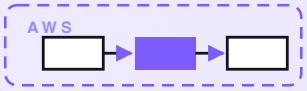
7-PART SERIES · FREE COMPANION



Sentiment monitor

A serverless monitor that watches the mentions you point it at — reviews, social posts, comments — reads the mood of each one, tracks whether the overall tone is rising or falling, and sends a short weekly pulse plus an instant alert if something starts going badly. It only listens and reports; it never posts replies, and it flags the angriest items for a human first. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Sentiment monitor

PART 1 OF 7

JUNE 1, 2026 PART 1 OF 7 · SENTIMENT MONITOR SERIES ~5 MIN READ

A sentiment monitor on AWS for a few dollars a month

A small business is talked about in more places than anyone keeps up with. The one-star review that landed at 11pm and is already showing on the search result. The comment thread on last week's post that quietly soured. The handful of replies under an ad that went from curious to annoyed. By the time someone notices, the mood has already moved — and the first you hear of it is a customer asking why everyone seems upset. This post walks through the design of a small monitor that watches the mentions you point it at, reads the mood of each, tracks whether tone is rising or falling, and tells you — clearly, and before it gets worse. It never replies. That part stays human.

KEY TAKEAWAYS

- Three sources for mentions: review-site feeds, a social-listening export, and your own comment webhooks.
- Every mention gets a mood score and a one-line reason from one cheap model call.
- A rolling trend tracks whether tone is rising or falling, in plain Python — no model on the math.
- Two instant-alert triggers: a single very angry mention, or a sharp drop in the average.
- It only listens and reports. It never posts a reply. Designed on AWS for about $2/month at SMB volume.

The whole system on one page

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

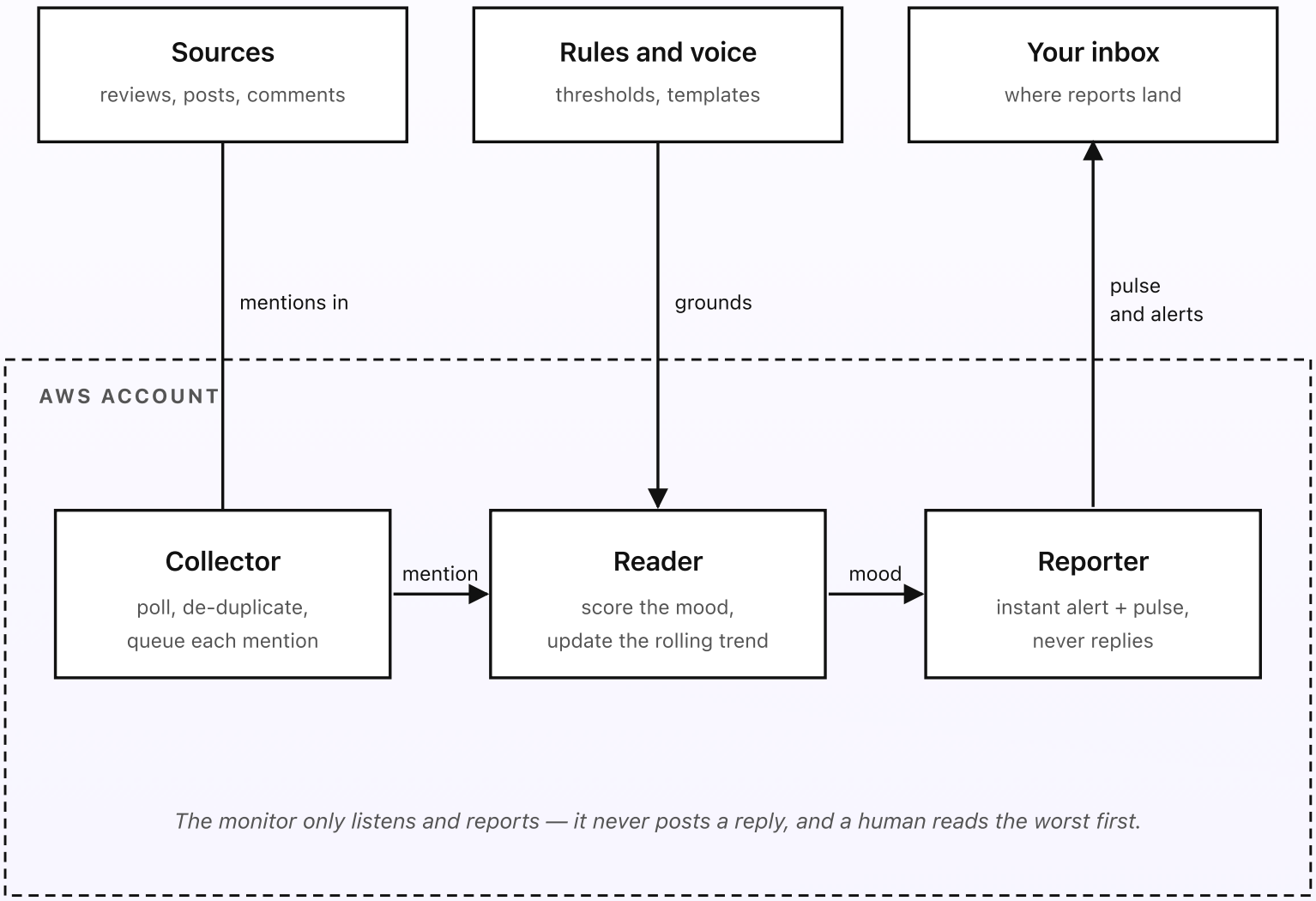

Sources
reviews, posts, comments
Rules and voice
thresholds, templates
Your inbox
where reports land
mentions in
grounds
pulse
and alerts
AWS ACCOUNT
Collector
poll, de-duplicate,
queue each mention
mention
Reader
score the mood,
update the rolling trend
mood
Reporter
instant alert + pulse,
never replies
The monitor only listens and reports — it never posts a reply, and a human reads the worst first.

Fig 1. Three sources outside, three pieces inside AWS. Mentions flow in from review feeds, a social-listening export, and comment webhooks. The Reader scores each mention's mood and updates the trend. The Reporter sends the weekly pulse and the instant alert — and never replies.

What you set up once (the outside)

- **Sources.** A short list of the places you want watched: a review-site feed for each platform you care about, an export from whatever social-listening tool you already pay for, and a webhook for the comments on your own site or app. Each source has a type and a way to reach it — a feed URL, an export bucket, or a webhook secret. You point the monitor at the handful that matter and ignore the rest; it only reads what you list. New sources can be added later without a deploy — covered in Part 2.
- **A rules folder.** Two short docs in a folder you control. The *rules* doc holds the numbers: the mood floor that makes a single mention an instant alert, the trend slope that makes a sharp drop an instant alert, the trailing window the average is measured over (default 14 days), and quiet hours so a routine alert doesn't ping at 3am. The *voice* doc holds the layout of the weekly pulse and the wording of the alert — what the email actually says. Both are plain text; a non-technical owner can edit either one.
- **Your inbox.** Where reports land. The weekly pulse is an email every Monday morning. The instant alert is an email (and an optional text) the moment a trigger fires, with the angriest mention quoted at the top and a link straight to the original so a human can decide what to do. There is no button that replies for you — on purpose.

What runs on each cycle (the inside)

- **The collector.** A poller checks each source on a schedule (every 15 minutes by default). For each source it pulls anything new, and de-duplicates — it keeps a record of every mention it has already seen, so the same review never gets read twice even if a feed re-lists it. Each genuinely new mention is dropped into a queue, which evens out bursts (a viral post can produce hundreds of comments in an hour) so the rest of the system isn't overwhelmed.
- **The reader.** Pulls mentions off the queue and reads the mood of each one with a single Bedrock Haiku 4.5 call. The call returns a mood score (how positive or negative, on a fixed scale) and a one-line reason ("upset about a late delivery"). The score is stored against the mention. Then plain Python updates a rolling average — the trend — so the system always knows whether tone over the trailing window is rising or falling. The model only labels mood; it never decides whether to alert, and it never writes a reply.
- **The reporter.** After each batch is read, plain Python checks the two alert triggers. If a single mention scored at or below the mood floor, or if the rolling average dropped faster than the configured slope, it fires an instant alert through SNS — an email, optionally a text — with the angriest item listed first. Separately, once a week it emails the pulse: the trend line, counts by source, the standout good and bad mentions, and what changed since last week. One Bedrock call writes the pulse's summary sentence; the rest is layout.

In plain words

On Thursday afternoon a delivery goes wrong and three customers say so — one in a review, two in comments under the same post. None is the worst review

you've ever had, but together they pull the trailing-14-day average down sharply in a few hours. The collector picks all three up within fifteen minutes of each appearing. The reader scores them — two clearly negative, one furious — and the rolling average drops past the slope you set. The reporter fires an instant alert to your phone: "Mood falling fast — 3 negative mentions in 4 hours, worst one quoted below, all about delivery." You read the furious one, see it's the same root cause, and handle it yourself — reply to the customer, fix the delivery, post an update. The monitor did none of that. It only made sure you knew while there was still time to act.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the slow week where the mood quietly turned and nobody saw it until a prospect mentioned "the reviews lately."

DESIGN RULES THAT SHAPED EVERY DECISION

- It only listens and reports. It never posts a reply or touches your public accounts. Replies stay human.
- The angriest item is always surfaced first, so a human reads the worst before deciding anything.
- One cheap model call per mention, for mood only. The trend math and the alert logic are plain Python.
- Two alert triggers, both deterministic: a single very angry mention, or a sharp drop in the average.
- Thresholds live in a doc you edit. Changing the mood floor or the trend slope doesn't need a deploy.
- Every mention and score is logged. Look back next quarter and you can see exactly what moved the trend.

Why this shape

Most owners track their reputation in one of three ways: a tab they open when they remember, an alert from one platform that misses the other five, or a gut feeling. The open-tab approach works until a busy week, when nobody opens it. The single-platform alert is loud about one place and silent about everywhere else. And the gut feeling is always a few days behind the actual mood — by the time it "feels off," the bad week already happened.

The setup above watches every source you name, reads the mood of each mention the same way, and turns that into one number you can trust: is tone rising or falling right now. It is quiet on the normal days. It speaks up the moment a single mention is furious or the average drops sharply. And it never, ever replies for you — because a wrong public reply from a bot is far worse than a slow one from a human. The monitor's whole job is to make sure the human is never the last to know.

The next four posts walk through each piece in turn: how a mention gets collected, how the mood of a mention gets read, how an angry mention reaches a human, and how the weekly pulse gets sent. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 1, 2026 PART 2 OF 7 · SENTIMENT MONITOR SERIES ~4 MIN READ

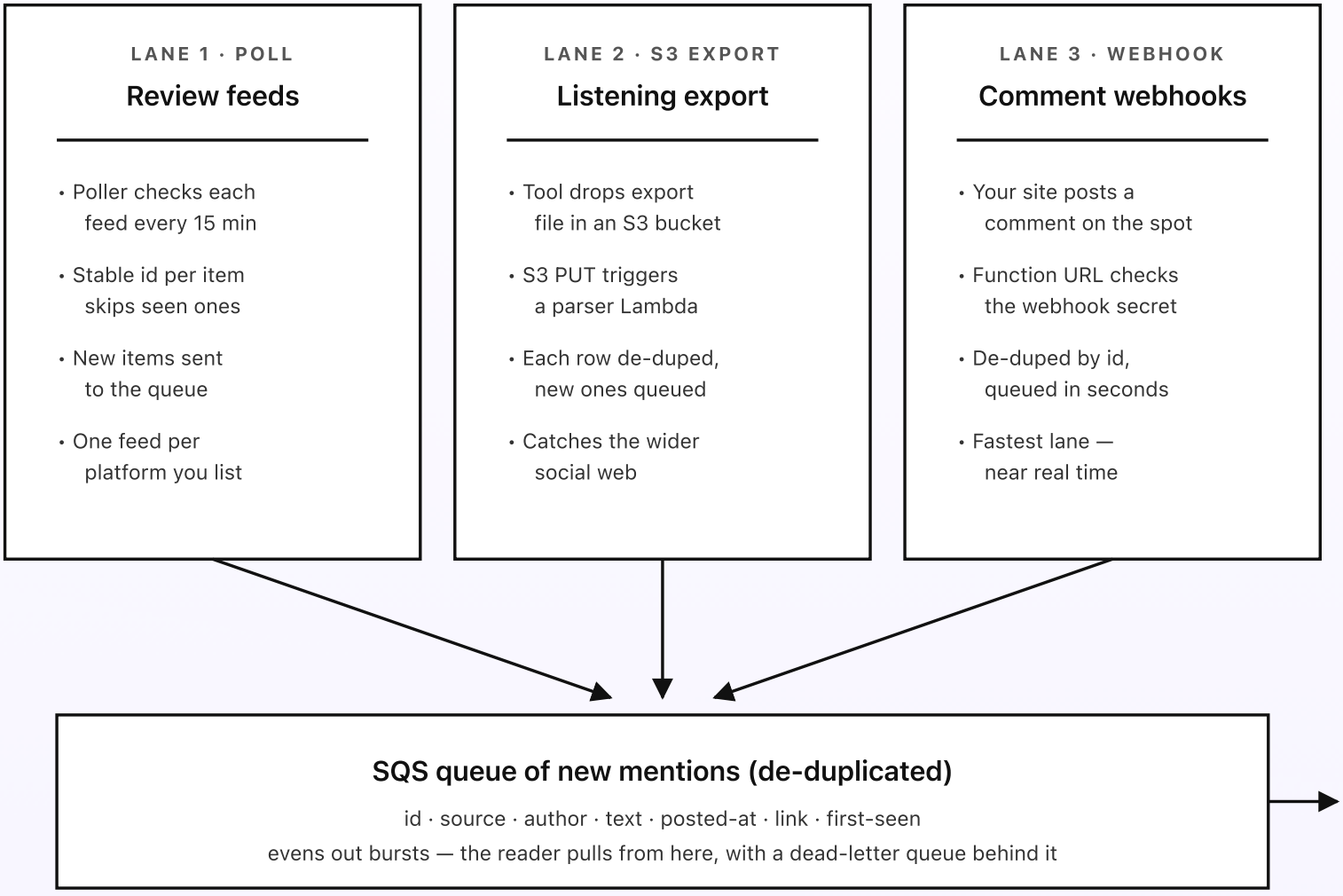
How a mention gets collected

The monitor only reads what it collects. So the first job is making sure it actually sees the places your business is talked about. There are three ways a mention gets in: a poller checks a review-site feed on a schedule, a social-listening tool drops its export in a bucket, or your own site sends a comment straight to a webhook the moment it’s posted. The three lanes look different, but they all end the same way — one clean, de-duplicated mention in a queue, waiting to be read.

KEY TAKEAWAYS

- Three intake lanes feed one queue: review feeds, a listening export, and comment webhooks.
- Each new mention is de-duplicated against a record of everything already seen.
- The queue evens out bursts — a viral post's comments don't overwhelm the reader.
- Webhook comments arrive in seconds; feed and export lanes are checked on a schedule.
- Adding or removing a source is a config-doc edit — no deploy needed.

Three lanes into one queue



Every mention is de-duplicated before the queue — the same review is never read twice.

Fig 2. Three lanes converge on one queue. Review feeds are polled, the listening export is parsed on arrival, and comment webhooks land in seconds. Each mention is de-duplicated first, then queued so the reader can pull them one at a time without being overwhelmed by a burst.

Lane 1: review-site feeds

The steadiest lane. For each review platform you care about, you give the monitor a feed — a URL it can read on a schedule. A small Lambda, the `poller`, runs every fifteen minutes (driven by EventBridge Scheduler), reads each feed, and looks at every item. For each one it computes a stable id from the platform and the review's own identifier, then checks that id against the `sm-seen` table. If the id is already there, the item is skipped — the monitor has seen it before. If it's new, the id is recorded and the mention is sent to the queue.

This lane covers the places that publish a feed and don't move fast: a new review every few hours at most for a typical small business. Fifteen-minute polling is plenty, and it keeps the number of reads predictable and cheap.

Lane 2: the social-listening export

Most owners already pay for a tool that watches the wider social web — posts and mentions that don't come with a tidy feed. Rather than rebuild that, the monitor reads its output. You point the tool at an S3 bucket (`sm-listening-export`) and have it drop a periodic export file there. The S3 PUT triggers a parser Lambda that reads each row of the file, computes a stable id the same way, de-duplicates against `sm-seen`, and queues anything new.

This lane is how the monitor sees beyond the platforms with feeds — the off-hand post on someone's own account, the mention in a community group, the reply under an ad. The de-duplication matters most here, because listening tools often re-list the same mention across several exports. The seen-table makes sure each one is read once and only once, no matter how many times it shows up in the file.

Lane 3: comment webhooks

The fastest lane, for the comments you own outright. When someone posts a comment on your own site or app, your code sends it straight to a Lambda Function URL (a plain web address that runs a small function — no heavier gateway in front of it). The handler checks a shared secret so only your site can post, computes the id, de-duplicates, and queues the mention. From comment posted to mention queued is a couple of seconds.

This is the lane that catches a thread souring in real time. A review feed checked every fifteen minutes is fine for a slow trickle; a comment section turning hostile during a launch is exactly when you want the mention in the queue before the next refresh. Webhooks give you that, and they cost nothing to keep open because nothing runs until a comment actually arrives.

Why everything passes through one queue

Three lanes in, but only one queue out. That's deliberate. If each lane fed the reader directly, a viral post producing four hundred comments in an hour would hit the reader four hundred times at once — and the reader makes a model call per mention, so that's a spike in both load and cost at the worst possible moment. The queue absorbs the burst: lanes drop mentions in as fast as they arrive, and the

reader pulls them out at a steady pace it can handle. Behind the queue sits a dead-letter queue — a holding pen for any mention that fails to process a few times — so one malformed item never blocks the rest or gets silently lost.

One queue also means one place to reason about. Every “why was this read?” question has a single answer: it was in the queue, and it was in the queue because exactly one lane put it there, once. Adding a new source — another review feed, another webhook — is an edit to the config doc, not a deploy.

Next post: how the reader takes a mention off the queue and reads its mood with a single model call, and how that score becomes a trend.

PART 3 OF 7

JUNE 1, 2026 PART 3 OF 7 · SENTIMENT MONITOR SERIES ~5 MIN READ

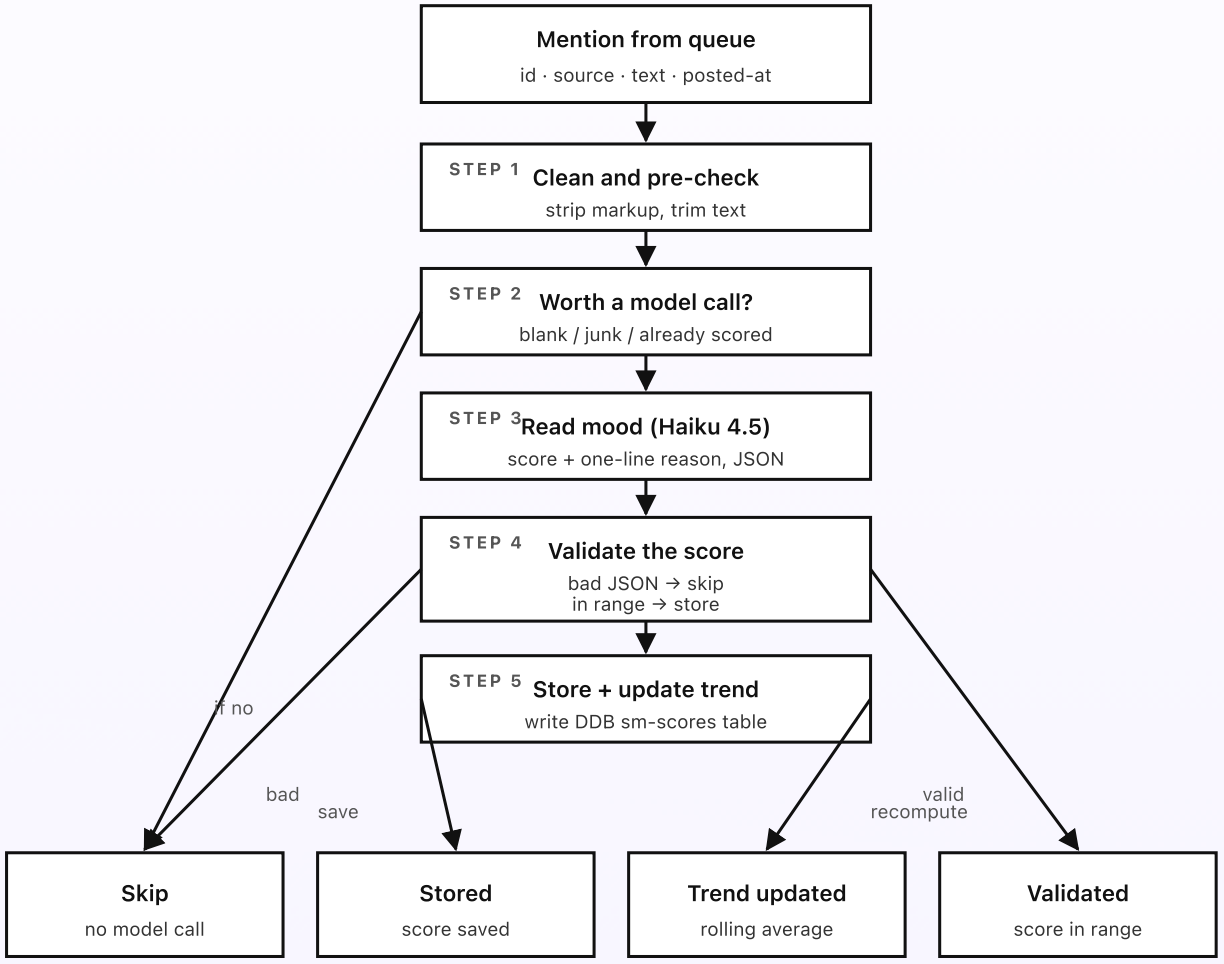
How the mood of a mention gets read

A mention comes off the queue: a review, a post, a comment. The reader has one job — turn that bit of text into a number that says how positive or negative it is, plus a one-line reason. That single step is the only place in the whole system a model is used. Everything after it — the trend, the alerts, the weekly pulse — is plain Python on top of the stored scores. The model labels the mood; it never decides what to do about it.

KEY TAKEAWAYS

- One Bedrock Haiku 4.5 call per mention returns a mood score and a one-line reason.
- The score is on a fixed scale, so every mention is measured the same way.
- Scores are stored in DynamoDB; the rolling trend is computed from them in plain Python.
- A short, blank or junk mention is skipped before the model call, to save cost.
- The model only labels mood. It never decides whether to alert and never writes a reply.

The read flow, per mention



The model only labels mood — the trend and the alerts are plain Python on the stored scores.

Fig 3. The reader's flow, per mention. Clean the text, skip what isn't worth a call, read the mood once with Haiku 4.5, validate, store the score, and recompute the rolling trend. The model labels mood; everything downstream is deterministic.

The mood score: one scale for everything

The model is asked for a single number on a fixed scale — say, −2 for furious, −1 for unhappy, 0 for neutral, +1 for pleased, +2 for delighted — plus a one-line reason in plain words. The fixed scale is the whole point. A five-star review and an off-hand comment are wildly different in shape, but once each becomes a number on the same scale, they can be averaged, compared, and trended together. A review platform's own star rating doesn't help here, because half your sources don't have stars at all — the comment under a post has no stars, just words. Reading every mention into the same scale is what lets the monitor treat them as one stream.

The prompt is short and strict: "Read this mention. Return JSON only: a mood score from −2 to +2 and a one-line reason. Do not guess at facts you can't see. If the text is unclear, score it 0 and say so." Asking for JSON only, with a tight range, keeps the output something plain Python can check and store without surprises.

A cheap model, and only when it's worth it

Reading mood is exactly the kind of small, well-defined job a fast, inexpensive model does well, so the reader uses Claude Haiku 4.5 — the cheap path. There's no need for heavier reasoning here; the task is "how does this feel," not "reason

through a contract." Each call is a few hundred tokens in and a few tokens out, so it costs a fraction of a cent.

Before any call, Step 2 checks whether the call is worth making. A blank comment, a one-word "ok," a mention that's already been scored, or obvious junk gets skipped with no model call at all. Most mentions are worth reading; the pre-check just stops the system from paying for the ones that aren't. It's a small saving per mention that adds up across a busy month, and it keeps the model focused on text that actually carries a mood.

Store the score, then compute the trend

A valid score is written to the `sm-scores` table in DynamoDB against the mention's id, with the source, the posted-at time, the score, and the reason. That stored row is the system's memory. Everything else reads from it.

The trend is plain Python on top of those rows. After each batch, the reader recomputes a rolling average over the trailing window (default 14 days) — one number that says how the mood has felt lately — and compares it to the same number from before this batch. The difference is the slope: rising, flat, or falling. No model touches this step. The trend is just arithmetic over the stored scores, which means it's fast, free, and gives the same answer every time for the same data. Part 4 uses the slope and the individual scores to decide when a human needs to hear about it.

Why the model is used only here

The model earns its place reading mood, because reading tone from free text is genuinely hard to do with rules — sarcasm, slang, and context defeat keyword lists. But the moment the mood is a number, rules win. Deciding whether a score is below the floor, whether the average dropped too fast, what to put in the weekly email — all of that is clearer, cheaper, and more predictable as plain Python. Keeping the model to one job also means the part of the system you can't fully predict is small and contained: one labeled number per mention, validated before it's trusted, and never allowed to decide an action on its own.

Next post: how an angry mention — a single furious score, or a sharp drop in the trend — reaches a human, worst item first, with the original a click away.

PART 4 OF 7

JUNE 1, 2026 PART 4 OF 7 · SENTIMENT MONITOR SERIES ~5 MIN READ

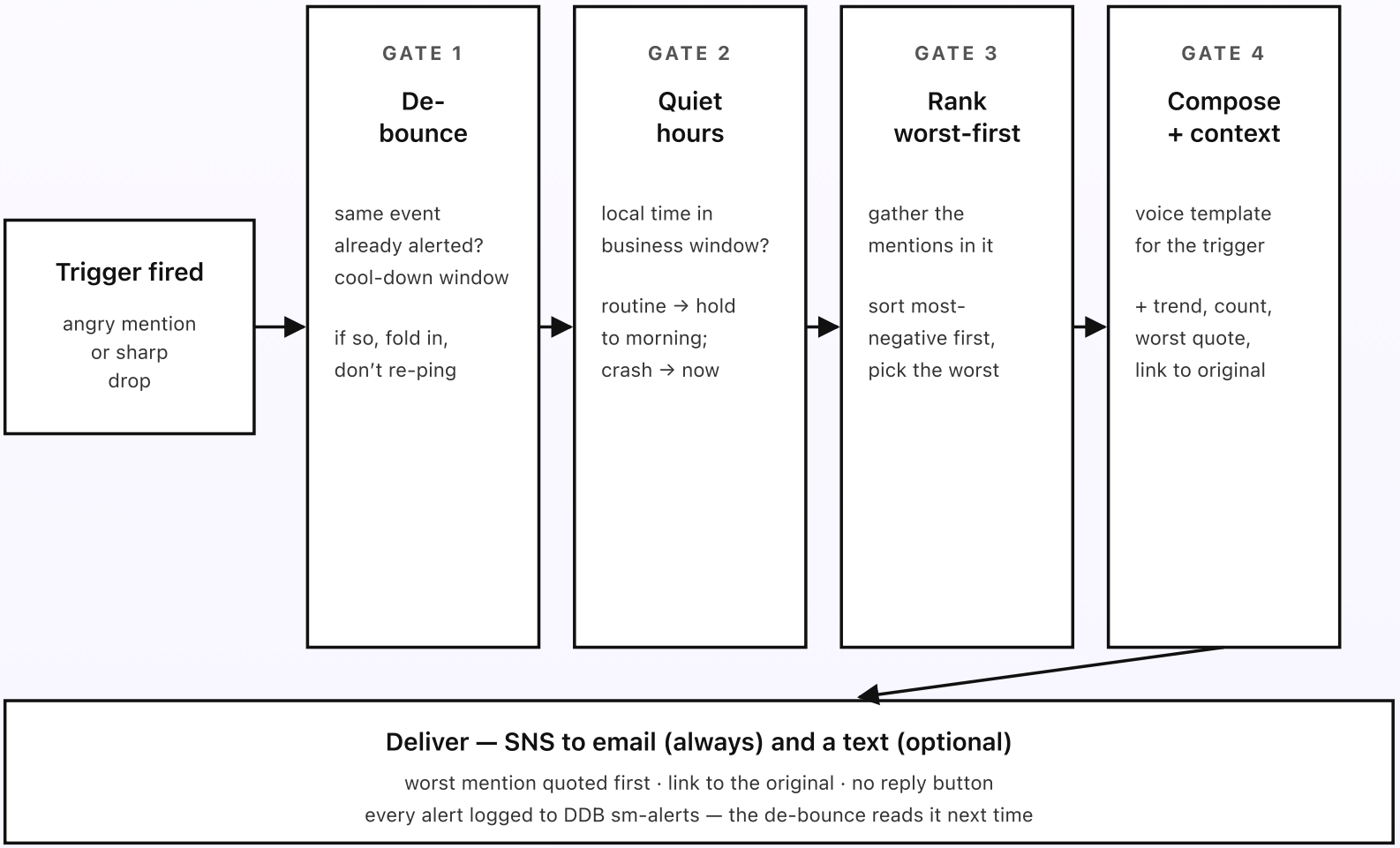
How an angry mention reaches a human

The reader stored a score and updated the trend. Now the reporter has to decide: does anyone need to hear about this right now? Get it wrong in one direction and a furious customer sits unseen for a day. Get it wrong in the other and you ping the owner's phone for every mildly grumpy comment until they mute you. Two triggers decide when an alert fires, and four small guardrails sit between the trigger and the actual ping landing — so the alert is loud when it should be and silent when it shouldn't.

KEY TAKEAWAYS

- Two triggers: a single mention at or below the mood floor, or a sharp drop in the average.
- Both triggers are plain Python over the stored scores — no model decides whether to alert.
- A de-bounce stops the same bad event from firing ten alerts in an hour.
- Quiet hours hold a non-urgent alert to morning; a true crash still goes straight through.
- Every alert lists the angriest item first, with a link to the original. The monitor never replies.

Four guardrails on every alert



Every gate is a deterministic check — and the alert ends at a human's inbox, never a reply.

Fig 4. Four guardrails between a trigger and the delivered alert. De-bounce repeated firings. Hold routine alerts for quiet hours but let a crash through. Rank the mentions worst-first. Compose with full context. Then ship via SNS — and stop there, at a human.

The two triggers

An alert fires on one of two conditions, and both are plain Python over the stored scores. The first is the **single angry mention**: any one mention that scores at or below the mood floor set in the rules doc (default: the most-negative band). One furious review can matter on its own, even on a day when everything else is fine — so it gets its own trigger, no averaging required.

The second is the **sharp drop**: the rolling average falling faster than the configured slope over the trailing window. This is the one that catches the quiet bad week — no single mention is extreme, but five mildly negative ones in two days pull the average down hard. The slope trigger sees the shape of the change even when no individual mention would trip the floor. Together the two cover both failure modes: the loud one and the slow one. The model is nowhere in this decision; it only produced the scores the math runs on.

Gate 1: de-bounce

One bad event tends to produce many mentions close together. Without a guard, each one re-trips the trigger and fires another alert — ten pings for one problem, which trains the owner to ignore all of them. Gate 1 checks the `sm-alerts` table: has an alert for this same event already gone out inside the cool-down window (default a few hours)? If yes, the new firing is folded into the existing alert — the

count goes up, a fresh quote may be added — but no second ping is sent. The owner gets one alert that grows, not a stream of duplicates.

Gate 2: quiet hours

The rules doc has a quiet-hours window (default 9pm to 7am). A routine alert that fires inside that window is held: the reporter creates a one-off EventBridge Scheduler rule that re-runs the dispatch at the next morning's start, and exits without sending. The owner wakes to it instead of being woken by it.

There's one exception, and it's important. A true crash — a mention scored far below the floor, or a very steep drop — is marked urgent and bypasses the hold entirely. The judgment here is simple: a normal dip can wait until morning; a customer publicly melting down at midnight, or a sudden cliff in the average, is exactly the thing you'd want to be woken for. The threshold for "urgent" is in the rules doc too, so you tune how easily the monitor is allowed to break quiet hours.

Gate 3: rank worst-first

Before composing anything, Gate 3 gathers the mentions behind the trigger — the one furious mention, or all the negative ones inside the window that dragged the average down — and sorts them most-negative first. The single angriest one is picked to quote at the top of the alert. This is the human-in-the-loop part made concrete: the worst item is always the first thing the reader sees, so the person deciding what to do reads the genuine low point before skimming the rest. No alert buries the worst mention three lines down.

Gate 4: compose with full context, then ship

The voice doc has a template per trigger type. The reporter fills it: which way the trend is moving, how many negative mentions are involved, the angriest quote, its one-line reason, and a link straight to the original so a human can go read it in context. The filled alert ships through SNS — an email always, and an optional text for the urgent ones. The email has no reply button, no “respond” action, nothing that touches your public accounts. It ends at the inbox.

Every alert sent writes a row to `sm-alerts` with the event, the time, and what was quoted. The next firing’s de-bounce reads that row. That’s the loop that keeps one event to one growing alert.

Why it always stops at a human

The firmest rule in the whole system lives in this post: the monitor reports, it does not reply. It would be easy to add a button that posts a draft response, and tempting — but a wrong public reply from an automated system, sent at the worst possible moment to the angriest possible customer, is exactly the kind of damage the monitor exists to prevent. So the design draws a hard line. The monitor’s job is to put the worst mention in front of the right person, fast, with enough context to act — and then get out of the way. The reply is a human’s, every time.

Next post: the weekly pulse — the calm, scheduled counterpart to the instant alert, summarizing the whole week’s mood in one short email.

PART 5 OF 7

JUNE 1, 2026 PART 5 OF 7 · SENTIMENT MONITOR SERIES ~5 MIN READ

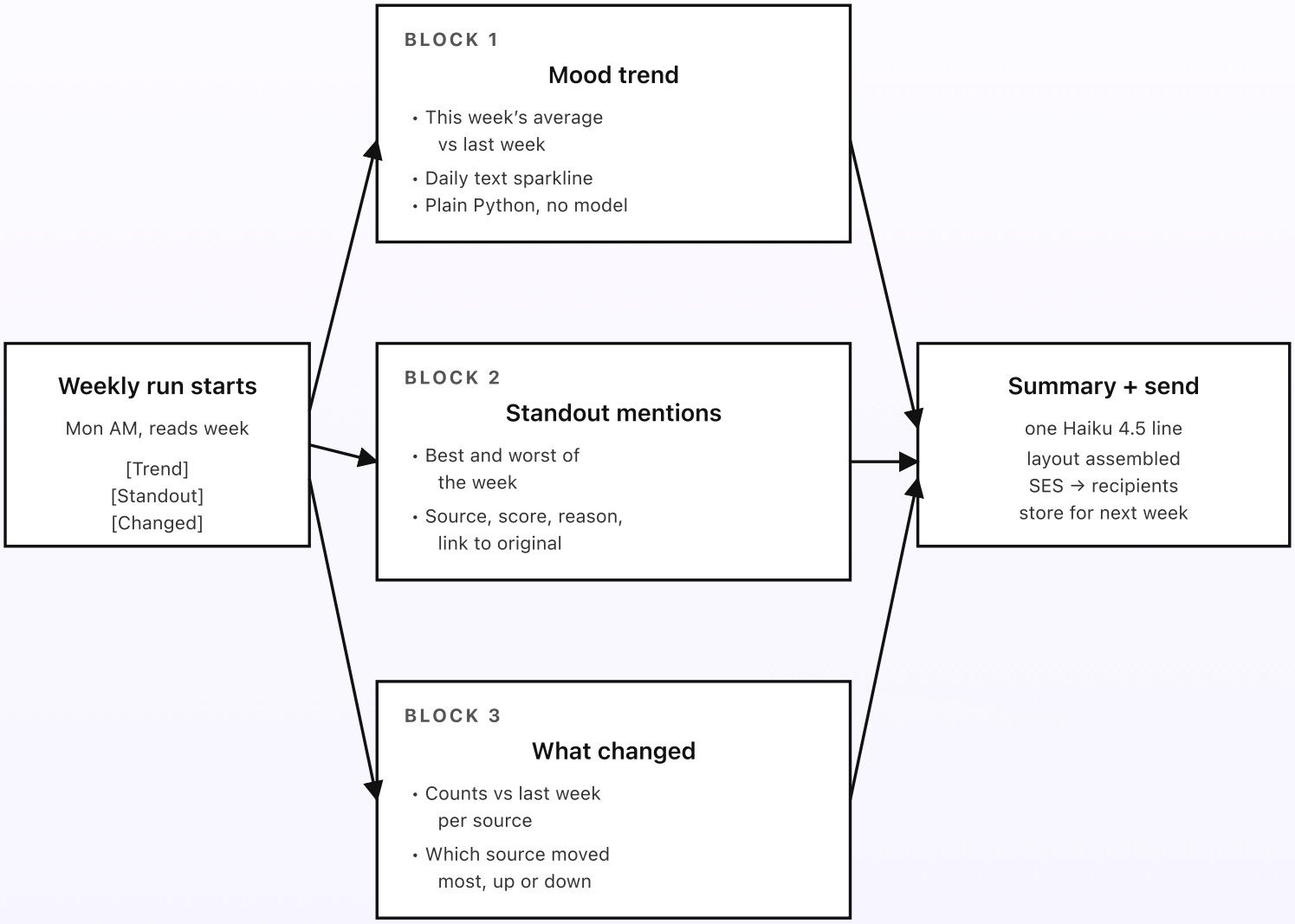
How the weekly pulse gets sent

The instant alert is for the bad moment. The weekly pulse is for the calm overview — the Monday-morning email that tells you how the week actually went, even if nothing ever crossed an alert threshold. It's short on purpose: three blocks an owner can read in thirty seconds. This post walks through what goes in each block, how the pulse is built almost entirely from plain arithmetic over the stored scores, and the single Bedrock call that turns those numbers into one human sentence.

KEY TAKEAWAYS

- Three blocks: the mood trend, the standout mentions, and what changed since last week.
- Built almost entirely from plain arithmetic over the stored scores — no model on the numbers.
- One Bedrock Haiku 4.5 call writes the human summary line at the top.
- EventBridge Scheduler fires it every Monday morning; SES sends the email.
- The pulse always goes out, even on a quiet week — absence of an alert is information too.

Three blocks in the pulse



Only the summary line uses a model — every number is plain arithmetic over the stored scores.

Fig 5. Three blocks assembled into one email. The trend, the standout mentions, and what changed are all built from the stored scores in plain Python. A single Bedrock call writes the human summary line, then SES sends it and this week's figures are stored for next week.

Block 1: the mood trend

The first block is the one most owners read and stop. It's the week's average mood as a single number, the direction compared to last week (up, down, or flat), and a tiny text sparkline — seven small marks, one per day, showing the daily average so the shape of the week is visible at a glance. A week that started fine and dipped on Thursday looks different from a week that was steadily low, and the sparkline shows that without anyone reading a single mention.

None of this needs a model. It's the same rolling-average math from Part 3, run once over the week's rows in the `sm-scores` table. Plain Python reads the scores, groups them by day, averages each day, and compares the week's overall figure to the one stored last Monday. The numbers are exact and reproducible, which is exactly what you want from the headline a business decision might lean on.

Block 2: the standout mentions

Numbers tell you the shape; the standout block tells you the why. Python sorts the week's scores and pulls two mentions: the single most positive and the single most negative. Each is shown with its source, its score, its one-line reason from the reader, and a link to the original. The best one is there because good news is worth seeing — a glowing review is a quote you might want to use, and a sign of

what's working. The worst one is there so that even on a week that never tripped an instant alert, the lowest point still gets a set of eyes on it.

Two mentions, not twenty. The pulse isn't a feed of everything — that's what the stored data is for if someone wants to dig. The block deliberately surfaces just the extremes, because those are the two an owner can actually act on in the thirty seconds they'll give the email.

Block 3: what changed

The third block is the comparison block, and it's why last week's figures are stored. Python compares this week to last: more or fewer mentions overall, the average up or down by how much, and which source moved the most. "Review mentions up 40% this week, mostly positive; comment mood down slightly, driven by one post." This block turns a single snapshot into a direction. A flat average can hide a source quietly climbing while another slips; the per-source deltas surface that.

This is the block that makes the pulse worth reading every week rather than once. The trend number alone can sit in the same place for a month while the underlying mix shifts; "what changed" is where that shift shows up. And like the other two blocks, it's pure arithmetic — subtract last week's stored summary from this week's computed one.

The one model call, and the send

At the top of the email sits one plain-English sentence: “A solid week — mood ticked up, your delivery fixes are showing in the reviews, and the one rough comment was an isolated billing question.” That sentence is the only place the pulse uses a model. The reporter hands the three assembled blocks to a single Bedrock Haiku 4.5 call and asks for one honest summary line — no spin, no padding, just what the numbers say in words. It’s the cheap path doing a small writing job, and it never invents a figure, because the figures are already fixed by the blocks above it.

Then the layout is assembled and SES sends it to the configured recipients. EventBridge Scheduler fires the whole run every Monday morning. Crucially, the pulse goes out even on a dead-quiet week with no alerts and no drama — because “nothing happened” is itself a thing an owner wants confirmed, not a silence they have to wonder about. This week’s figures are written to the summary store, and next Monday the cycle compares against them.

Next post: the cost breakdown. The whole monitor — collecting, reading, alerting, and this weekly pulse — runs in coffee-money territory at SMB volume; Part 6 shows exactly where the dollars go.

PART 6 OF 7

JUNE 1, 2026 PART 6 OF 7 · SENTIMENT MONITOR SERIES ~3 MIN READ

What the sentiment monitor costs

The monitor is one of the cheaper systems in this series. The collector polls a few feeds and queues new mentions. The reader makes one cheap model call per mention. The reporter does some arithmetic and sends an email. There's no always-on server, no heavy infrastructure, and nothing that posts anything back. At typical SMB volume the bill is a couple of dollars a month, fixed cost essentially zero — and unlike most systems here, the one variable that matters is simply how many mentions you get.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (around 200 mentions a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The dominant cost is the per-mention mood read on Bedrock — a fraction of a cent each.
- Collecting, the trend math, the alerts, and the weekly pulse are all pennies.
- At 1,000 mentions the bill is around $5. At 2,000 it's around $9.

Cost at three volumes

Fig 6. Monthly cost at three mention volumes. Bedrock is the dominant slice because it's one mood read per mention. SNS, SES, and the fixed services stay small because alerts and the weekly pulse fire only a handful of times. Everything else — the collector, the queue, the trend math — is pennies.

Where the dollars actually go

Bedrock (the bulk). One Haiku 4.5 call per mention to read its mood: a few hundred input tokens (the mention text) and a few tokens out (the score and reason), so a fraction of a cent each. At 200 mentions a month that's a dollar or so; at 2,000 it's a few dollars. Add the one weekly summary line — four or five calls a month — and Bedrock is still the single largest line, but it grows only as fast as your mention count does. This is the one cost that tracks volume directly, by design: the system pays to read exactly the mentions you actually get.

Lambda runtime. The poller every fifteen minutes, the reader pulling from the queue, the webhook handler, the reporter, and the weekly pulse. Each run is short — reading a feed, scoring a mention, sending an email. Pennies a month at all three volumes.

DynamoDB on-demand. Three small tables: `sm-scores`, `sm-seen`, `sm-alerts`. One write per scored mention, one de-duplication read per collected mention, and small reads for the trend and the alerts. Pennies a month.

SQS + S3. The mention queue and its dead-letter queue, plus the listening-export bucket and the stored config. A handful of messages and a few hundred KB. Effectively free at this scale.

EventBridge Scheduler. The 15-minute poll, the hourly housekeeping, the weekly pulse, and the occasional deferred-alert one-off. A few invocations an hour. Pennies.

SNS + SES. SNS sends the instant alerts (email, optional text); SES sends the weekly pulse. Both are a handful of messages a month for a typical SMB — alerts

only fire on bad events, the pulse goes once a week. A few cents in total, dominated by any text messages you opt into.

What doesn't cost money

- **API Gateway.** Replaced by a Lambda Function URL for the comment webhook.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Nothing runs between a poll and a mention arriving.
- **A vector store.** The monitor scores and trends — it doesn't search past mentions by meaning. No embeddings, no S3 Vectors, no Knowledge Base.
- **Anything that posts back.** The monitor never replies, so there's no outbound posting cost and no risk surface that comes with it.

How the cost scales

Bedrock grows roughly linearly with mention count, because it's one call per mention. Lambda and DynamoDB grow linearly too, but from a much smaller base. SNS, SES, and the fixed services barely move — you don't get more weekly pulses by having more mentions, and alerts fire on events, not volume. So the bill at 5,000 mentions a month is around $20; at 10,000 it's around $40. Past those volumes the per-mention model still holds; you'd just look at batching several short mentions into one model call to trim the Bedrock line, which is an optimization rather than a redesign.

Set an AWS Budgets alarm at $15/month so anything unusual — a misconfigured feed that re-queues the same mentions, say — pages you before the bill matters. The monitor'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, the SQS and DLQ config, and EventBridge Scheduler config.

PART 7 OF 7

JUNE 1, 2026 PART 7 OF 7 · SENTIMENT MONITOR SERIES ~8 MIN READ

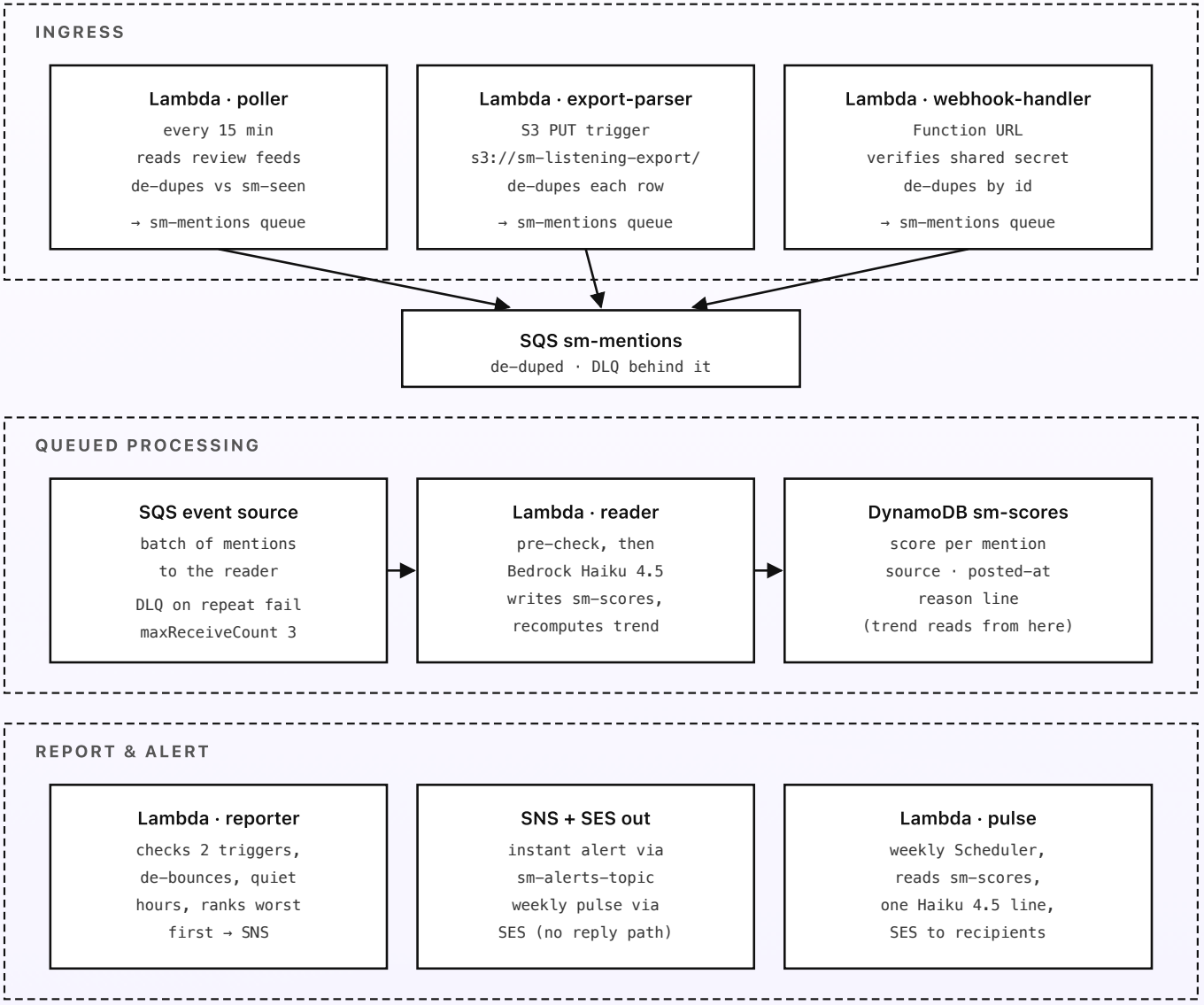
Engineering reference: the sentiment monitor architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SQS and dead-letter-queue config, EventBridge Scheduler config, the DynamoDB schemas, and the alert delivery path. 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, SQS, SNS, SES, 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 missed mood swing, not a regional outage, and a stalled poll simply resumes on the next tick. One AWS account dedicated to the monitor (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. The monitor has no write path to any external account by design: it reads sources and sends to your own inbox, nothing else.

Topology



The monitor only reads and reports — there is no write path back to any source.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into one queue), queued processing (the reader scoring each mention into `sm-scores`), and report and alert (the reporter and pulse fanning out through SNS and SES). Every Lambda is event-, queue-, or schedule-driven; nothing is synchronous-chained, and nothing writes back to a source.

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.

- **`poller`** — EventBridge Scheduler target, fires every 15 minutes. Reads each review feed listed in `/sm/config/sources` (Parameter Store), computes a stable id per item (`sha256(platform + native_id)`), checks it against `sm-seen` with a conditional write, and sends genuinely new items to the `sm-mentions` SQS queue. Feed credentials, where needed, live in Secrets Manager under `sm/sources/*`. Memory: 256 MB. Timeout: 30 s.
- **`export-parser`** — S3 PUT trigger on `s3://sm-listening-export/`. Reads the dropped export (CSV or JSON Lines), iterates rows, computes the stable id, de-duplicates against `sm-seen`, and enqueues new mentions to `sm-mentions`. Tolerant of partial files: a malformed row is logged and skipped, never fatal. Memory: 256 MB. Timeout: 60 s.
- **`webhook-handler`** — Lambda Function URL, `AuthType: NONE`, verifies an HMAC shared secret (`sm/webhook/secret` in Secrets Manager) on the raw request body before doing anything. Parses the comment payload, computes

the id, de-duplicates, and enqueues to `sm-mentions`. Returns 200 quickly; all work after signature check is minimal. Memory: 256 MB. Timeout: 15 s.

- **`reader`** — SQS event source on `sm-mentions`, batch size 5, with partial-batch-response enabled so one bad mention doesn't fail the batch. For each mention: clean/pre-check; if worth a call, invoke Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) for a mood score (−2..+2) and a one-line reason, JSON only; validate; write to `sm-scores`; recompute the trailing-window rolling average and write it to `sm-trend`. Heavier reasoning is not justified here, so Sonnet is not used. Memory: 512 MB. Timeout: 60 s. Mentions that fail 3 times land in the DLQ.
- **`reporter`** — invoked after a reader batch (via an internal event) to evaluate the two alert triggers. Reads `sm-trend` and the relevant `sm-scores` rows; applies the single-mention floor and the slope check from `/sm/config/rules`; de-bounces against `sm-alerts` within the cool-down window; applies quiet hours (deferring routine alerts via a one-off EventBridge Scheduler rule, letting urgent ones through); ranks worst-first; composes from the voice template; and publishes to SNS topic `sm-alerts-topic`. Writes the alert to `sm-alerts`. *No Bedrock calls.* Memory: 256 MB. Timeout: 30 s.
- **`pulse`** — EventBridge Scheduler target, weekly Monday 8am local. Reads the past week of `sm-scores` and last week's stored summary; builds the three blocks (trend, standout mentions, what-changed) in plain Python; makes one Bedrock Haiku 4.5 call for the summary line; renders the email and sends via SES to the recipients in `/sm/config/recipients`; writes this week's figures to `sm-summary` for next week's comparison. Memory: 512 MB. Timeout: 60 s.

- `housekeeping` — EventBridge Scheduler target, hourly. Expires old `sm-seen` rows past the de-dup horizon (TTL-driven, this is a safety sweep), and re-drives any DLQ messages that look transient. Memory: 256 MB. Timeout: 30 s.

Storage

- **DynamoDB** · `sm-scores` — one row per scored mention. PK `mention_id`; sort key `posted_at`; attributes: `source`, `score` (−2..+2), `reason`, `author`, `link`. On-demand. GSI on `(source, posted_at)` for per-source weekly rollups. No TTL — this is the long-term record.
- **DynamoDB** · `sm-seen` — de-duplication ledger. PK `mention_id`; attribute `first_seen`. TTL on a 90-day horizon so the table self-trims. On-demand. Written with a conditional put so concurrent lanes can't double-enqueue.
- **DynamoDB** · `sm-alerts` — one row per instant alert. PK `event_key` (a hash of the trigger type + window bucket); attributes: `fired_at`, `count`, `worst_mention_id`, `severity`. On-demand. The de-bounce reads this; `count` grows as repeat firings fold in.
- **DynamoDB** · `sm-trend` — the current rolling average and slope. PK `window` (e.g. `rolling_14d`); attributes: `avg`, `prev_avg`, `slope`, `updated_at`. On-demand. Tiny; effectively a single hot row.
- **DynamoDB** · `sm-summary` — one row per weekly pulse for week-over-week deltas. PK `week_start`; attributes: per-source counts, weekly average, standout ids. On-demand.
- **S3** · `sm-listening-export` — drop zone for the social-listening tool's exports. Versioning enabled. Lifecycle to Glacier at 30 days; expiry at 1 year.

- **S3 · `sm-config-source`** — the rules and voice docs as plain text, mirrored to Parameter Store keys on change. Versioning enabled so a bad edit rolls back in one click.

SQS and the dead-letter queue

- **`sm-mentions`** — standard queue, visibility timeout 90 s (six times the reader timeout headroom is not needed; the reader is fast). Redrive policy: `maxReceiveCount: 3` to the DLQ. Absorbs ingest bursts from a viral thread so the reader processes at a steady, cost-predictable rate.
- **`sm-mentions-dlq`** — dead-letter queue for mentions that fail to process three times (bad payload, transient Bedrock error that didn't clear). A CloudWatch alarm on DLQ depth > 0 pages the admin; `housekeeping` re-drives anything that looks transient.
- **Partial-batch response** — the reader returns `batchItemFailures` so a single poison mention is retried/dead-lettered without re-processing (and re-charging Bedrock for) the rest of its batch.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `reader` for the per-mention mood read, and `pulse` for the weekly summary line. Sonnet 4.6 is deliberately not used — mood labeling and a one-line summary don't justify the heavier path.

- **Prompts.** Both prompts demand JSON-only output with a fixed schema; the reader pins the score to the −2..+2 range and instructs the model to return 0 with a note when text is ambiguous rather than guessing.
- **Embeddings.** Not used. The monitor scores and trends; it doesn't retrieve past mentions by meaning. No Titan embeddings, no S3 Vectors, no Knowledge Base.
- **Quotas.** Default account quotas are more than enough at SMB volume. The per-mention calls are short; bursts are smoothed by the SQS queue and the reader's batch size, so Bedrock is never hit by the raw ingest spike.

EventBridge Scheduler config

- `sm-poll` — `rate(15 minutes)`. Target: `poller` Lambda.
- `sm-housekeeping` — `rate(1 hour)`. Target: `housekeeping` Lambda.
- `sm-weekly-pulse` — `cron(0 8 ? * MON *)` in `TZ_NAME`. Target: `pulse` Lambda.
- **One-off rules** — created on the fly by `reporter` when a quiet-hours defer is needed. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SNS and SES (outbound only)

- SNS topic `sm-alerts-topic` carries the instant alerts. Subscriptions: the admin's email (always) and an optional SMS endpoint for urgent severity. The topic is the only fan-out point for alerts.

- SES outbound sends the weekly pulse. Verify a sender identity at `monitor@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request. There is no SES *inbound* rule set — the monitor never receives mail and never replies.
- No outbound path posts to any review site, social platform, or comment thread. The IAM roles below contain no permission that could.

IAM (least privilege per Lambda)

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

- **poller role:** `dynamodb:PutItem` (conditional) on `sm-seen`; `sqs:SendMessage` on `sm-mentions`; `ssm:GetParameter` on `/sm/config/*`; `secretsmanager:GetSecretValue` on `sm/sources/*`; outbound network to the feed hosts. *No* `bedrock:*`.
- **export-parser role:** `s3:GetObject` on `sm-listening-export`; `dynamodb:PutItem` on `sm-seen`; `sqs:SendMessage` on `sm-mentions`.
- **webhook-handler role:** `secretsmanager:GetSecretValue` on `sm/webhook/secret`; `dynamodb:PutItem` on `sm-seen`; `sqs:SendMessage` on `sm-mentions`.
- **reader role:** `sqs:ReceiveMessage` + `DeleteMessage` on `sm-mentions`; `bedrock:InvokeModel` on the Haiku ARN; `dynamodb:PutItem` + `UpdateItem` on `sm-scores` and `sm-trend`; `dynamodb:Query` on `sm-scores` for the trend recompute.
- **reporter role:** `dynamodb:Query` on `sm-scores` and `GetItem` on `sm-trend`; `dynamodb:PutItem` + `UpdateItem` on `sm-alerts`; `sns:Publish` on `sm-`

`alerts-topic`; `scheduler:CreateSchedule` for the quiet-hours one-offs; `ssm:GetParameter` on `/sm/config/*`. *No* `bedrock:*`.

- **pulse role:** `dynamodb:Query` on `sm-scores` and `sm-summary`; `dynamodb:PutItem` on `sm-summary`; `bedrock:InvokeModel` on the Haiku ARN; `ses:SendEmail` from the verified sender identity; `ssm:GetParameter` on `/sm/config/*`.

Trend math and thresholds

The rolling average is a trailing-window mean over `sm-scores`, default 14 days, recomputed incrementally on each reader batch (subtract the rows now outside the window, add the new ones) so the cost is constant per batch rather than a full table scan. The slope is `avg − prev_avg` over the batch. Two configurable thresholds gate alerts, both in `/sm/config/rules`: `mood_floor` (default −2, the single-mention trigger) and `slope_drop` (default a drop steeper than 0.4 of a point over the window). A third value, `urgent_floor`, marks a mention or slope severe enough to bypass quiet hours. All three are plain numbers an owner can edit; no deploy.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a metric for alerting.
- **Alarms:** DLQ depth > 0 (a mention is stuck); reader Bedrock error rate > 2% in 1h (the model path is degraded); poller failures > 0 in an hour (ingest stalled);

webhook signature-verification failures > 10/hour (a misconfigured or hostile caller).

- **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 `sm-cost-alarm` subscribed to the admin's email. A runaway re-queue loop (a feed re-listing the same items) shows up here first.

Config and secrets

The source list, the rules thresholds, the quiet-hours window, the timezone, and the recipient list all live in Parameter Store under `/sm/config/`, mirrored from the plain-text docs in `sm-config-source`. Feed credentials live in Secrets Manager under `sm/sources/*`; the webhook HMAC secret under `sm/webhook/secret`. Lambdas fetch config on cold start and cache it for the lifetime of the execution environment. There are no credentials anywhere that grant write access to an external platform — the monitor has nothing to post with.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys), AWS SAM for the stack. The opinionated bits: deploy the SQS queue and its DLQ together with the redrive policy in one template so they can't drift apart; turn on S3 versioning for both `sm-listening-export` and `sm-config-source`; enable partial-batch-response on the reader's event source mapping so a poison mention never re-charges Bedrock for its whole batch; and pin the EventBridge Scheduler timezone so the weekly pulse doesn't silently move to UTC after a CI rotation. Total

deployable surface: around seven Lambdas, five DynamoDB tables, two S3 buckets, one SQS queue plus DLQ, two SNS topics, one SES sender identity, the Scheduler rules, 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.