’t in the set, that sentence is dropped, not sent.

The write flow, step by step



The model writes the words — Python owns every number; change a threshold, next week uses it.

Fig 3. How the report is written, step by step, on the weekly run. Five steps turn one figure set into a checked report. Python computes and verifies every number; the model only writes the prose.

Comparisons first, in plain Python

The gather step already produced this week, last week, and the four-week average. Step one computes the changes: how much each figure moved versus last week, and how it sits against the four-week average. That's simple arithmetic — percentages and differences — and it's done in Python where it can be tested and trusted. The point of doing it here, before any writing, is that the report's "up 12%" or "down from 14" is a fact the code computed, not a phrase a model produced and hoped was right.

Comparing to both last week and the four-week average matters because a single week can mislead. Sales down 8% on last week sounds bad until you see last week was the best week of the quarter and this week is still above the average. The report carries both comparisons so a one-off doesn't read like a trend.

What counts as "notable" is in the config, not the model

Step two decides which changes are worth calling out. This is a threshold in the config doc, not a judgment the model makes. The doc says things like "flag any week-over-week move bigger than 25%" or "always call out new customers, even small moves." The code applies those thresholds and marks each figure as either background or a highlight. If nothing crosses a threshold, the week is steady, and

the report is honest about that — a short "quiet week, nothing moved much" note beats a model padding three paragraphs out of noise.

Keeping "notable" in the config means the owner controls what the report gets excited about. A business that lives and dies on new customers turns that threshold down; a business with naturally spiky weekly sales turns that one up so it isn't flagged every Monday.

The one model call, and what it's allowed to do

Step four is the only place a model touches the report. The code builds a facts list — a fixed set of computed statements, each with its number attached: "sales: $18,400, up 12% on last week, above the four-week average"; "new customers: 9, down from 14, the slowest week in a month." That list is handed to Claude Haiku 4.5 with a short instruction: write a few plain sentences that describe these facts for a busy owner, lead with the headline, keep it short, and do not introduce any number that isn't in the list.

The model is good at exactly this — turning a list of facts into readable prose — and bad at exactly the thing it's not being asked to do, which is compute or recall figures. By the time it's called, there is nothing for it to get wrong about the numbers, because all the numbers are already decided. It's writing, not analyzing.

Checking the numbers back before anything sends

Step five is the guardrail. After the model returns its paragraph, the code scans the draft for every figure it mentions and matches each one against the computed

set. A number that matches stays. A number that doesn't — a stray figure, a rounded value that drifted, anything the model added — gets the sentence containing it dropped, and the drop is logged. In practice Haiku 4.5 handed a tight facts list rarely invents a number, but "rarely" isn't "never," and a weekly report that an owner makes decisions on has to be never. The check is cheap and absolute: if a sentence's number isn't in the set, the owner never sees that sentence.

This is why the email always carries the numbers table next to the summary. The words are the convenient version; the table is the source of truth, and any sentence in the summary can be checked against the row right below it.

Next post: how the report reaches the owner — the Monday send, the timezone handling, and the four checks between the written report and the inbox.

PART 4 OF 7

JUNE 18, 2026 PART 4 OF 7 · WEEKLY REPORT BUILDER SERIES ~5 MIN READ

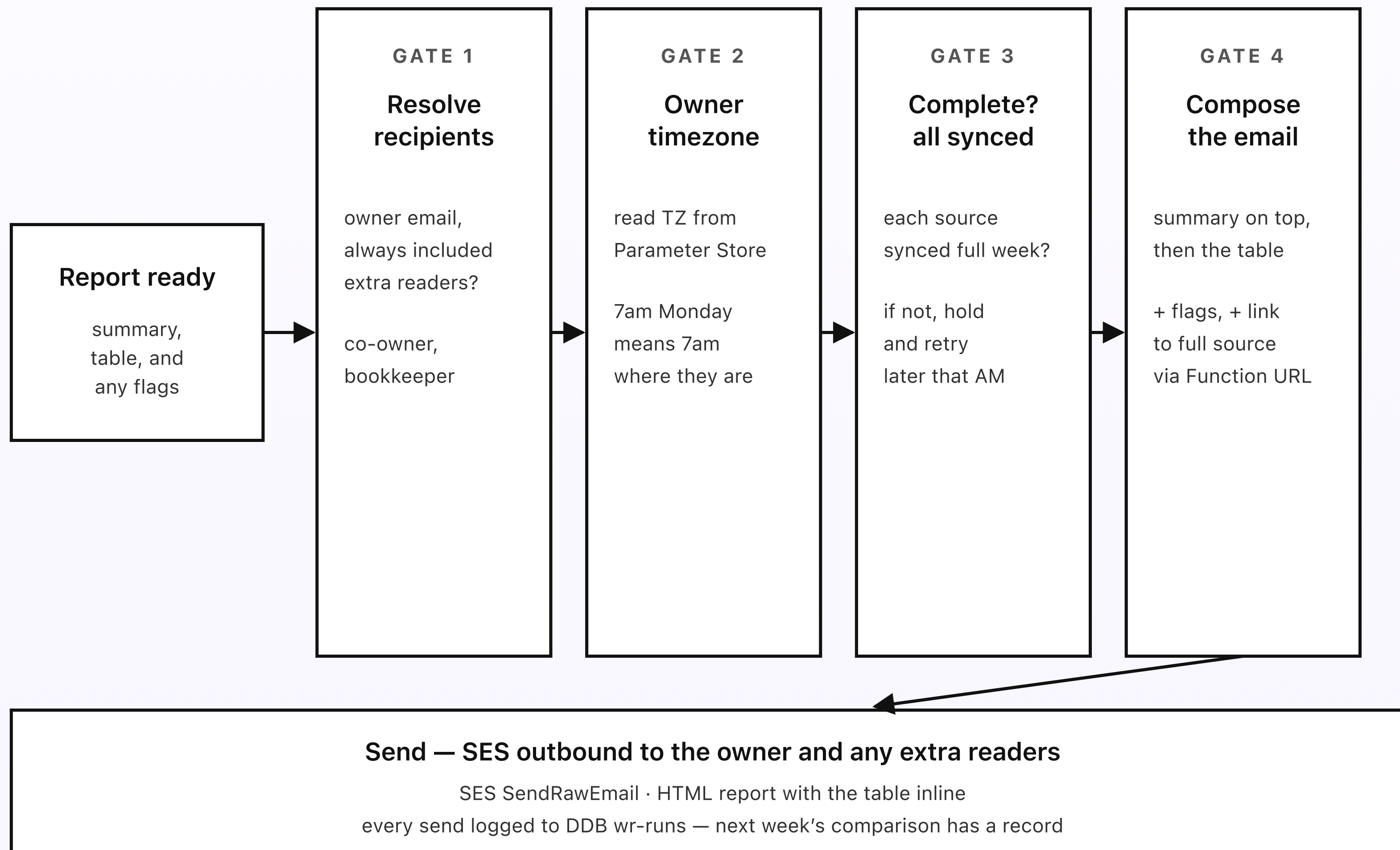
How the report reaches the owner

The report is written and the numbers are checked. Now the send step has to put it in the right inbox, at the right local time, only when the week's data is actually complete, and in a shape that reads well on a phone over coffee. Get any of those wrong and the report is worse than no report: a 2am email, a half-week of numbers because a source hadn't synced yet, a wall of figures nobody scrolls past. Four small checks sit between the written report and the actual send.

KEY TAKEAWAYS

- Recipient resolution: the configured owner, plus any extra readers listed in the config doc.
- The Monday run fires in the owner’s timezone, so 7am means 7am where they are.
- A completeness check holds the send if a source hasn’t synced for the full week yet.
- The email leads with the summary, then the numbers table, then anything flagged.
- Every send is logged so the run is auditable and next week’s comparison has a clean record.

Four checks on every send



Every check is a deterministic gate — no model calls, no surprise behavior on a quiet Monday.

Fig 4. Four checks between the written report and the inbox. Resolve the recipients. Honor the owner's timezone. Hold if the week's data isn't complete. Compose with summary, table, and flags. Then send via SES and log the run so next week has a clean record.

Gate 1: resolve the recipients

The report is for the owner, but it's often read by more than one person. The config doc lists the owner's email and any extra readers — a co-owner who wants the same Monday view, a bookkeeper who likes to see the cash figures, a manager who runs a location. Gate 1 reads that list. The owner is always on it; the extras are opt-in. If the list is somehow empty (a config mistake), the report goes to the admin fallback rather than to nobody, and the run is logged with a note so the config can be fixed.

There's no per-reader customization here on purpose — everyone gets the same report. The point of the system is one shared, trusted weekly view, not five slightly different ones that drift apart.

Gate 2: the owner's timezone

"Every Monday morning" only means something if it's the owner's Monday morning. The EventBridge schedule that fires the run is set in the owner's timezone, read from Parameter Store, so the default 7am send lands at 7am wherever the business actually is — not 7am UTC, which could be the middle of the night. Gate 2 is mostly a guard: it confirms the run really is firing at the configured local time and the timezone setting looks sane before anything goes

out. The failure it prevents is the embarrassing one — a CI rotation quietly resetting the schedule to UTC and the owner getting their report at 2am.

Gate 3: is the week actually complete?

A report built from half the week's data is worse than no report, because it looks authoritative while being wrong. Gate 3 checks each source's last sync time against the week the report is supposed to cover. If every source has synced through Sunday, the send proceeds. If one hasn't — the bank export didn't land, the point-of-sale was offline Saturday night — the send is held, and a one-off EventBridge Scheduler rule retries it a couple of hours later that same morning. If a source is still missing by the final retry, the report goes out anyway, but that source is flagged in the *Needs a look* section as incomplete, so the owner knows a number is partial rather than assuming it's the real total.

The trade-off favors a slightly late but complete report over a punctual but partial one. A report that's an hour late is fine; a report that quietly understates sales because Saturday hadn't synced is not.

Gate 4: compose, then send

The voice doc sets the shape of the email: the plain-English summary right at the top so the owner gets the gist in the preview pane, then the numbers table with this-week / last-week / four-week-average columns, then the *Needs a look* section if anything was flagged. The table is inline HTML so it reads on a phone without downloading anything. A footer link points to a Function URL where the owner can

pull the full underlying rows for any figure — useful when a number surprises them and they want to see exactly which orders made it up.

Once composed, the report ships via SES outbound to the resolved recipients. The sender identity is a verified address on the business's domain so the email lands in the inbox, not the spam folder. Every send — recipients, the figures reported, and the run timestamp — writes a row to `wr-runs` in DynamoDB. Next Monday's run reads that row to build its "versus last week" comparison from exactly what was reported, not a recomputed guess.

Why the checks exist

None of these gates are clever. They're the small care a thoughtful person would take if they were sending the report by hand — make sure it goes to the right people, don't send it at 2am, don't send it until the week's actually finished, and lay it out so it's readable in ten seconds. Putting them in code as four sequential checks makes them part of the system, not something you're trusting yourself to remember at 7am every Monday for the next three years.

Next post: how a number that looks off gets flagged — the checks that run before the report is even written, and how a flagged figure is shown without ever being reported as fact.

PART 5 OF 7

JUNE 18, 2026 PART 5 OF 7 · WEEKLY REPORT BUILDER SERIES ~5 MIN READ

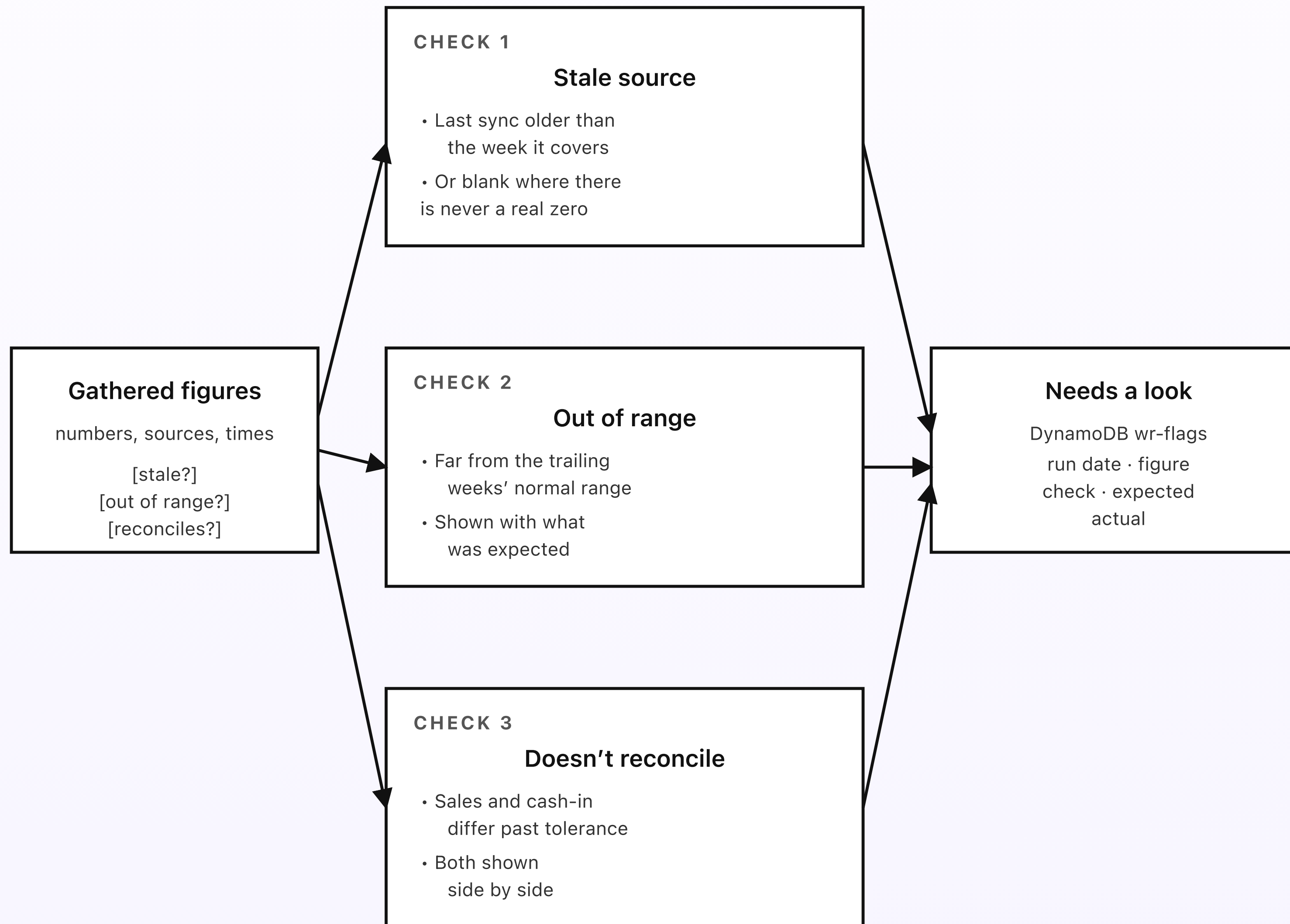
How a number that looks off gets flagged

The whole promise of the report is that the owner can trust it. That promise only holds if the builder is honest about the numbers it isn't sure of. A blank source read as zero, a figure ten times its normal size because a tool double-counted, a sales total that doesn't match the cash that came in — if any of those gets reported as a clean fact, the report has lied. This post walks through the three kinds of check that catch those before the report is written, and what happens to a number that trips one.

KEY TAKEAWAYS

- The checks run on the gathered figures *before* the report is written, all in plain Python.
- Three kinds of flag: a stale source, a figure out of its normal range, and a total that doesn't reconcile.
- A flagged figure is shown but clearly marked unverified — it never appears as a plain fact.
- Everything flagged is collected into a *Needs a look* section at the top of the report.
- Every flag is written to `wr-flags` so you can see how often a source misbehaves.

Three kinds of flag



A flagged number is shown but never stated as fact — the owner decides, the builder never guesses.

Fig 5. Three kinds of check, three ways a figure can look off. A stale source, a figure out of its normal range, and two numbers that don't reconcile. Each flag is shown in the report, written to `wr-flags`, and never stated as plain fact.

Check 1: a source that didn't update (the most common)

The quietest failure is a source that simply didn't refresh. The bank export didn't land this week, so the cash figures are last week's. The point-of-sale was offline Saturday, so the weekend is missing. The danger isn't a loud error — it's that a missing source reads as a zero or a stale repeat, and a zero looks like a real number. Check 1 compares each source's last sync time against the week the report covers. If a source hasn't synced through the end of the week, or a figure comes back blank where this business never has a true zero (a shop that always sells something can't have a $0 sales day), the figure is marked unverified and the source is named in the flag.

This is the check that pairs with the completeness gate from Part 4. The gate tries to hold the send until everything is in; if the data still isn't complete by the last retry, this check makes sure the partial figure is labelled rather than reported straight.

Check 2: a figure far outside its normal range

Some bad numbers are present, not missing — they're just wrong. A tool double-counts a batch and sales come back at ten times a normal week. A currency column gets read in cents and every figure is off by a hundred. Check 2 catches

these by comparing each figure against its own recent history: the trailing several weeks set a normal range, and anything more than the configured number of standard deviations outside it — or a multiple of last week that no real week ever reaches — trips the flag. The figure is still shown, but next to it the report says what was expected: “$184,000 this week — the four-week average is $18,000; this looks off, please check the source.”

The threshold is in the config doc, so a business with genuinely spiky weeks (seasonal, event-driven) can widen the range and not get flagged every busy week. The goal is to catch the impossible, not to second-guess a real good week.

Check 3: numbers that should agree but don’t

The strongest check is one number against another. Sales recorded in the sales sheet should roughly match the cash that came in through Stripe and the bank. The parts of a total — sales by category, or by location — should sum to the total. Check 3 compares the pairs that ought to agree and flags any gap bigger than a small tolerance. When sales say $20,000 but cash-in says $14,000, that’s not necessarily an error — it could be timing, refunds, or an unpaid invoice — but it’s exactly the kind of thing the owner should see. Both figures are shown side by side with the gap named, and the builder makes no attempt to pick which one is “right.”

Reconciliation is the check that most often surfaces something real rather than a glitch — a refund batch nobody logged, an invoice that was sent but not paid, a category that stopped reporting. It’s the difference between a report that lists numbers and one that notices when they disagree.

What a flag does to the report

Every flag from all three checks is collected into a *Needs a look* section that sits at the very top of the report, above the summary. Each entry names the figure, the check that tripped, what was expected, and what was actually there. The flagged figure still appears in the numbers table below, but marked — a small “unverified” tag — so it’s never mistaken for a clean fact. Crucially, a flagged figure is also withheld from the writer’s facts list in Part 3: the model is never handed a number that failed a check, so the summary paragraph can’t accidentally state a bad figure as if the week really went that way.

Every flag is also written to the `wr-flags` DynamoDB table with the run date, the figure, the check, the expected value, and the actual value. Over a few months that table tells its own story — the bank export that’s late one Monday in three, the point-of-sale that drops every long weekend. A source that keeps tripping the same flag is a source to fix at the source, and the table is the evidence for that conversation.

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 one model call a week is the biggest single line.

PART 6 OF 7

JUNE 18, 2026 PART 6 OF 7 · WEEKLY REPORT BUILDER SERIES ~3 MIN READ

What the weekly report builder costs

The builder is one of the cheapest systems in this whole series. The weekly run reads a few CSVs from S3, does some arithmetic, writes a handful of rows to DynamoDB, makes one Bedrock call, and sends one email. Almost everything is plain Python and costs pennies. The one model call a week is the biggest single line on the bill, and even that is a fraction of a dollar. At typical SMB volume, the total is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (a handful of sources, a few thousand rows a week).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The one Bedrock call a week is the biggest single line — and it's still under a dollar a month.
- Gathering, comparing, and the checks are all plain Python — pennies.
- At a mid setup the bill is around $5. At ten sources and heavy volume it's around $9.

Cost at three volumes

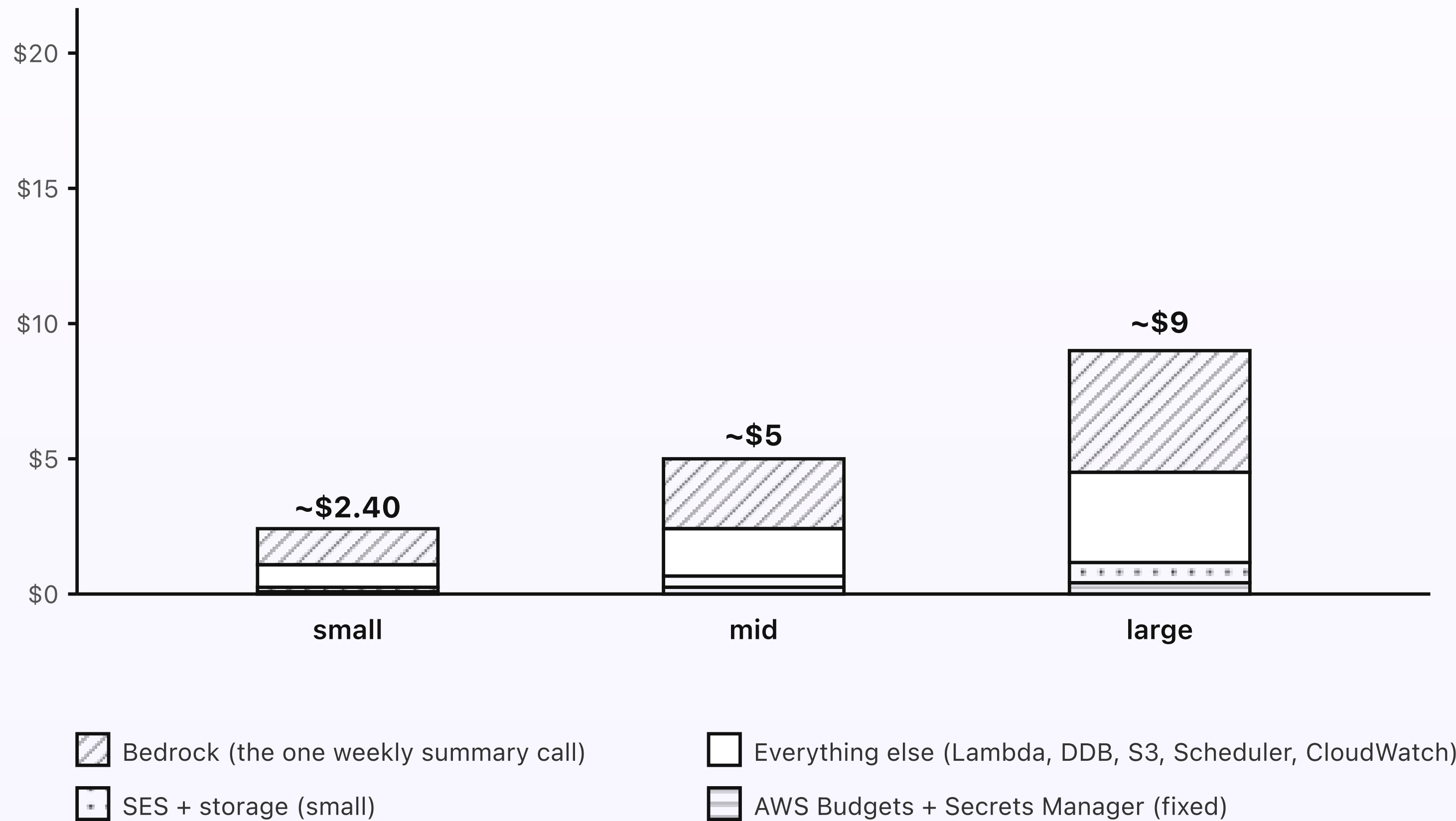


Fig 6. Monthly cost at three setup sizes. Bedrock — the one weekly summary call — is the largest single slice, because each run is one model call no matter how many rows were gathered. Everything else grows slowly with data volume but stays under it.

Where the dollars actually go

Bedrock (the biggest single line). The builder makes exactly one model call a week: hand Claude Haiku 4.5 the computed facts list, get back a short paragraph. That's a few thousand input tokens and a few hundred output tokens, so a fraction of a cent per run — but because almost nothing else costs anything, it's still the largest slice of the bill. Four or five calls a month, well under a dollar. The call is the same size whether you gathered 200 rows or 20,000, because the model only ever sees the small facts list, never the raw data.

Lambda runtime. The weekly run reads the source CSVs from S3, computes the figures and comparisons, runs the checks, and sends the email. At a few sources that's a second or two of compute, once a week. Add the `source-sync` Lambda mirroring each source on its schedule and the Function URL Lambda for the occasional full-table lookup — the Lambda total still lands in pennies at all three sizes.

DynamoDB on-demand. Two small tables: `wr-runs` (one row per weekly send) and `wr-flags` (one row per flagged figure). A handful of writes a week, a handful of reads. Pennies a month at any size.

S3 + storage. The mirrored source CSVs plus the archived report HTML for each week. A few hundred KB to a few MB total at SMB volume. Effectively free.

EventBridge Scheduler. The weekly run rule plus any one-off retry rules from the completeness gate, plus the `source-sync` schedules. A few invocations a week. Pennies.

SES. Outbound for the report email: $0.10 per thousand sent. One email a week to a handful of readers is a couple of cents a year. Negligible.

What doesn’t cost money

- **API Gateway.** Replaced by a Lambda Function URL for the full-table lookup link.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The builder runs once a week and sleeps the rest.
- **A Knowledge Base.** The numbers are structured rows, not free text — plain arithmetic beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the hot path.** Gathering, comparing, and the checks are plain Python. Bedrock fires once, at the end, to write the paragraph — never to source a number.

How the cost scales

Lambda runtime and DynamoDB grow slowly with the number of sources and the weekly row count, because more data means a slightly longer run. But Bedrock — the biggest line — is flat: one call a week regardless of volume, because the model only ever sees the small facts list. So the bill barely moves as you add sources. A business with twenty sources and very heavy weekly volume lands around $14; past that you’d split the gather step to read sources in parallel, but that’s a tuning detail, not a redesign.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The builder's normal-volume bill stays well under that ceiling.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, DynamoDB schemas, and EventBridge Scheduler config.

PART 7 OF 7

JUNE 18, 2026 PART 7 OF 7 · WEEKLY REPORT BUILDER SERIES ~8 MIN READ

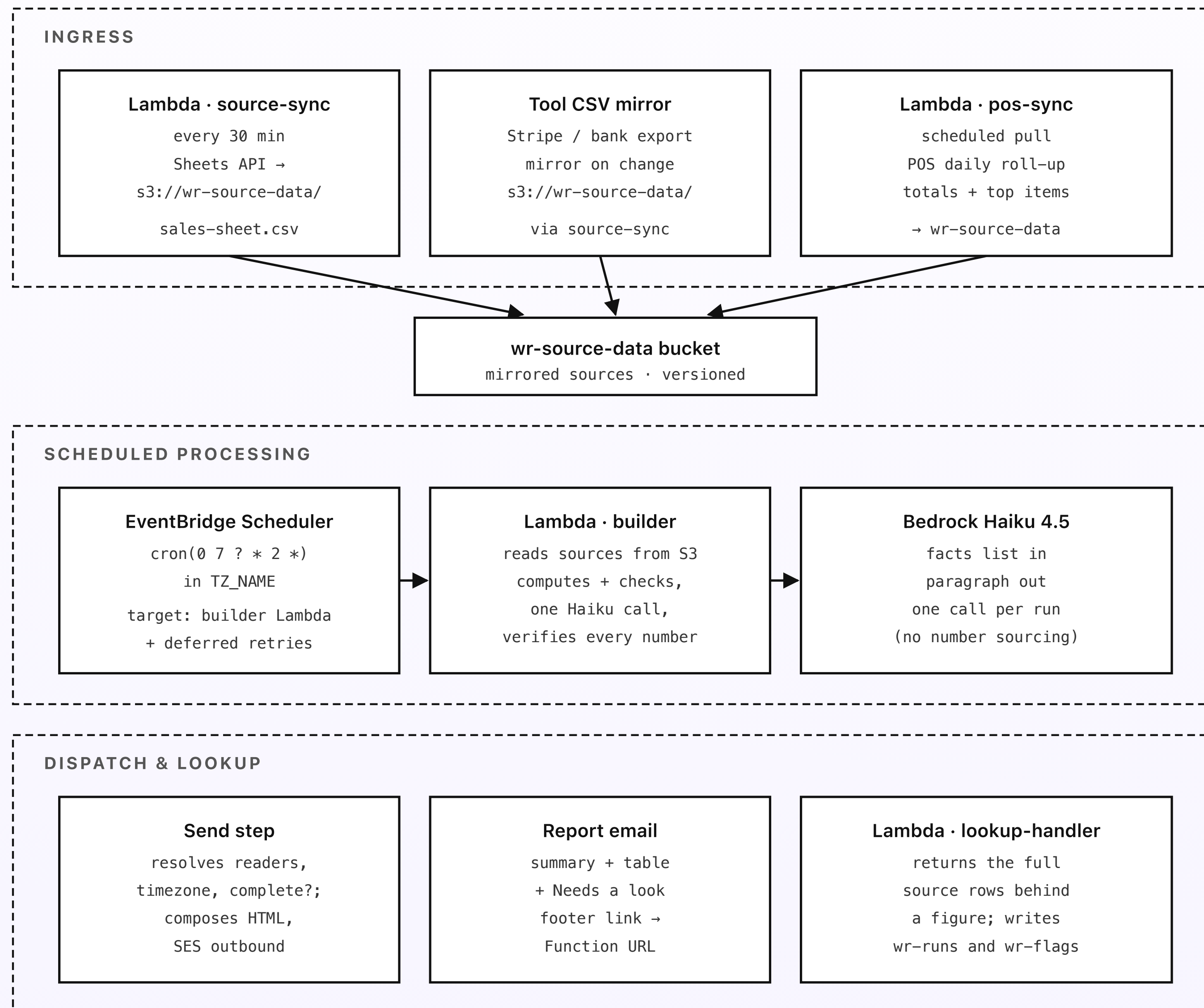
Engineering reference: the weekly report builder architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, EventBridge Scheduler config, the DynamoDB schemas, and the grounding contract that keeps the model from ever sourcing a number. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). Bedrock cross-Region inference, SES outbound, 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 report landing an hour late, not a regional outage. One AWS account dedicated to the builder (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every number in the report comes from the data — and every run is logged to wr-runs.

Fig 7. AWS topology, in three regions of the diagram: ingress (three source kinds into one bucket), scheduled processing (the weekly builder run computing, checking, and writing), dispatch and lookup (the report ships and the full rows stay one link away). Every Lambda is schedule- or request-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.

- **`source-sync`** — EventBridge Scheduler target, fires every 30 minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `wr/google/sa`) to export the hand-kept registry sheet as CSV and mirror any tool CSVs in the configured Drive folder to `s3://wr-source-data/`, writing only if a source has changed since the last sync. The same pattern syncs the config and voice docs to `s3://wr-config-source/`. Memory: 256 MB. Timeout: 30 s.
- **`pos-sync`** — EventBridge Scheduler target, scheduled per the POS's roll-up cadence (typically nightly plus a morning catch-up). Pulls the point-of-sale daily summary from wherever it lands (an SFTP drop, a vendor API, or a Drive file) and writes it to `s3://wr-source-data/pos/`. Kept separate from `source-sync` because POS integrations vary the most and benefit from their own retry and timeout tuning. Memory: 256 MB. Timeout: 60 s.

- **`builder`** — EventBridge Scheduler target, weekly Monday 7am in the owner's timezone (the schedule expression runs in `TZ_NAME`, e.g. `Asia/Singapore`). Reads every source from `s3://wr-source-data/` and the config and voice docs. Normalizes into one figure set; computes this week, last week, and the four-week average per figure; runs the three look-off checks; builds the facts list (flagged figures withheld); calls Bedrock Haiku 4.5 once for the summary; verifies every number in the draft against the figure set, dropping any unmatched sentence; assembles the report. Hands the assembled report to the send step in the same invocation. Memory: 512 MB. Timeout: 120 s. *Exactly one Bedrock call per run.*
- **send step** — runs inside the `builder` invocation (not a separate function) once the report is assembled. Resolves recipients from the config doc, confirms the owner timezone from Parameter Store, runs the completeness check, composes the HTML email, and ships via SES `SendRawEmail` from the verified sender identity. On an incomplete week, instead of sending it creates a one-off EventBridge Scheduler rule that re-invokes `builder` in retry mode a couple of hours later. Writes a row to `wr-runs` after a successful send and any flags to `wr-flags`.
- **`lookup-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a short-lived signed token embedded in the email footer link. Triggered when the owner clicks "show the rows behind this figure." Reads the relevant source slice from `s3://wr-source-data/` for the reported week and returns it as a simple HTML table. Read-only; writes nothing. Memory: 256 MB. Timeout: 15 s.

Storage

- **DynamoDB** · `wr-runs` — one row per weekly send. PK `(owner_id, week_start)`; attributes: `sent_at`, `recipients`, `figures` (the reported figure set as a map), `summary_text`, `dropped_sentences` count. On-demand. No TTL — this is what next week's comparison reads.
- **DynamoDB** · `wr-flags` — one row per flagged figure. PK `(owner_id, week_start)`; sort key `figure_check`; attributes: `check` (stale/out_of_range/reconcile), `figure`, `expected`, `actual`, `source`. On-demand. No TTL — the long-term record of which sources misbehave.
- **S3** · `wr-source-data` — mirrored source CSVs and the POS roll-ups. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3** · `wr-config-source` — mirrored config and voice docs as plain text. Versioning enabled.
- **S3** · `wr-reports` — the assembled HTML report for each week, kept for reference and for the lookup link. Versioning enabled. Lifecycle to Glacier at 90 days.

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`. One callsite: `builder`, for the weekly summary paragraph. The heavier `claude-sonnet-4-6` isn't used — turning a short facts list into a paragraph is well within Haiku's range and doesn't justify the cost.

- **Grounding contract.** The prompt hands the model only the computed facts list (statements with numbers attached) and instructs it to introduce no figure not in that list. The output is then number-checked in code against the figure set; any sentence whose number doesn't match the set is dropped before send. The model never sees raw source data and never sources a number.
- **Embeddings.** Not used. The numbers are structured rows; deterministic arithmetic beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are far more than enough — one call a week per owner.

EventBridge Scheduler config

- **`wr-weekly-run`** — `cron(0 7 ? * 2 *)` (Monday 7am) in the owner's timezone. Target: `builder` Lambda.
- **`wr-source-sync`** — `rate(30 minutes)`. Target: `source-sync` Lambda.
- **`wr-pos-sync`** — per the POS cadence, e.g. `cron(30 1 * * ? *)` plus a morning catch-up. Target: `pos-sync` Lambda.
- **One-off retry rules** — created on the fly by the send step when the completeness check holds a send. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions in TZ with `--action-after-completion DELETE` so the rule self-cleans.

SES outbound

- Verify a sender identity at `reports@your-company.com` with DKIM and SPF on the parent domain so the weekly email lands in the inbox, not spam.

- The send step uses `SendRawEmail` so the report can be a full multipart HTML message with the numbers table inline.
- Out of the SES sandbox by request before go-live; the recipient list is small and static, so this is a one-time step.

IAM (least privilege per Lambda)

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

- **builder role:** `s3:GetObject` on `wr-source-data`, `wr-config-source`; `s3:PutObject` on `wr-reports`; `dynamodb:Query` + `GetItem` + `PutItem` on `wr-runs` and `wr-flags`; `bedrock:InvokeModel` on the Haiku ARN; `ses:SendRawEmail` from the verified sender; `scheduler:CreateSchedule` for retry one-offs; `ssm:GetParameter` on `/wr/config/*`.
- **source-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on `wr-source-data` and `wr-config-source`; outbound network to `www.googleapis.com`.
- **pos-sync role:** `secretsmanager:GetSecretValue` on the POS credential secret; `s3:PutObject` on `wr-source-data`; outbound network to the POS endpoint only.
- **lookup-handler role:** `s3:GetObject` on `wr-source-data` and `wr-reports`; `ssm:GetParameter` on the link-signing key. Read-only — no write permissions, no Bedrock, no SES.

The grounding flow, in code

The contract that makes the report trustworthy is enforced in three places, not one. First, the gather step is the only thing that ever computes a figure; everything downstream reads from its output, never from raw sources. Second, the facts list handed to Bedrock is a closed set — flagged figures are excluded, so the model can't even mention a number that failed a check. Third, the post-model number-check scans the draft, extracts every numeric token, and matches it against the figure set within a small rounding tolerance; an unmatched token drops its whole sentence and increments the `dropped_sentences` counter on the run.

The counter matters operationally: a run with a non-zero drop count is logged at `WARN`, and a sustained pattern of drops means the prompt or the model is drifting and should be reviewed. In steady state the count is zero — Haiku 4.5 handed a tight facts list and told to describe only it rarely strays — but the system is built so that "rarely" never reaches the owner as a wrong number.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` + `"dropped_sentences"` to a CloudWatch metric for alerting.
- **Alarms:** `builder` failures > 0 on a Monday (the weekly run is the one piece that has to succeed); send failures > 0; `dropped_sentences` > 0 for two consecutive weeks (the model may be drifting); a source that trips the stale check three weeks running (fix it at the source).
- **X-Ray:** off by default. Not worth the cost at SMB volume.

- **AWS Budgets:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `wr-cost-alarm` subscribed to the admin's email.

Config and secrets

Service-account credentials for the Drive, Sheets, and Calendar APIs live in Secrets Manager under `wr/google/sa` (one service account, read-only scopes). POS credentials live under `wr/pos/*`. The configured timezone, the recipient list, the notable-change thresholds, the look-off check thresholds, and the link-signing key all live in Parameter Store under `/wr/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment. Nothing in the system has write access to any source — every Google and POS scope is read-only, which is the single most important guardrail: the builder can never alter the numbers it reports on.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys), and AWS SAM for the stack. The opinionated bits: turn on S3 versioning for `wr-source-data` and `wr-config-source` so a bad Drive edit can be rolled back in one click; version the EventBridge Scheduler timezone setting so a CI rotation can't silently start running the weekly job in UTC; and keep every Google and POS credential scoped read-only so the builder is structurally incapable of writing to a source. Total deployable surface: around five Lambdas, two DDB tables, three S3 buckets, a handful of Scheduler rules, one verified SES identity, 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**.