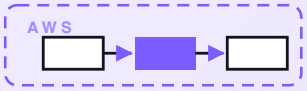
7-PART SERIES · FREE COMPANION



Newsletter composer

A serverless composer that gathers the week's new blog posts, product notes, and wins from the sources you point it at; drafts a clear, friendly email issue in your voice grounded in those items; and hands it to the owner to approve, edit, or skip before it sends. A human stays in control — nothing goes out without sign-off. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Newsletter composer

PART 1 OF 7

MAY 14, 2026 PART 1 OF 7 · NEWSLETTER COMPOSER SERIES ~5 MIN READ

A newsletter composer on AWS for a few dollars a month

A small business has more to share than it ever gets around to sending. The two blog posts that went up this month and nobody told the list about. The new feature that quietly shipped. The big client win the team is proud of. The case study that's been sitting in a folder. A newsletter is the easiest way to stay top of mind with the people who already like you — and it's the first thing that slips, because writing one from a blank page on a busy Friday is nobody's favorite task. This post walks through the design of a small composer that gathers the week's updates, drafts a clear, friendly issue in your voice, and hands it to you to approve, edit, or skip before anything sends.

KEY TAKEAWAYS

- Three sources for items: a Drive sheet anyone can add to, a feed lane for your blog, and an inbox forwarding lane.
- Every weekly run ends in one of four moves: skip, draft, redraft, or ready for review.
- The draft is grounded only in the items you gathered, and every claim links back to a source.
- The owner is always the last step — approve, edit, or skip. Nothing sends without sign-off.
- Designed on AWS for about $2.80/month at typical small-business volume.

The whole system on one page

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

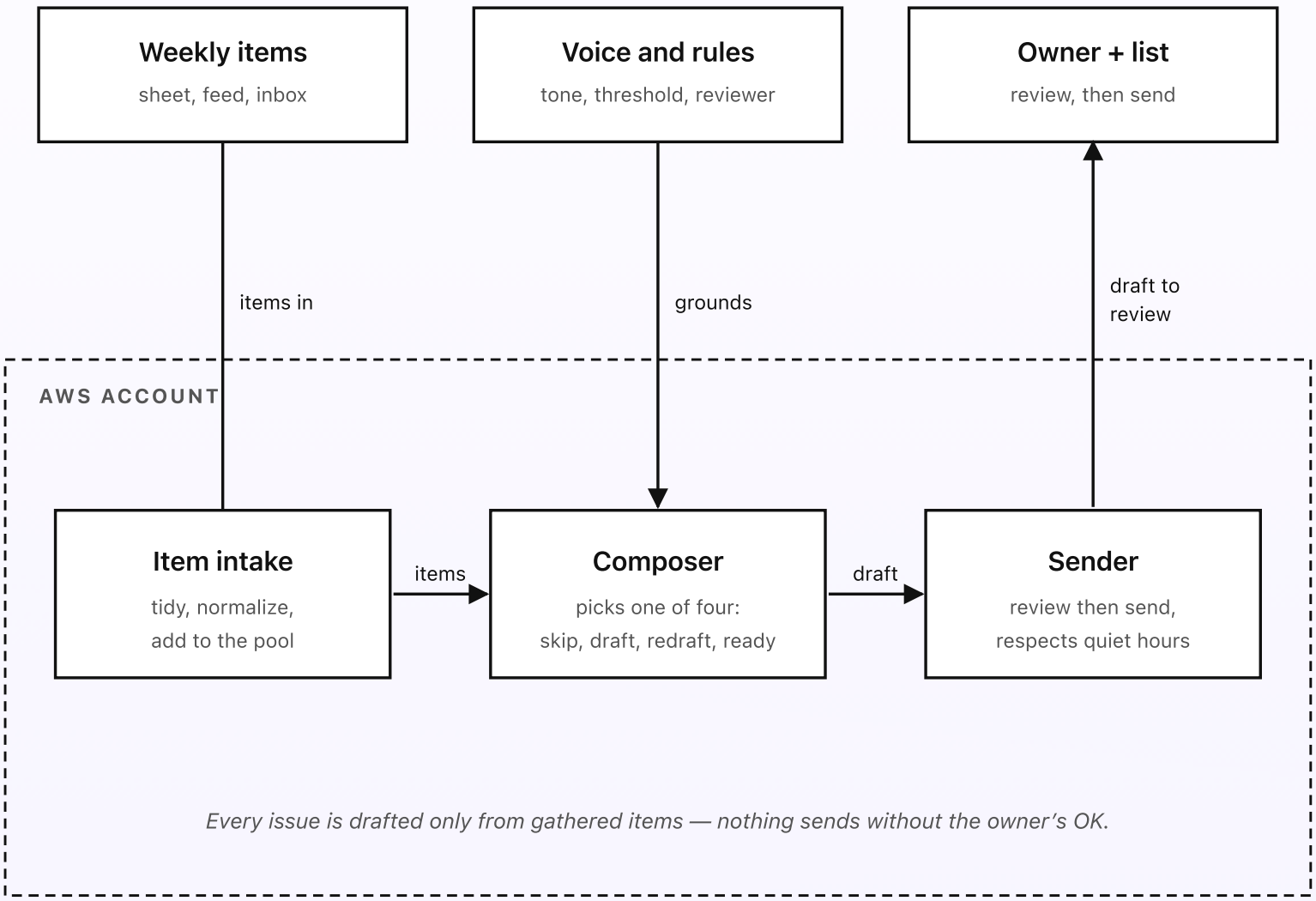

Weekly items
sheet, feed, inbox
Voice and rules
tone, threshold, reviewer
Owner + list
review, then send
items in
grounds
draft to review
AWS ACCOUNT
Item intake
tidy, normalize, add to the pool
items
Composer
picks one of four: skip, draft, redraft, ready
draft
Sender
review then send, respects quiet hours
Every issue is drafted only from gathered items — nothing sends without the owner's OK.

Fig 1. Three sources outside, three pieces inside AWS. Items flow in from a Drive sheet, a feed lane, and an inbox forwarding lane. The Composer runs weekly and picks one of four moves. The Sender shows the draft to the owner and only mails an approved issue to the list.

What you set up once (the outside)

- **Weekly items.** A Google Sheet in a Drive folder, one row per item: a short title, a one-line note, a category (blog post, product note, win, event, other), a link, and the date it happened. Anyone on the team can drop a win or a product note in here. New items also flow in from two other lanes covered in Part 2 — a feed lane (the composer watches your blog's RSS feed and proposes a row for each new post) and an inbox-forwarding lane (forward an update to a dedicated address and the composer proposes a row for one-tap approval).
- **A voice and rules folder.** Two short Google Docs in a Drive folder. The *voice* doc holds your tone, your standard greeting and sign-off, a few example past issues, and a short list of things to never say. This is what keeps the draft sounding like you and not like a robot. The *rules* doc covers how many fresh items make a worthwhile issue (default three), which day the issue sends, who reviews it, the quiet hours, and the cooling-off window between approval and send.
- **Owner and list.** The owner reviews every draft. The draft lands with the full issue, a flag on anything the model couldn't tie to a source item, and three buttons — *approve*, *edit*, *skip*. The subscriber list is the people who opted in; only an approved issue is ever sent to them.

What runs on every run (the inside)

- **The item intake.** Three sources feed the pool. The Drive sheet is the canonical store. New items can also be added via the feed lane (a small sync Lambda checks your blog's RSS every hour and proposes a row for each new post) and the inbox forwarding lane (forward an update to `updates@your-company.com`, the composer uses Bedrock Haiku 4.5 to tidy it into a clean title and one-line note, then drops a one-tap approval card in the team's Slack to confirm before the row is added).
- **The composer.** Runs once a week, the morning before the send day. Reads the item pool. Counts the fresh items added since the last issue. Compares against the threshold in the rules doc. Picks one of four moves. *Skip*: too few fresh items — tell the owner why, send nothing. *Draft*: enough items — call Bedrock Sonnet 4.6 to write the issue, grounded strictly in those items and the voice doc. *Redraft*: a self-check found a claim with no source — rewrite once to ground it. *Ready*: the draft passed its checks — hand it to the owner for review. The heavy writing is one Bedrock call per issue, not per item.
- **The sender.** Posts the finished draft to the owner for review (Slack message or email) with the full issue and the approve/edit/skip buttons. On approve, it waits out a short cooling-off window (default 30 minutes, so a typo caught after the fact can still be pulled), then sends the issue to the subscriber list through SES outbound, respecting quiet hours. Every run, every action, and every send writes a row in DynamoDB so the trail is clear. A monthly summary writes a short note: issues sent, open rate if available, items that never made it in.

In plain words

It’s Thursday morning. Over the past week your blog published two posts, the team closed a nice deal with a regional client, and a small feature shipped. Three of those landed in the item pool automatically (the blog posts via the feed lane, the feature via a forwarded note); someone typed the client win into the sheet. The composer runs, sees four fresh items, and drafts a friendly issue: a warm intro, a short paragraph on each item with a link, and your usual sign-off. It checks its own work, finds every line traces back to an item, and posts the draft to you in Slack with three buttons. You read it, fix one phrase you’d say differently, tap *Approve*. Thirty minutes later the issue goes out to your list. Nobody wrote it from a blank page, and nothing went out that you hadn’t read.

The cost of running this is about $2.80 a month at SMB volume. The cost of *not* running it is the list that hears from you twice a year, the wins nobody outside the building knows about, and the slow drift of an audience you worked hard to earn.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every issue is drafted only from gathered items — the composer never invents news. Every claim links back to a source.
- Four moves, always. Skip, draft, redraft, ready. There is no fifth.
- The owner is the last step. Approve, edit, or skip. Nothing sends without a human signing off.
- Quiet hours and a cooling-off window are respected. A send is hard to undo; the design slows it down on purpose.
- Too few items means skip, not pad. A thin issue trains people to stop opening you.
- Every run and every action is logged. Ask next year why an issue did or didn't go out and the trail answers.

Why this shape

Most teams write a newsletter in one of three ways: someone blocks an afternoon when they remember, a marketing tool with a blank template waits to be filled, or it never happens. The afternoon works until the afternoon gets eaten — and the weeks it gets eaten are the busy weeks, which are exactly the weeks worth writing about. The blank template is a chore, not a system; it tells you to write but does none of the writing. And "never" is how a hard-won audience quietly forgets you exist.

The setup above keeps your updates in a sheet the team already touches, but adds a small system that *gathers* them every week and does the first draft for you — grounded only in what actually happened, in your voice, with the links already in. It skips weeks that aren't worth sending instead of padding them. It hands you a finished draft you can ship in a tap or fix in a minute. And it never, ever sends on its own. The composer does the part you dread; you keep the part only you should own.

The next four posts walk through each piece in turn: how a newsletter gathers the week's updates, how an issue gets drafted, how a draft reaches the owner, and how an issue gets approved and sent. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 14, 2026 PART 2 OF 7 · NEWSLETTER COMPOSER SERIES ~4 MIN READ

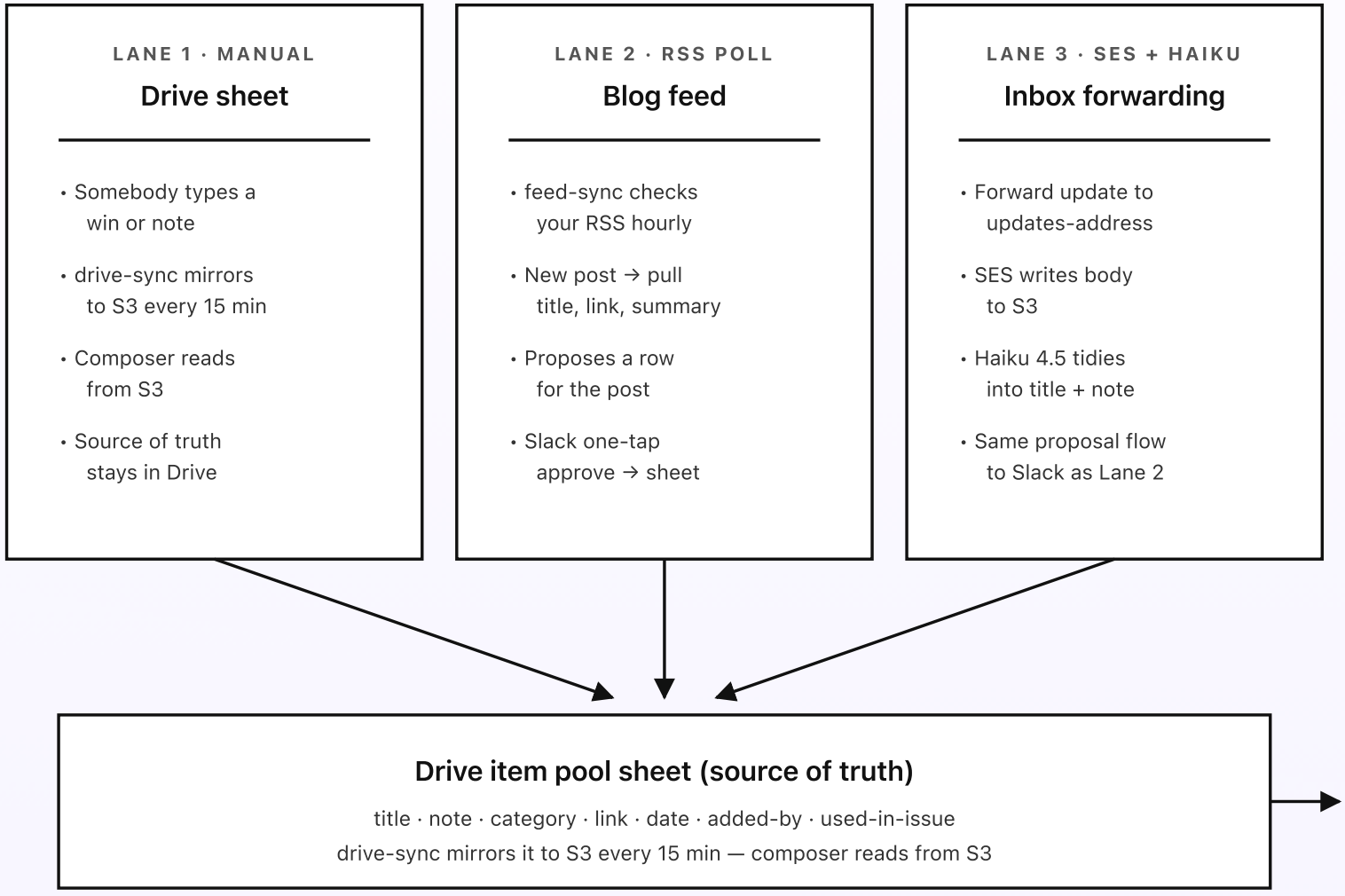
How a newsletter gathers the week's updates

The composer can only write about what it's been given. So the first job is making sure the item pool actually reflects what happened this week. There are three ways an item gets in: somebody types a row in the Drive sheet, your blog publishes a post the feed lane picks up, or somebody forwards an update to a dedicated address. The first one is obvious. The other two exist because in real life nobody types a row in a sheet for the post that went live three minutes ago.

KEY TAKEAWAYS

- Three intake lanes feed one item pool: the Drive sheet, a blog feed lane, and an inbox-forwarding lane.
- The feed lane watches your blog's RSS and proposes a row for each new post.
- Forwarded updates are tidied by Bedrock Haiku 4.5 into a clean title and a one-line note.
- Every proposed row goes to the team's Slack for one-tap approval before it lands in the pool.
- The Drive sheet stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one item pool



The Drive sheet stays the source of truth — the other lanes are conveniences that propose rows for it.

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the feed lane and the inbox lane are conveniences that propose rows for human approval. The drive-sync Lambda mirrors the sheet to S3 so the composer can read it without hitting Drive on every run.

Lane 1: the Drive sheet itself

The simplest lane. Open the item-pool sheet in Drive, add a row, save. The columns are short: a title, a one-line note, a category (blog post, product note, win, event, other), a link, the date it happened, and who added it. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://nc-items-source/items.csv` if the sheet has changed since the last sync. The composer reads from S3, not Drive directly. That keeps Drive API calls predictable and gives you S3 versioning for free, so a bad bulk-edit can be rolled back in one click.

This lane covers the things that don't show up anywhere else: the client win, the trade-show booth that went well, the kind note a customer sent. Anyone on the team can drop those in throughout the week without learning a new tool.

Lane 2: the blog feed (the lane that fills itself)

Most of what belongs in a newsletter is already published somewhere — on your blog. A small `feed-sync` Lambda runs hourly, fetches your blog's RSS or Atom feed, and compares the entries against the ones it has already seen (tracked by a stable id per entry in DynamoDB). For each genuinely new post, it pulls the title, the link, and the short summary the feed provides, and creates a proposed row. The proposal goes to the team's Slack with *approve*, *edit*, and *discard* buttons. On *approve*, the row is written to the Drive sheet via the Sheets API.

The reason a new post is a proposal and not an automatic add is simple: not every post belongs in every issue. A small internal note, a re-publish, or a post that's off-topic for the list shouldn't pad the issue. One tap keeps the good ones and drops the rest, and the team stays in the loop on what the newsletter will cover.

Lane 3: inbox forwarding

Set up a dedicated inbound address — something like `updates@your-company.com` — via Amazon SES. Anyone on the team forwards an update to that address: a Slack message they copied, a quick "we shipped X" note, a screenshot caption, a paragraph about a customer story. SES writes the raw message to `s3://nc-raw-mime/`. The S3 PUT triggers a parser Lambda. The Lambda reads the body and calls Bedrock Haiku 4.5 to tidy the raw text into a clean title, a one-line note, a category, and any link it found. The model prompt is short: "Turn this forwarded update into one newsletter item. Return JSON only. Keep the facts; do not add any that aren't in the text."

The tidied item goes to the same Slack proposal flow as Lane 2 — the proposed row, and three buttons: *approve*, *edit*, *discard*. On *approve*, a Lambda writes the row to the Drive sheet. On *edit*, the person gets a fillable modal pre-populated with the proposal. On *discard*, the message is logged and the raw forward moved to a discarded prefix in S3 for audit. Forwarding is the lane that catches the wins that live in someone's head or a chat thread and would otherwise never reach the list.

Why the pool stays the source of truth

Three lanes in, but only one place the composer actually reads. That's a deliberate constraint. If the feed lane and the inbox lane both wrote straight into the draft, every "why is this in the issue?" question would mean checking three places. Funneling everything through the Drive sheet means there is exactly one row per item, anyone can read or edit any of it, and the composer's input is a single, reviewable list. The convenience lanes are first-class for getting items in, but they always pass through the sheet on the way.

Next post: how the composer reads the pool, decides whether there's enough to send, and drafts the issue grounded only in those items.

PART 3 OF 7

MAY 14, 2026 PART 3 OF 7 · NEWSLETTER COMPOSER SERIES ~5 MIN READ

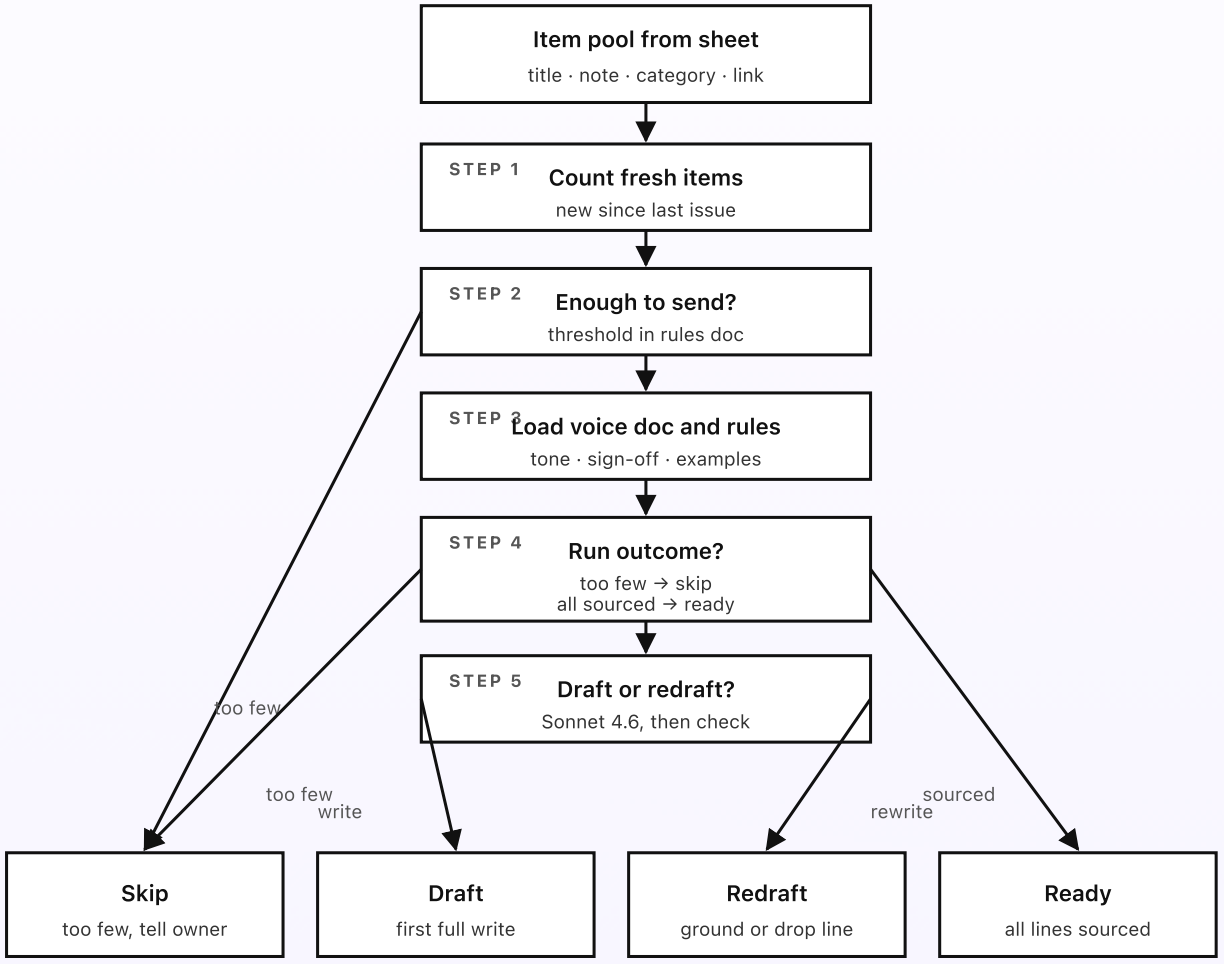
How a newsletter issue gets drafted

Once a week, the morning before the send day, an EventBridge Scheduler rule fires the composer Lambda. The Lambda reads the item pool, counts what's fresh since the last issue, and decides whether there's even enough to send. If there is, it calls a model once to write the whole issue — grounded only in those items and your voice doc — then checks its own work before handing the result on. The counting and the rules are plain Python. The writing is one Bedrock call, not a dozen.

KEY TAKEAWAYS

- The composer runs once a week via EventBridge Scheduler, the morning before the send day.
- It counts fresh items first — too few (default fewer than three) means skip, not pad.
- Four moves per run: skip, draft, redraft, ready.
- The draft is grounded only in the gathered items and the voice doc — every claim links to a source.
- A self-check catches any unsourced line and triggers one redraft before the issue is marked ready.

The decision flow, per run



The voice doc holds the tone — change it and next week's issue uses the new voice.

Fig 3. The composer's decision tree, per weekly run. Five steps decide which of four moves applies. The rules doc holds the threshold and the voice doc holds the tone; the composer only drafts from gathered items.

Three items isn't magic, it's in the doc

The rules doc has one short line for the threshold: "Send an issue when at least three fresh items are in the pool. Below that, skip and tell me why." The number is how many genuinely new items have to be waiting before an issue is worth your readers' time. Set it to three and a slow week skips. Set it to one and you'll send every week, thin issues included. Set it to five and you'll only send when there's real news. The number is yours to change without touching code.

The threshold exists for one reason: a thin issue is worse than no issue. The fastest way to train your list to stop opening your email is to send them a newsletter with one stretched-thin update and a lot of filler. Skipping a quiet week respectfully protects the weeks when you do have something to say.

Grounded drafting: the model writes only from your items

When there's enough to send, the composer makes one Bedrock call. The prompt has three parts: your voice doc (tone, greeting, sign-off, examples, the never-say list), the fresh items as a clean list (each with its title, note, category, and link), and a short instruction: "Write this week's issue using only these items. One short paragraph per item, in this order. Use the voice. Put the link in. Do not add news that isn't in the items. Tag each paragraph with the id of the item it came from."

That last instruction is the important one. Because every paragraph is tagged with the item it came from, the composer can check — and the owner can see — exactly where each line came from. The model isn't asked to be clever or to find news; it's asked to turn a list you approved into a friendly email in your voice. The judgment about *what* goes in the issue was already made when the items were gathered and approved in Part 2.

The heavier reasoning model (Sonnet 4.6) is used here, and only here, because writing a coherent issue across several items in a consistent voice is the one genuinely hard writing task in the whole system. Everywhere else — tidying a forwarded note in Part 2, the monthly summary in Part 6 — the cheaper Haiku 4.5 is plenty.

Four moves, always

Every weekly run lands in exactly one of four buckets. The names are simple on purpose.

- **Skip.** Fewer fresh items than the threshold. Send nothing this week. Write a short note to the owner: "Only two new items this week, skipping. Here they are if you disagree." Log the skip so the trail shows why no issue went out.
- **Draft.** Enough items. Make the one Bedrock call and produce the first full issue, grounded in the items and the voice.
- **Redraft.** The self-check found a sentence that doesn't trace back to any item. Make one more call asking the model to either ground the line in a real item or drop it. One redraft, not a loop — if the second pass still has a gap, the issue is handed over with the gap flagged for the owner rather than rewritten forever.

- **Ready.** Every line traces to a source. Mark the draft ready and hand it to the sender for the owner's review (Part 4). Most weeks with enough items land here after one draft and a clean check.

State that keeps the run honest

The composer reads and writes a small DynamoDB table, `nc-items-state`, that records which items have already been used in a past issue: `(item_id, used_in_issue, used_date)`. That's how "fresh" is defined — an item that's already gone out in a previous issue isn't counted again. A second table, `nc-issues`, records each run: the move chosen, the item ids used, and a pointer to the draft stored in S3. With those two tables, the count is deterministic and a re-run produces no duplicate issue: the state shows what already happened.

Approving an issue (Part 5) marks its items as used. A skipped week leaves the items fresh, so they roll into next week's count naturally.

Why the model only writes, never decides to send

The composer could ask a model whether the week is worth sending, or let it pick which items make the cut. It doesn't. The decision to send and the choice of items are deterministic and human-controlled — the threshold is a number you set, and the items were approved one by one in Part 2. The model's only job is to turn an approved list into a well-written email. Keeping the model out of the "should we send?" decision means the system never surprises you by mailing your list on a week you'd have sat out, and never quietly drops an item you wanted in.

Next post: how the finished draft reaches the owner — reviewer resolution, the grounding flag, quiet hours, and the four guardrails between the draft and the review message landing.

PART 4 OF 7

MAY 14, 2026 PART 4 OF 7 · NEWSLETTER COMPOSER SERIES ~5 MIN READ

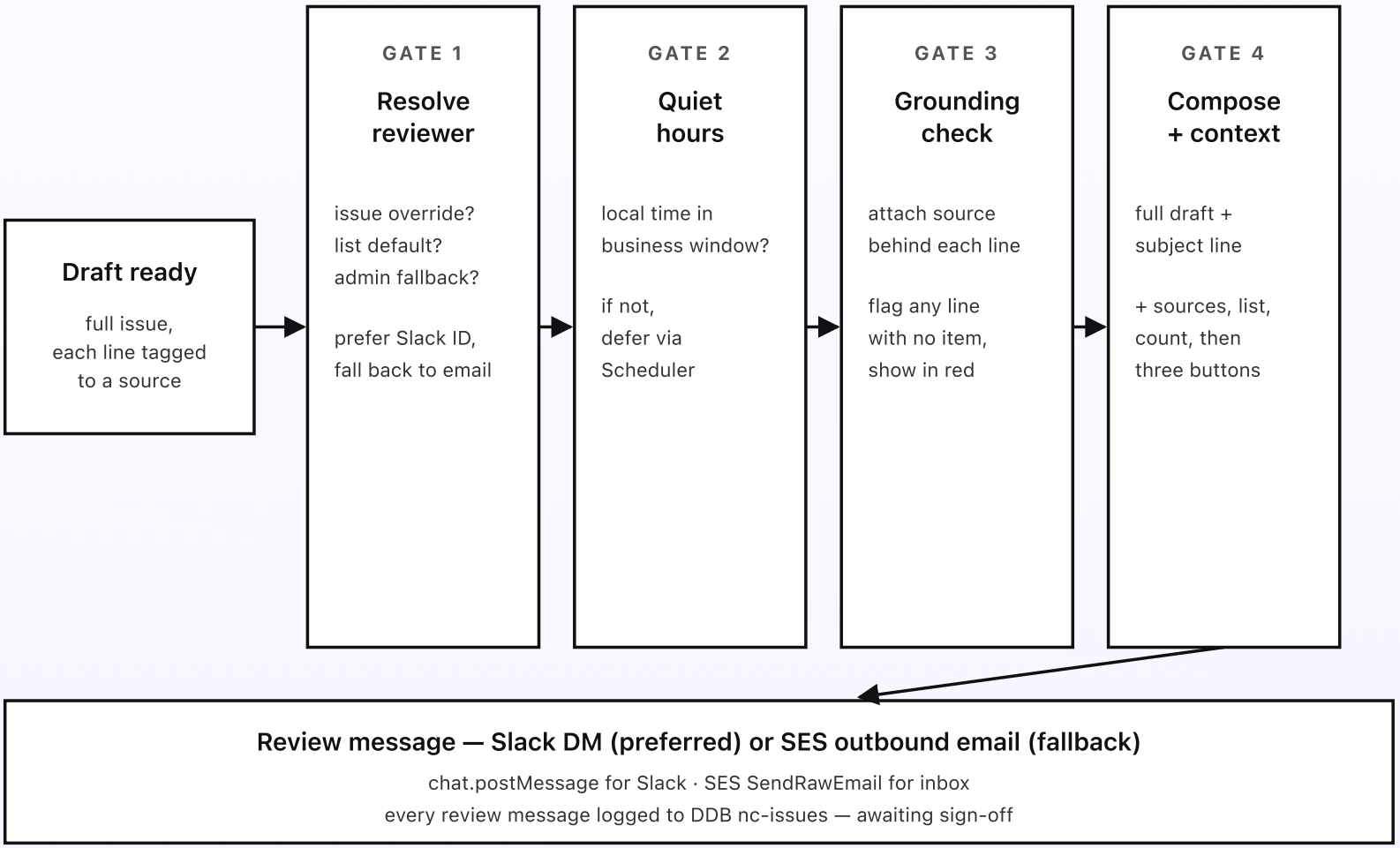
How a newsletter draft reaches the owner

The composer marked an issue ready. Now the sender Lambda has to figure out who reviews it, on what channel, at what time of day, and with what attached. Get any of those wrong and the review is worse than useless: a draft sent to someone who left, a 2am ping, a draft that hides the one line nobody can back up. Four small guardrails sit between the finished draft and the review message landing — and the most important one makes sure the owner sees exactly what the model couldn't source.

KEY TAKEAWAYS

- Reviewer resolution: per-issue override beats the per-list default beats fallback to the configured admin.
- Slack is the default review channel; email is the fallback if no Slack ID is configured.
- Quiet hours defer the review message to the next available business hour.
- The grounding check attaches the source item behind every paragraph and flags anything unsourced.
- Every review message ships with the full draft, the sources, and the Approve, Edit, and Skip buttons.

Four guardrails on every review message



Every gate is a deterministic check — and nothing is sent to the list until the owner approves.

Fig 4. Four guardrails between the finished draft and the review message. Resolve the reviewer. Honor quiet hours. Run the grounding check. Compose with the full draft and its sources. Then ship via Slack or email and log it so the system knows the draft is awaiting sign-off.

Gate 1: resolve the reviewer

Three places the sender Lambda looks for the reviewer of an issue, in order. First, a per-issue override in the rules doc — if this week's issue is assigned to a specific person (say the founder is out and a colleague is covering), that person reviews it. Second, the per-list default in the rules doc ("the customer newsletter is reviewed by the marketing lead"). Third, the configured admin fallback — the person who set the composer up and gets every unrouted draft. The fallback should never fire in steady state; if it does, the monthly summary names it so the rules doc can be fixed.

Once the sender knows which person to ask, it looks up their delivery preference. The rules doc maps each reviewer to a Slack member ID if one is set, otherwise to an email address. Slack is preferred because a review with action buttons is faster than an email round-trip, and the draft sits right there in the thread. Email is the fallback so a draft never gets stuck waiting on a channel nobody checks.

Gate 2: quiet hours

The composer runs in the morning before the send day, so the first review message usually lands in business hours. But a redraft, a deferred run, or a manually triggered re-compose can finish later. Gate 2 reads the rules doc's quiet-

hours setting (default 6pm to 8am, configurable per business). If the current local time is in the quiet window, the sender creates a one-off EventBridge Scheduler rule that fires at the next business-hour minute and exits without sending. The Scheduler re-invokes the same sender Lambda with the same payload at the deferred time, where Gate 2 lets it through.

A review at 11pm is a review that gets read at 8am anyway — but now with a notification badge that's been nagging all night. Deferring is kinder and costs nothing but a few minutes' wait.

Gate 3: the grounding check (the one that matters most)

This is the guardrail that makes the whole system trustworthy. Because every paragraph in the draft was tagged with the id of the item it came from (Part 3), the sender can walk the draft line by line and pull up the exact item behind each one. The review message shows them side by side: the paragraph, and a small "from: [item title] [link]" under it. The owner can see at a glance that the line about the new feature came from the forwarded note, and the line about the client win came from the row someone typed in the sheet.

If any sentence has no matching source — which should be rare after the redraft step in Part 3, but never assume — it's flagged in red with a plain warning: "This line isn't backed by any item. Edit or remove it before sending." The owner can't miss it. A newsletter that quietly states something that didn't happen is the one failure mode worth real care here, and this gate makes it visible instead of buried.

Gate 4: compose with full context, then ship

Gate 4 assembles the review message: the proposed subject line, the full draft as it would appear in the email, the per-paragraph sources from Gate 3, the name of the list it would go to, and the subscriber count. Below all of that sit three buttons — *Approve*, *Edit*, *Skip*. The whole point is that the owner can make a confident decision without leaving the message: read the issue, see where every claim came from, and act.

For Slack, the message is posted via `chat.postMessage` with Block Kit blocks so the buttons work. For email fallback, the same content is wrapped in a small HTML email, and the buttons become links that hit a Function URL to record the action. Every review message — Slack or email — writes a row to `nc-issues` in DynamoDB marking the issue as awaiting sign-off, with a pointer to the draft in S3. Part 5 picks up what happens when the owner taps one of those buttons.

Why the guardrails exist

None of these gates are exotic. They're the care a thoughtful editor would take before handing a draft to the boss — check who's actually reviewing, don't interrupt them at 11pm, show your sources, and lay it all out so the decision is easy. Putting them in code as four small sequential gates makes them part of the design, not something you're trusting any one weekly run to remember. And none of them send anything to the list — that only happens after the owner approves, which is the whole subject of the next post.

Next post: how an issue gets approved and sent — the three actions on the review message, the cooling-off window, and how the send, the items, and the audit trail

all stay in sync.

PART 5 OF 7

MAY 14, 2026 PART 5 OF 7 · NEWSLETTER COMPOSER SERIES ~5 MIN READ

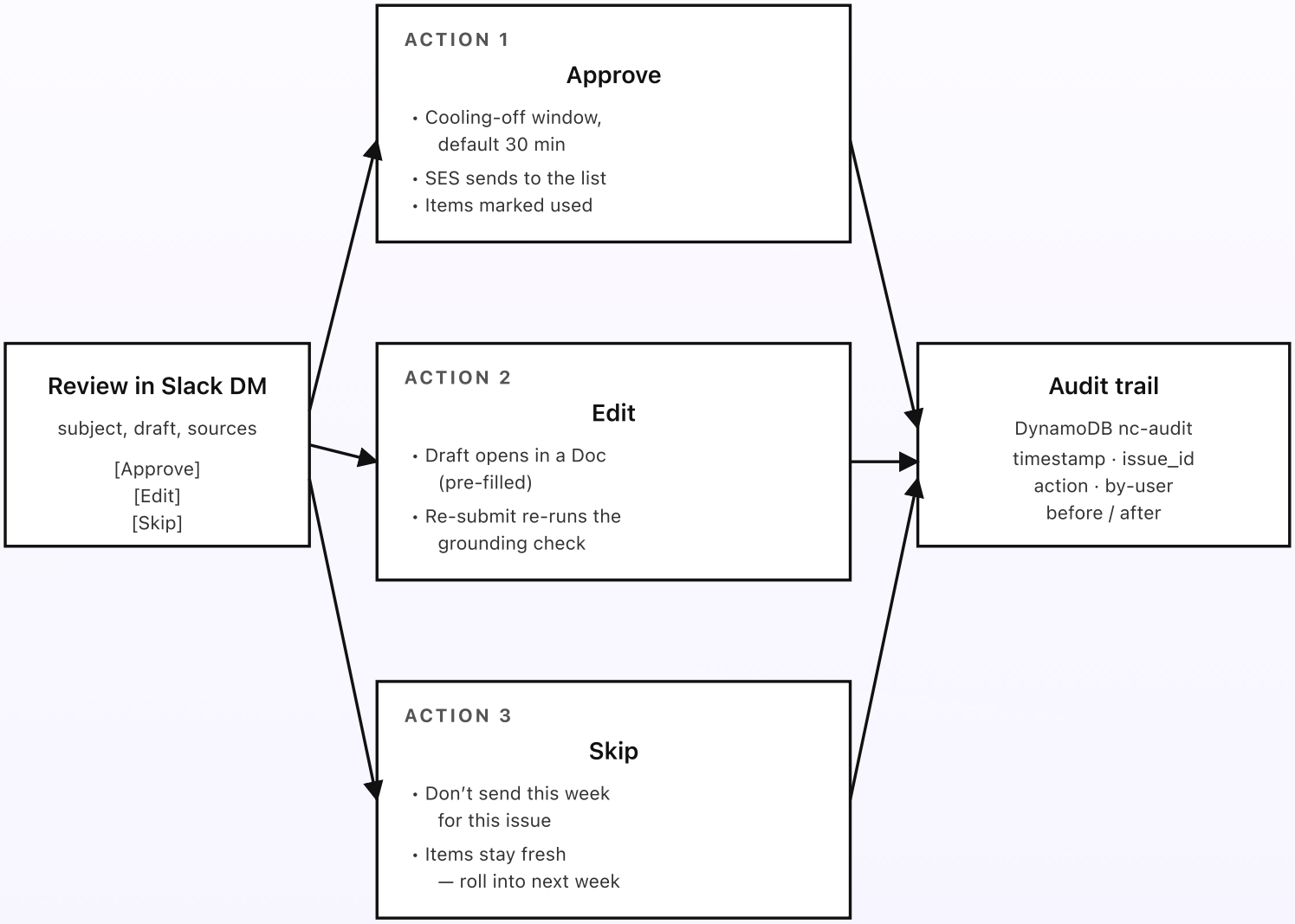
How a newsletter issue gets approved and sent

A review message lands in the owner's Slack at 8:03am. This week's issue is drafted, the sources are shown, there are three buttons. What happens when one gets tapped? The honest answer is "it depends which one." This post walks through the three things the owner can do — approve, edit, skip — and how the send, the item state, and the audit trail all stay in sync. The one rule underneath all of it: nothing reaches the list until the owner approves.

KEY TAKEAWAYS

- Three actions per review: *approve* (queue to send), *edit* (change the draft), *skip* (don't send this week).
- Approve waits out a cooling-off window before sending, so a typo caught late can still be pulled.
- Edit opens the draft in a doc; re-submitting re-runs the grounding check on the changed text.
- A sent issue marks its items as used, so they don't show up in next week's count.
- Every action writes a before-and-after snapshot to the audit trail.

Three actions on the review message



Nothing reaches the list until Approve — and even then, only after the cooling-off window.

Fig 5. Three actions per review, three different effects. Approve queues the issue and sends after a cooling-off window. Edit reopens the draft and re-runs the grounding check. Skip sends nothing and keeps the items fresh. Every action writes to the audit trail.

Action 1: approve (the most common)

The owner read the issue, the sources check out, the voice sounds right. They tap *Approve*. The button submits to a Function URL Lambda. The Lambda marks the issue approved in `nc-issues`, then — and this is the part that matters — it doesn't send right away. It starts a cooling-off window by creating a one-off EventBridge Scheduler rule (default 30 minutes, set in the rules doc). During that window the review message shows a small "Sending in 28 min — [Pull it back]" line. If the owner spots a typo two minutes after approving, they tap *Pull it back* and the issue returns to draft, unsent.

When the window closes, the Scheduler invokes the send Lambda. It renders the final issue, sends it to the subscriber list through SES outbound (in batches, respecting SES send limits), marks every item used in the issue as `used` in `nc-items-state` so it won't pad a future count, and writes a `sent` row to `nc-audit` with the recipient count and a snapshot of exactly what went out. The cooling-off window is the difference between "a send is permanent the instant you tap a button" and "a send is permanent after a half-hour grace period." For something as hard to unsend as an email blast, that half hour earns its keep.

Action 2: edit (the most useful)

Often the issue is 90% right. The owner likes it but would phrase the opening differently, or wants to drop one item, or wants to add a line of their own. They tap *Edit*. A Lambda copies the draft into a Google Doc in the issues folder, pre-filled with the current text, and replies with the link. The owner edits the Doc like any other document — rewrite the intro, delete a paragraph, tighten the sign-off — and taps a *Re-submit* button when they're done.

Re-submit matters more than it looks. The changed text goes back through the same grounding check from Part 4: any new sentence the owner added is checked against the items too, and if they wrote a claim with no source, it's flagged on the fresh review message. The owner's own edits don't get a free pass on accuracy. The re-composed review message comes back with the updated draft and the three buttons again. Editing can loop as many times as needed; the issue only sends when the owner finally taps *Approve*.

Action 3: skip (the "not this week")

Sometimes the owner reads the draft and decides it can wait. Maybe a bigger announcement is coming next week and they'd rather lead with it. Maybe the items are fine but thin and they'd rather hold for one more. Maybe the timing is wrong — a sober week isn't the moment for a cheerful update. They tap *Skip*.

Skip writes a `skipped` row to `nc-audit` and marks the issue skipped in `nc-issues`. Crucially, it does *not* mark the items as used. They stay fresh in `nc-items-state`, so next week's run counts them again alongside whatever new items arrived. Nothing is lost by skipping — the week's news simply waits and joins the next issue. This is the manual twin of the automatic skip from Part 3: there, the

composer skips because there aren't enough items; here, the owner skips because the timing isn't right. Both are logged the same way, so the trail always explains why an issue did or didn't go out.

Every action is logged, every send is reversible up to a point

The `nc-audit` table records every approve, edit, skip, and pull-back with the user who acted, the timestamp, and a snapshot of the draft before and after. Up until the cooling-off window closes, an approval is fully reversible — the pull-back returns the issue to draft and nothing went out. After the send, the snapshot is your record of exactly what every subscriber received, which is what you'll want the next time someone asks "wait, did we say that?" six months later.

There's no "undo a sent email" button, because there's no such thing — once SES hands the mail to the world, it's gone. That's exactly why the cooling-off window, the grounding check, and the owner's approval all sit in front of the send. The system spends its care before the irreversible step, not after it.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the weekly draft is the one real line item.

PART 6 OF 7

MAY 14, 2026 PART 6 OF 7 · NEWSLETTER COMPOSER SERIES ~3 MIN READ

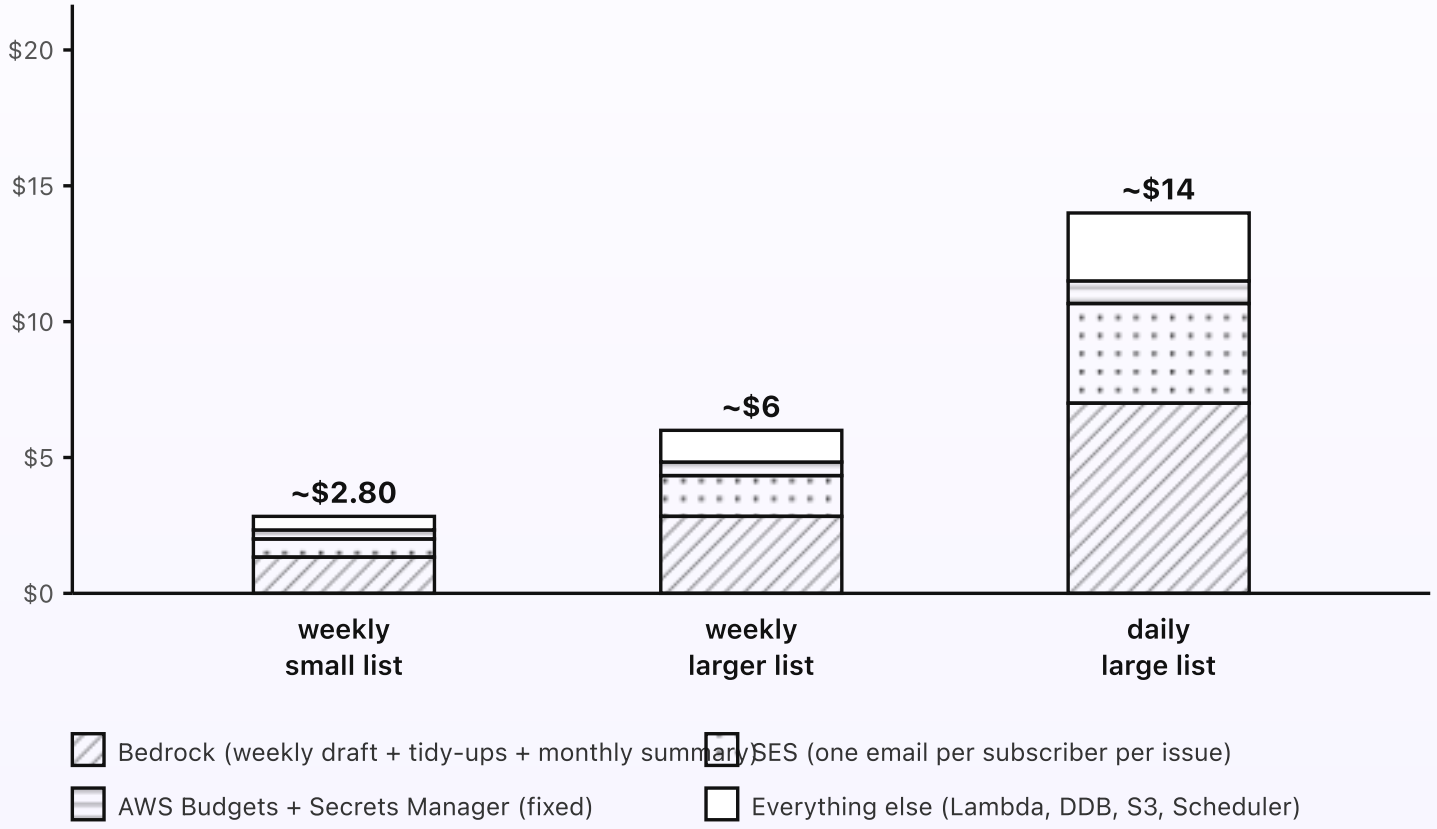
What the newsletter composer costs

The composer is one of the cheaper systems in this whole series. The weekly run reads a CSV from S3, counts a handful of items, makes one Bedrock call to write the issue, and sends one email per subscriber. The gathering and the daily checks are pennies. The two real line items are the weekly draft and the email send. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.80/month for one issue a week to a small list (a few hundred subscribers).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The weekly Bedrock draft is the single biggest line item — and it's one call per issue.
- SES email sends scale with your list size, not with how clever the issue is.
- At one issue a week to a larger list the bill is around $6. At a daily issue it's around $14.

Cost at three volumes



The draft and the send are the two real line items — everything else is rounding error.

Fig 6. Monthly cost at three sending volumes. The Bedrock draft and the SES email send are the two slices that matter; fixed cost and everything-else stay small. More issues a month means more drafts and more sends, which is what grows the bill.

Where the dollars actually go

Bedrock (the draft). The single biggest line item, and still small. Once a week the composer makes one Sonnet 4.6 call to write the whole issue: the prompt is your voice doc plus a dozen short items (a few thousand input tokens), and the output is the finished issue (a thousand or so output tokens). That's a few cents per issue. Add the cheaper Haiku 4.5 calls that tidy forwarded updates in Part 2 (a fraction of a cent each, a handful a month) and the monthly summary (one small call). Weekly, that's pennies a month. A daily issue is seven times the drafts, which is why Bedrock is the tallest slice in the daily bar.

SES (the send). Outbound email is $0.10 per thousand messages. One issue to 500 subscribers is 500 messages — five cents. Four issues a month is twenty cents. The send cost scales with your list size and how often you send, not with anything clever. Even a 10,000-person list sent weekly is four dollars a month in SES. This is the slice that grows with a big list.

Lambda runtime. The weekly composer run, the hourly feed-sync, the every-15-minute drive-sync, the SES inbound parser for forwarded updates, and the Function URL handler for the buttons. None of them run long. The Lambda total lands well under a dollar at all three volumes.

DynamoDB on-demand. Three small tables: `nc-items-state`, `nc-issues`, `nc-audit`. A few reads and writes per run and per action. Pennies a month at any of these volumes.

S3 + Storage. The mirrored item CSV, the voice and rules docs, the stored drafts, and any raw forwarded MIME. A few hundred KB total at SMB volume. Effectively free.

EventBridge Scheduler. The weekly run, the hourly and 15-minute syncs, and the one-off cooling-off and quiet-hours rules. A handful of invocations a day. Pennies.

SES inbound. For the forwarding lane: $0.10 per thousand received messages. A few forwarded updates a month is a fraction of a cent.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the approve, edit, and skip endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The composer sleeps almost the entire week.
- **A separate email platform.** SES sends the issue directly — no per-seat newsletter tool subscription on top.
- **A model on every item.** The issue is one draft call, not one call per item. Item tidy-ups use the cheaper Haiku 4.5, and only on forwarded updates.

How the cost scales

Two things move the bill: how often you send, and how big your list is. Sending more often multiplies the Bedrock drafts; a bigger list multiplies the SES sends. Both scale linearly and both are cheap. A weekly issue to a few hundred people is under three dollars. A weekly issue to a few thousand is around six. A daily issue to a large list — which is more newsletter than most small businesses want to write

— is around fourteen. Past that you're in real publishing territory, and the cost is still dominated by sends, which any email platform charges for too.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The composer'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, SES rule set, and EventBridge Scheduler config.

PART 7 OF 7

MAY 14, 2026 PART 7 OF 7 · NEWSLETTER COMPOSER SERIES ~8 MIN READ

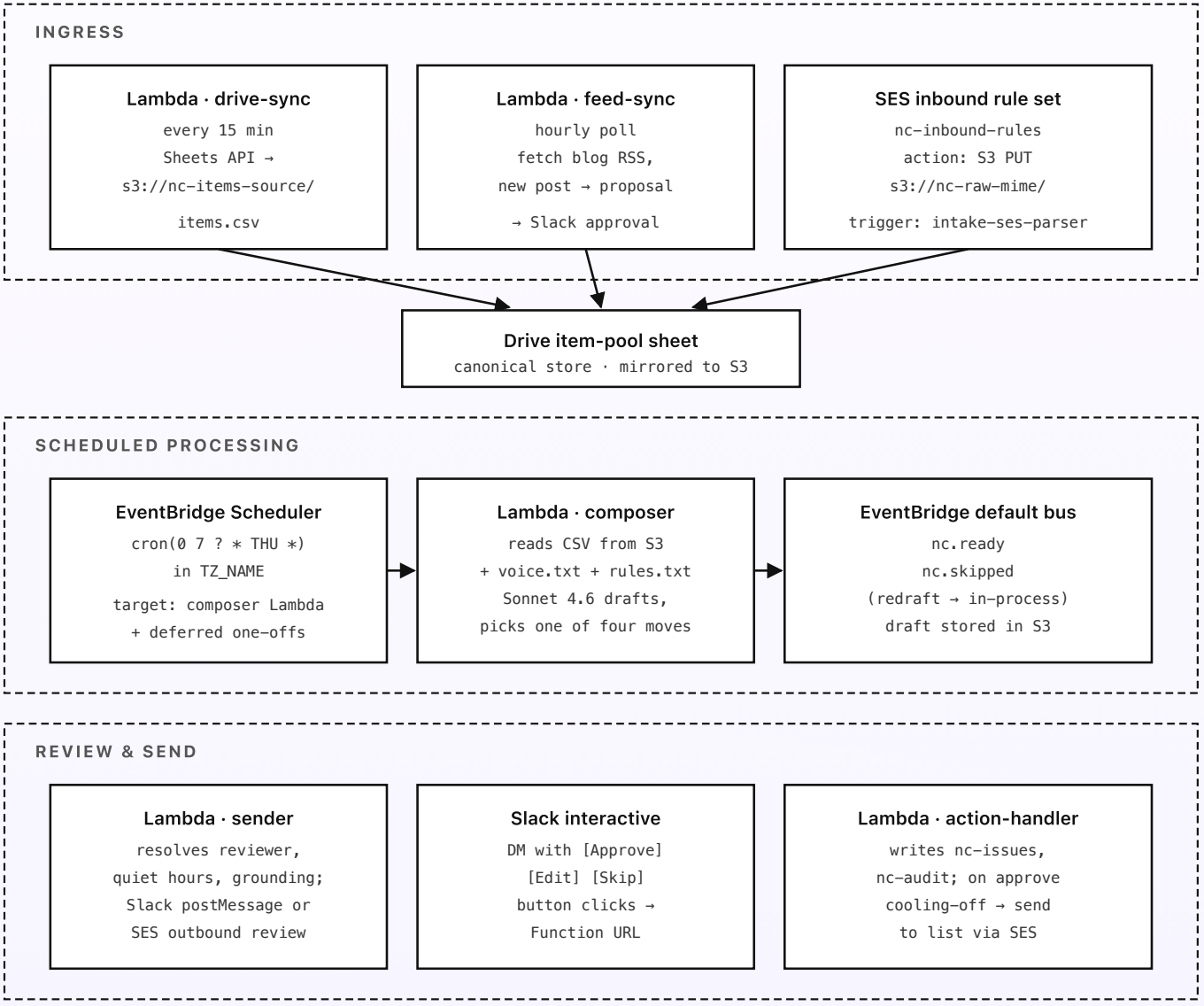
Engineering reference: the newsletter composer architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, EventBridge Scheduler config, the DynamoDB schemas, and the Slack interactive flow. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound and outbound, Bedrock Global cross-Region inference, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup at SMB volume — the failure mode for an SMB is one weekly issue going out a day late, not a regional outage. One AWS account dedicated to the composer (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every issue is drafted only from gathered items — nothing sends without sign-off.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the item pool), scheduled processing (the weekly composer run drafting and emitting events), review and send (the draft is reviewed and only an approved issue is mailed). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

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

- **`drive-sync`** — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `nc/drive/sa`) to export the item-pool sheet as CSV and write to `s3://nc-items-source/items.csv` only if the sheet has changed since the last sync. Same pattern syncs the voice and rules docs to `s3://nc-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`feed-sync`** — EventBridge Scheduler target, hourly. Fetches the configured blog RSS/Atom feed, diffs entry ids against the `nc-feed-seen` DynamoDB table, and for each new entry creates a Slack interactive proposal (title, link, summary). For lower latency you can switch to a WebSub/PubSubHubbub subscription that pushes to a Function URL instead of polling, at the cost of managing the subscription lease. Memory: 256 MB. Timeout: 30 s.
- **`intake-ses-parser`** — S3 PUT trigger on `s3://nc-raw-mime/`. Parses MIME, extracts the text body (and any forwarded-quote text), and calls Bedrock Haiku

4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to tidy the update into a proposed item (title, one-line note, category, link). Posts the proposal to Slack via the bot token with Approve/Edit/Discard buttons. Memory: 512 MB. Timeout: 30 s. *No Sonnet calls.*

- **`composer`** — EventBridge Scheduler target, weekly the morning before the send day (the schedule runs in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). Reads `s3://nc-items-source/items.csv`, the voice and rules docs, and `nc-items-state` to determine fresh items. If the fresh count is below threshold, emits `nc.skipped`. Otherwise calls Bedrock Sonnet 4.6 (`anthropic.claude-sonnet-4-6-20250115-v1:0` via the Global cross-Region profile) once to draft the issue grounded in the items, runs the self-check, stores the draft to `s3://nc-drafts/<issue-id>.json`, and emits `nc.ready`. Memory: 1024 MB. Timeout: 120 s.
- **`sender`** — EventBridge rule on `nc.ready`. Resolves the reviewer, checks quiet hours, runs the grounding check (attaches the source item behind each paragraph, flags any unsourced line), formats the review message from the draft, and ships via Slack `chat.postMessage` (`nc/slack/bot-token` in Secrets Manager) or SES `SendRawEmail`. On a quiet-hours defer, creates a one-off EventBridge Scheduler rule that re-invokes `sender` at the next business minute. Writes a row to `nc-issues` marking the issue awaiting review. Memory: 256 MB. Timeout: 30 s. *No Bedrock calls.*
- **`action-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive button clicks (Approve/Edit/Skip, plus Pull-it-back) and by email-link clicks. Writes to `nc-issues` and `nc-audit`. On *edit*, copies the draft to a Google Doc

and replies with the link; on re-submit, re-runs the grounding check. On *approve*, creates a cooling-off one-off EventBridge Scheduler rule (default 30 min) targeting `send-issue`. Memory: 256 MB. Timeout: 15 s.

- **`send-issue`** — EventBridge Scheduler target, fired once by the cooling-off rule. Renders the approved issue, sends it to the subscriber list via SES `SendBulkEmail` in batches that respect the account's send rate, marks every used item as `used` in `nc-items-state`, and writes a `sent` row to `nc-audit` with the recipient count and a snapshot of exactly what went out. Memory: 512 MB. Timeout: 120 s.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `nc-issues` and `nc-audit` (issues sent, skips, open rate if a webhook feeds it back); calls Bedrock Haiku 4.5 to write a short narrative; emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB · `nc-items-state`** — one row per item. PK `item_id`; attributes: `used_in_issue`, `used_date`, `category`, `added_by`. On-demand. Drives the "fresh" count.
- **DynamoDB · `nc-issues`** — one row per weekly run. PK `issue_id`; attributes: `move` (skip/draft/redraft/ready/approved/sent/skipped-by-owner), `item_ids`, `draft_s3_key`, `reviewer`, `state`. On-demand.
- **DynamoDB · `nc-audit`** — one row per write action of any kind. PK `(issue_id, ts)`; attributes: `action` (approve/edit/skip/pull-back/sent), `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.

- **DynamoDB** · `nc-feed-seen` — one row per blog feed entry already seen. PK `entry_id`; attribute: `first_seen`. On-demand. TTL at 180 days.
- **S3** · `nc-items-source` — mirrored CSV from the Drive item-pool sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3** · `nc-rules-source` — mirrored voice and rules docs as plain text. Versioning enabled.
- **S3** · `nc-drafts` — each issue draft as JSON (subject, body, per-paragraph sources). Versioning enabled, so an edited issue keeps its draft history.
- **S3** · `nc-raw-mime` — raw inbound MIME from forwarded updates. Lifecycle to Glacier at 30 days; expiry at 7 years.

Bedrock

- **Foundation models.** `anthropic.claude-sonnet-4-6-20250115-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-sonnet-4-6-20250115-v1:0` for the one weekly issue draft (the only genuinely hard writing task). `anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0` for the forwarded-update tidy-ups in `intake-ses-parser` and the monthly `summary`.
- **Embeddings.** Not used. The item pool is a short structured list passed in full to the draft prompt; there's nothing to retrieve. No Knowledge Base, no S3 Vectors. (If a future version drew on a large archive of past issues for style, Titan Text Embeddings V2 at 1024-dim into S3 Vectors would be the path.)
- **Quotas.** Default account quotas are more than enough at SMB volume. The composer fires one Sonnet call per issue; Haiku fires a few times a month.

EventBridge Scheduler config

- `nc-weekly-compose` — `cron(0 7 ? * THU *)` in the SMB's timezone (Thursday 7am, the morning before a Friday send). Target: `composer` Lambda.
- `nc-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `nc-feed-sync` — `rate(1 hour)`. Target: `feed-sync` Lambda.
- `nc-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `action-handler` for the cooling-off window, and by `sender` for quiet-hours defers. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `updates.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `nc-inbound-rules`: one rule with recipient `updates@your-company.com` → spam scan → S3 PUT to `s3://nc-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the review messages and the issue sends: verify a sender identity at `news@your-company.com` with DKIM and SPF on the parent domain, plus a custom MAIL FROM and a List-Unsubscribe header on every issue. Out of sandbox by request; a configuration set with event publishing feeds bounces and complaints back to an SNS topic so a hard bounce removes a subscriber.

IAM (least privilege per Lambda)

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

- **composer role:** `s3:GetObject` on the items, voice, and rules keys; `s3:PutObject` on `nc-drafts`; `dynamodb:Query` + `GetItem` on `nc-items-state`; `bedrock:InvokeModel` on the Sonnet ARN; `events:PutEvents` on the default bus.
- **sender role:** `s3:GetObject` on `nc-drafts`; `events:CreateSchedule` for quiet-hours one-offs; `secretsmanager:GetSecretValue` on the Slack bot token; `ses:SendRawEmail` from the verified sender; `dynamodb:PutItem` on `nc-issues`; outbound network access to `slack.com`.
- **action-handler role:** `dynamodb:PutItem` on `nc-issues` and `nc-audit`; `events:CreateSchedule` for the cooling-off one-off; `secretsmanager:GetSecretValue` on the Sheets and Slack secrets; outbound network access to `sheets.googleapis.com` and `slack.com`.
- **send-issue role:** `s3:GetObject` on `nc-drafts`; `ses:SendBulkEmail` from the verified sender; `dynamodb:UpdateItem` on `nc-items-state`; `dynamodb:PutItem` on `nc-audit`; `secretsmanager:GetSecretValue` on the subscriber-list secret if the list lives outside DynamoDB.
- **intake-ses-parser role:** `s3:GetObject` on `nc-raw-mime`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token.
- **drive-sync and feed-sync roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the items and rules buckets;

`dynamodb:*Item` on `nc-feed-seen` (feed-sync only); outbound network to `www.googleapis.com` and the blog host.

Slack interactive flow

Review messages and item proposals are posted via the `chat.postMessage` Web API with Block Kit blocks containing the action buttons, because the simpler incoming-webhook surface doesn't support interactive responses. Button clicks are sent by Slack to the configured Interactivity request URL, which is the `action-handler` Function URL. `action-handler` verifies the Slack signing secret on the inbound request, parses the `action_id` (`approve`, `edit`, `skip`, `pull_back`, and the proposal actions `item_approve`, `item_edit`, `item_discard`), opens a modal where needed, and processes the response on submit.

The Slack app needs `chat:write` and `im:write`, plus the Interactivity URL configured. The bot token lives in Secrets Manager under `nc/slack/bot-token`. The signing secret is `nc/slack/signing-secret`.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch metric for alerting.
- **Alarms:** composer Lambda failures > 0 on a run day (the weekly run is the piece that has to work); send-issue failure rate > 1% in a send; action-handler

signature-verification failures > 5/hour (might mean the Slack secret rotated); SES complaint rate above the SES threshold.

- **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 `nc-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive, Sheets, and Docs APIs all live in Secrets Manager under `nc/drive/sa` (one service account with scopes for all three APIs). Slack bot token and signing secret under `nc/slack/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, item threshold, send day, reviewer mapping, quiet-hours window, cooling-off minutes, and admin fallback all live in Parameter Store under `/nc/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) and AWS SAM. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for `nc-items-source`, `nc-rules-source`, and `nc-drafts` so a bad edit can be rolled back in one click, and version the EventBridge Scheduler timezone setting so you don't accidentally start drafting in UTC after a CI rotation. Keep the cooling-off window and the subscriber-list handling in their own reviewable config so a change to either is a visible PR. Total deployable surface: around eight Lambdas, four DDB tables, four

S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SES rule set, and one Budgets alarm.

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