7-PART SERIES · FREE COMPANION

Content moderator

A serverless moderator that keeps a community page or comment section clean without a full-time moderator. Every new comment, review, or post is checked against your house rules; obvious-safe ones pass, clearly-against-the-rules ones are held for review (never auto-deleted), and borderline ones go to a human with the reason and the rule they may break. It explains its calls and learns from a correction. A person makes the final call on anything removed. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Content moderator

PART 1 OF 7

JUNE 17, 2026 PART 1 OF 7 · CONTENT MODERATOR SERIES ~5 MIN READ

A content moderator on AWS for a few dollars a month

A community page is only as good as the worst comment on it. The spam link that goes up overnight and is the first thing a prospect sees. The personal attack under your most-shared post. The off-topic rant that drowns out a real question. Most small businesses can't afford a full-time moderator, so the job falls to whoever happens to check the page — which means it gets done late, or not at all. This post walks through the design of a small moderator that checks every new comment, review, or post against your own house rules, lets the obvious-safe ones through, holds the clear rule-breakers for review without ever deleting them, and sends the borderline ones to a person with the reason already attached.

KEY TAKEAWAYS

- Every comment, review, or post arrives through one webhook and is checked the same way.
- Every item ends in one of three calls: pass, hold for review, or send to a human.
- A fast rule pass settles the easy cases; only the borderline middle goes to the model.
- Nothing is ever auto-deleted. Holding is reversible; removal is always a human's call.
- Designed on AWS for about $2/month at typical small-business volume.

The whole system on one page

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

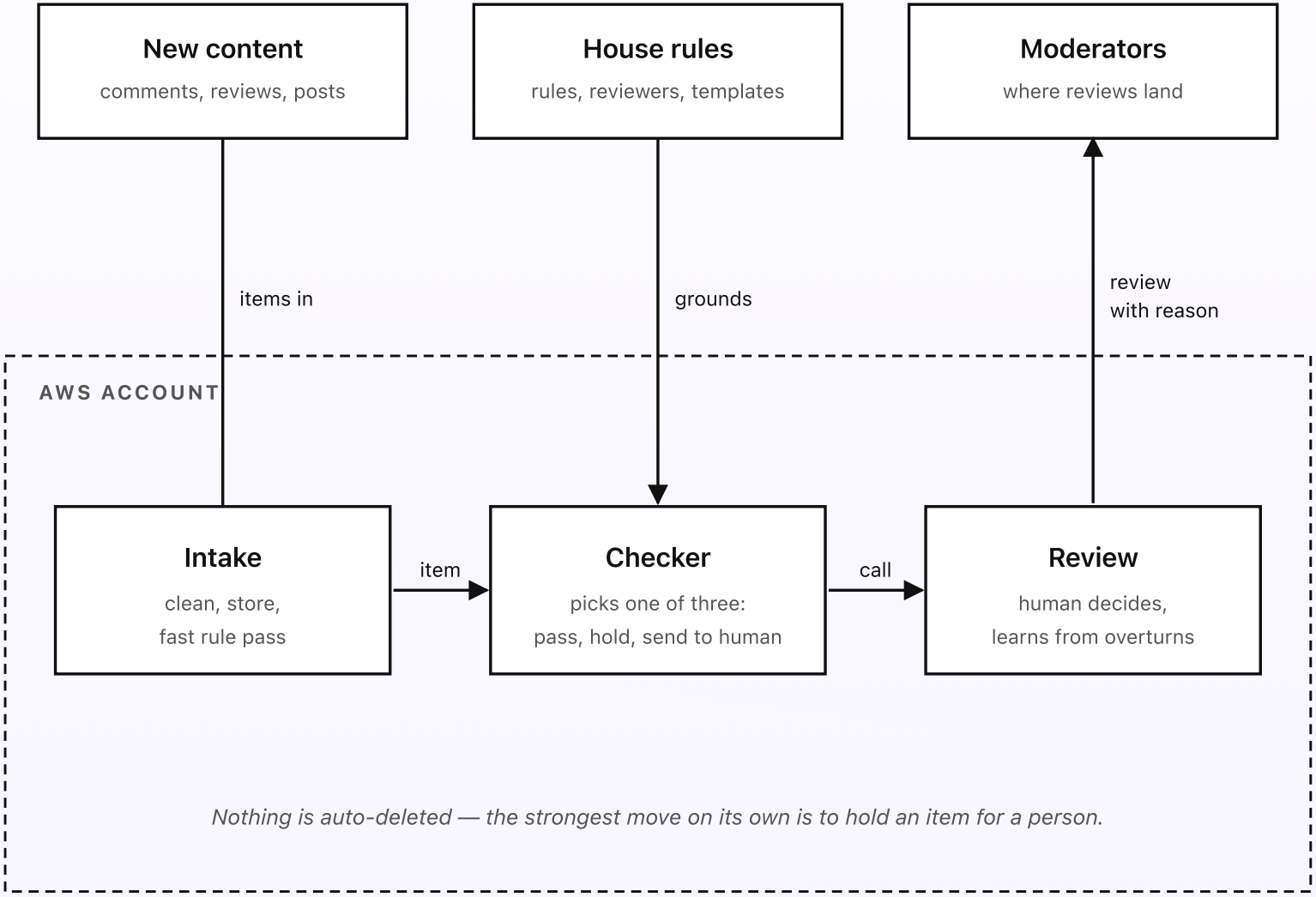

New content
comments, reviews, posts
House rules
rules, reviewers, templates
Moderators
where reviews land
items in
grounds
review
with reason
AWS ACCOUNT
Intake
clean, store,
fast rule pass
item
Checker
picks one of three:
pass, hold, send to human
call
Review
human decides,
learns from overturns
Nothing is auto-deleted — the strongest move on its own is to hold an item for a person.

Fig 1. Three sources outside, three pieces inside AWS. Items flow in from your community page, comment section, and review platforms. The Checker picks one of three calls. Review puts anything risky in front of a person before it comes down.

What you set up once (the outside)

- **New content.** Wherever people post — your community page, the comments under your posts, your review pages — each new item is sent to the moderator by a webhook (a small message the platform fires whenever something new appears). One item, one message: the author, the text, a link back to where it lives, and which area it belongs to. You set this up once per platform and then forget it.
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc holds your house rules in plain words — the banned words and link rules, what counts as spam, what counts as a personal attack, what's simply off-topic, and which of those mean "hold for review" versus "just flag it." It also names the reviewer for each area and the quiet hours. The *voice* doc holds the review-card wording and any auto-reply templates — what the moderator's message to the poster actually says when an item is removed.
- **Moderators.** The people who own each area. Each has a Slack member ID (so the review card is a private message, not a public ping) or, if Slack isn't set up for them, an email address. Cards land with the original item, the proposed call, the confidence, the exact rule the content may break, a link to where it lives, and buttons to publish, remove, or edit-and-publish.

What runs on every item (the inside)

- **The intake.** A Function URL (a plain web address that runs a small piece of code — no heavy gateway in front) receives the webhook, cleans the text, and stores the item. Then a fast rule pass runs in plain Python: is the author on the known-good list, does the text trip a banned word, does it carry a link from a blocked domain, is it absurdly short or long? Most items are settled right here, with no model. Clean ones pass. Obvious spam is held.
- **The checker.** Only the borderline middle — the items the rule pass can't settle on its own — goes to Bedrock Haiku 4.5. The model reads the item against the house rules and returns one of three calls. *Pass*: it's fine, publish it. *Hold*: it clearly breaks a rule — keep it out of public view and queue it for review, but don't delete it. *Send to a human*: it's genuinely borderline — queue it with the reason and the rule so a person can decide. The model always returns a confidence score and the exact rule it leaned on. It never deletes anything.
- **The review.** Held and flagged items go into a review queue. The right moderator gets a card — in Slack, or email as a fallback — with the item and the proposed call already explained. They publish, remove, or edit-and-publish. Every call is logged. When a moderator overturns the system — publishes something it held, or removes something it passed — that correction is saved as a worked example so the next similar item gets the better call. A weekly digest summarizes what was held, removed, and overturned.

In plain words

A comment lands under your most-shared post at 11pm: "Great tips — check out cheap-deals dot info for even better prices!!!" The rule pass sees a link from a domain on your blocked list and holds the item before anyone sees it. It never

goes public. The next morning your community lead, Sam, gets one Slack card: “Held — outbound link to a blocked domain (rule: no promotional links from unknown sites). Confidence high. *[Publish] [Remove] [Edit & publish]*.” Sam taps Remove; the poster gets the polite house reply from the voice doc, and the removal is logged with the reason. Meanwhile a second comment — “this is the dumbest thing I’ve read all week” — isn’t a clear rule break, so the model sends it to a human as borderline. Sam reads it, decides it’s rude but within the line, and taps Publish. The system remembers both calls.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the spam link that greets your next prospect, or the pile-on under a post that you only notice after it’s done its damage, or the real question that nobody answered because it was buried under noise.

DESIGN RULES THAT SHAPED EVERY DECISION

- Three calls, always. Pass, hold, or send to a human. There is no fourth.
- Nothing is auto-deleted. Holding keeps an item out of view but reversible; removal is a person's call.
- Every flagged item ships with the reason and the exact rule. The moderator never has to guess.
- The cheap rule pass runs first; the model only reads the borderline middle.
- The house rules live in Drive. Changing a rule doesn't need a deploy.
- Every call is logged, and every overturn teaches the system the better answer.

Why this shape

Most small teams moderate in one of three ways: a blunt keyword filter that blocks half the real comments and misses the clever spam, a platform's built-in tool that can't read your house rules, or a person who checks the page when they remember. The keyword filter is too dumb to trust and too noisy to leave on. The built-in tool doesn't know what "off-topic" means for your business. And the person, of course, isn't watching at 11pm when the spam goes up.

The setup above keeps the house rules in a doc the team already edits, but adds a small system that checks every new item against that doc the moment it arrives. The cheap rule pass handles the easy nine items out of ten for almost nothing.

The model reads only the hard one. Nothing comes down without a person agreeing it should. And every time a moderator disagrees with a call, the system gets a little better at the next one. The moderator is invisible most of the time; visible only on the items that actually need a human.

The next four posts walk through each piece in turn: how a comment gets checked, how a comment gets a verdict, how flagged content reaches a moderator, and how the moderator makes the final call. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 17, 2026 PART 2 OF 7 · CONTENT MODERATOR SERIES ~4 MIN READ

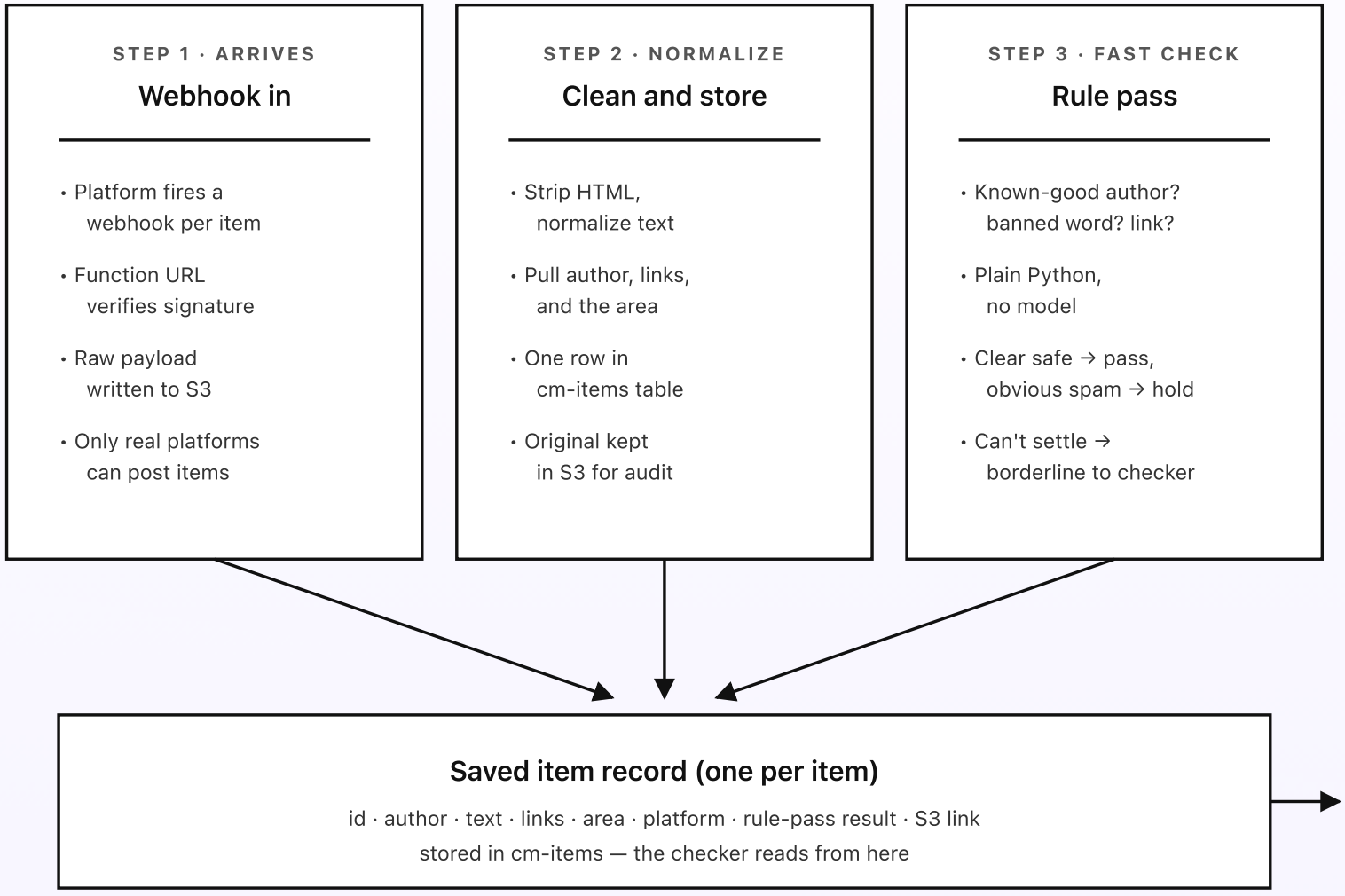
How a comment gets checked

The moderator only checks what reaches it. So the first job is making sure every new comment, review, or post actually arrives — and that the easy calls get made before any model is asked to look. There are three steps an item passes through before the checker ever sees it: it comes in through a webhook, it gets cleaned and stored, and it runs through a fast rule pass. The rule pass settles most items on its own. Only what it can't settle moves on.

KEY TAKEAWAYS

- One webhook brings in every item — comments, reviews, and posts arrive the same way.
- Each item is cleaned and stored once, so there is exactly one record per item.
- A fast rule pass in plain Python checks banned words, links, length, and known-good authors.
- Most items are settled by the rule pass with no model call at all.
- Only the borderline middle moves on to the checker in Part 3.

Three steps before the checker



The rule pass settles most items for almost nothing — only the borderline middle costs a model call.

Fig 2. Three steps converge on one saved item. The webhook brings it in, the clean-and-store step makes exactly one record, and the rule pass settles the easy calls. Only the borderline middle moves on to the checker.

Step 1: the webhook brings it in

Every platform that lets people post — your community page, the comment plugin under your blog, your review pages — can fire a *webhook*: a small message sent to a web address you control whenever something new appears. You point each platform's webhook at one Lambda Function URL. A Function URL is just a plain web address that runs a small piece of code; there's no heavy gateway in front of it, which keeps the cost near zero. The first thing that code does is check the signature on the message, so only your real platforms can post items — not a random script that found the address. The raw payload is written to S3 exactly as received, so there's always an untouched copy to fall back on.

One item, one message. A comment, a review, and a post all look the same once they're in: an author, some text, maybe a link or two, and a note about which area of your site they belong to.

Step 2: clean and store, exactly once

Raw webhook payloads are messy — HTML tags, tracking junk, odd spacing, the same item sometimes delivered twice. A small Lambda strips the HTML, normalizes the text, and pulls out the parts that matter: the author, the list of links, the length, and the area. It writes one record to the `cm-items` table in DynamoDB, keyed by a stable item id from the platform. If the same item arrives twice (platforms sometimes retry), the second write lands on the same key and nothing

is duplicated. The original payload stays in S3, so if anyone ever asks “what exactly did this person post?” the answer is one click away.

From here on, the rest of the system works off that one clean record. There is exactly one row per item, and any moderator can read it without learning a new tool.

Step 3: the fast rule pass

Before any model is asked to read anything, a fast check runs in plain Python against the house-rules lists. Is the author on the known-good allow list — a long-time member, a verified customer, a teammate? Then pass it; trusted people don’t need a model reading their every word. Does the text trip a banned word from your list? Does it carry a link from a domain you’ve blocked? Is it empty, or a wall of ten thousand characters that’s obviously a paste-bomb? Each of those is a clear signal.

The rule pass ends with one of three labels. *Pass* — clearly safe, publish it. *Hold* — obvious spam or a blocked link, keep it out of view and queue it (but never delete it). *Borderline* — the rules can’t settle it, so hand it to the checker. The whole pass is a few dozen lines of code and costs effectively nothing. At typical volume it settles most items right here, which is the whole point: the model should only ever read the items a simple rule genuinely can’t.

Why everything funnels to one record

Three steps in, but only one place the rest of the system looks: the saved item record. That’s a deliberate constraint. If the webhook code, the cleaner, and the

rule pass each kept their own copy of the truth, every “why was this held?” question would mean checking three places. Funneling everything through one record means there is exactly one row per item, carrying its rule-pass result forward, and the checker in Part 3 reads from that and nothing else.

Next post: how the borderline middle gets a verdict — how the checker asks the model, what it gets back, and how it lands on one of three calls.

PART 3 OF 7

JUNE 17, 2026 PART 3 OF 7 · CONTENT MODERATOR SERIES ~5 MIN READ

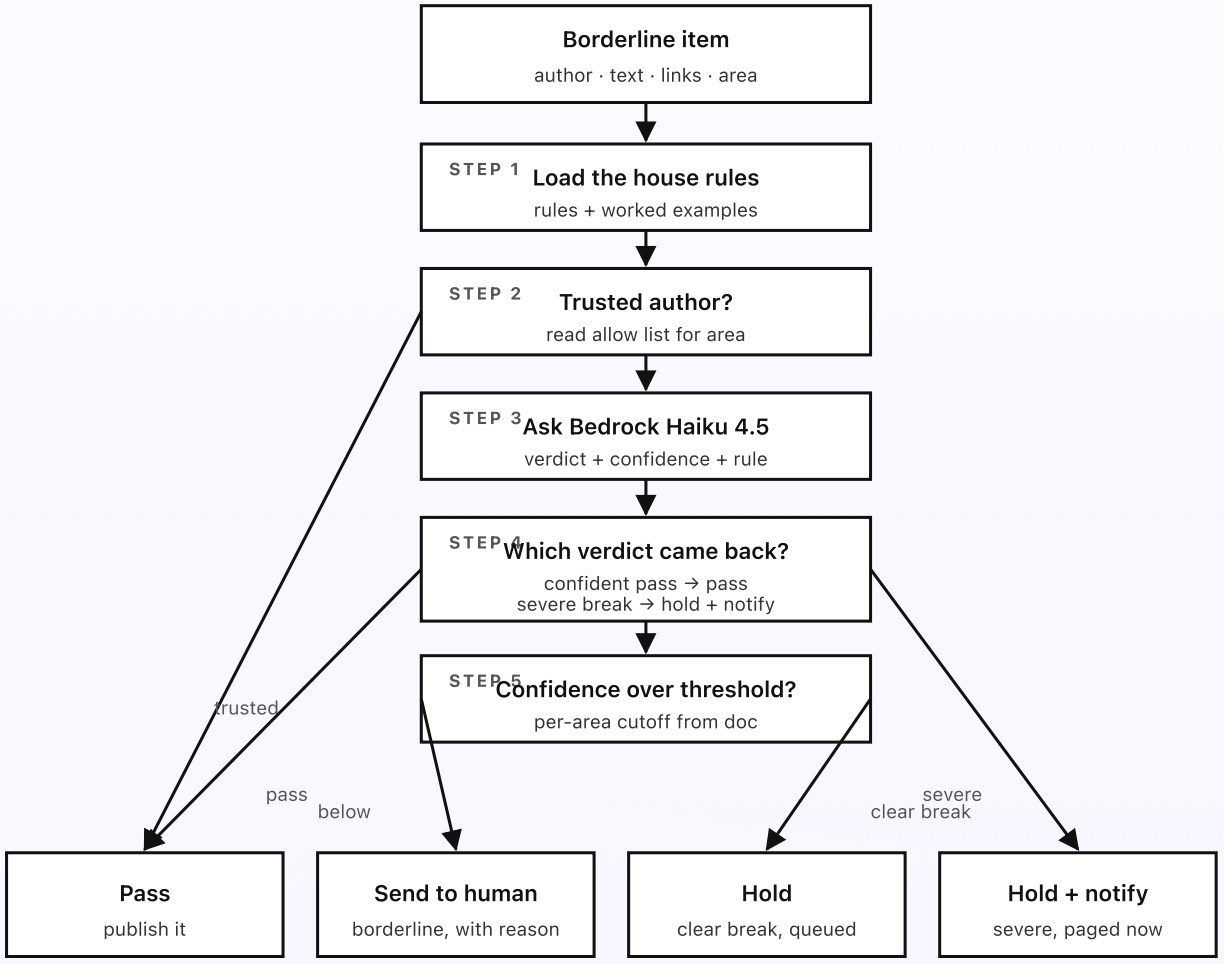
How a comment gets a verdict

The rule pass from Part 2 already settled the easy items. What's left is the borderline middle — the things a simple word list can't judge. Is "this is garbage" a rule-breaking attack or just a blunt opinion? Is a link to a real article spam, or useful? For those, the checker asks Bedrock Haiku 4.5 to read the item against your house rules and return a verdict. The decision is a few clear steps, and the model always says why and how sure it is.

KEY TAKEAWAYS

- Only borderline items reach the model — the rule pass already settled the rest.
- House rules live in the Drive doc; a rep can edit them without a deploy.
- The model returns a verdict, a confidence score, and the exact rule the content may break.
- Low confidence never auto-acts — it routes the item to a human instead.
- The model never deletes anything. Its strongest call is "hold for review."

The decision flow, per item



The rules doc holds every rule and threshold — change a rule and the next item uses the new wording.

Fig 3. The checker's decision tree, per borderline item. A trusted author short-circuits to pass; everyone else gets a model verdict with confidence and a rule. Low confidence routes to a human. The rules doc holds every rule; the model only applies them.

House rules: "no spam, no attacks" isn't magic, it's in the doc

The rules doc has one short section per area of your site. Each section names the rules in plain prose: "Comments: no promotional links from unknown sites, no personal attacks on other members, no off-topic reselling. Reviews: opinions are fine even if harsh, but no naming staff, no made-up claims. Posts: no recruiting, no spam." Each rule says what it means and whether breaking it is a *hold* (clear) or a *send-to-human* (judgment call). The model is handed this doc verbatim; it doesn't carry its own idea of what your community allows.

Rules differ by area for a reason. A harsh review is a customer being honest — you want it. The same words aimed at another member under a post are an attack — you don't. The doc lets you say so, in your own words, and a rep can change the wording any time without a developer. Each section also sets a *confidence threshold* — how sure the model has to be before its call is acted on automatically rather than sent to a person.

What the model actually returns

The prompt to Bedrock Haiku 4.5 is short and strict: "Here is a comment, here are the house rules for this area, and here are a few past calls a human corrected. Return JSON only. Give a verdict of pass, hold, or send-to-human; a confidence

score from 0 to 1; and the exact rule the content may break, quoted from the doc. Do not invent a rule that isn't there. If you are unsure, return send-to-human."

That last line matters. The model is told that "I'm not sure" is a perfectly good answer, and unsure means a person looks. There is no pressure to force a confident call. The confidence score is then checked against the per-area threshold: a confident pass publishes, a confident hold queues the item, and anything below the line goes to a human regardless of which way it leaned. The model citing the exact rule is what makes the review card in Part 4 useful — the moderator sees not just "held" but "held because of *this* rule."

The calls, and why "hold + notify" exists

Most items land in one of three calls: **pass** (publish), **hold** (a clear break — keep it out of view and add it to the review queue for the next batch), or **send to a human** (borderline — queue it with the reason). There's a fourth, rarer terminal for severity: **hold + notify**. A clear, severe break — a credible threat, a doxxing post, hate speech — is still held rather than deleted, but a moderator is paged at once instead of waiting for the next batched digest. Holding the item keeps the decision reversible; the immediate page just gets a human looking sooner. Even here, no content comes down without a person.

Why the model doesn't read everything

The checker could send every item to the model and skip the rule pass entirely. It doesn't, for two reasons. First, the rule pass is free and instant, and most items are clearly safe or clearly spam — spending a model call on them is waste.

Second, the items a simple rule can settle are exactly the items where a model adds nothing; the model earns its place only on the genuinely ambiguous middle. Keeping the cheap check in front means the bill stays a couple of dollars a month even as the page gets busy.

Next post: how a held or flagged item actually reaches a moderator — who gets which area, how quiet hours and batching keep the pings sane, and the four guardrails on every review card.

PART 4 OF 7

JUNE 17, 2026 PART 4 OF 7 · CONTENT MODERATOR SERIES ~5 MIN READ

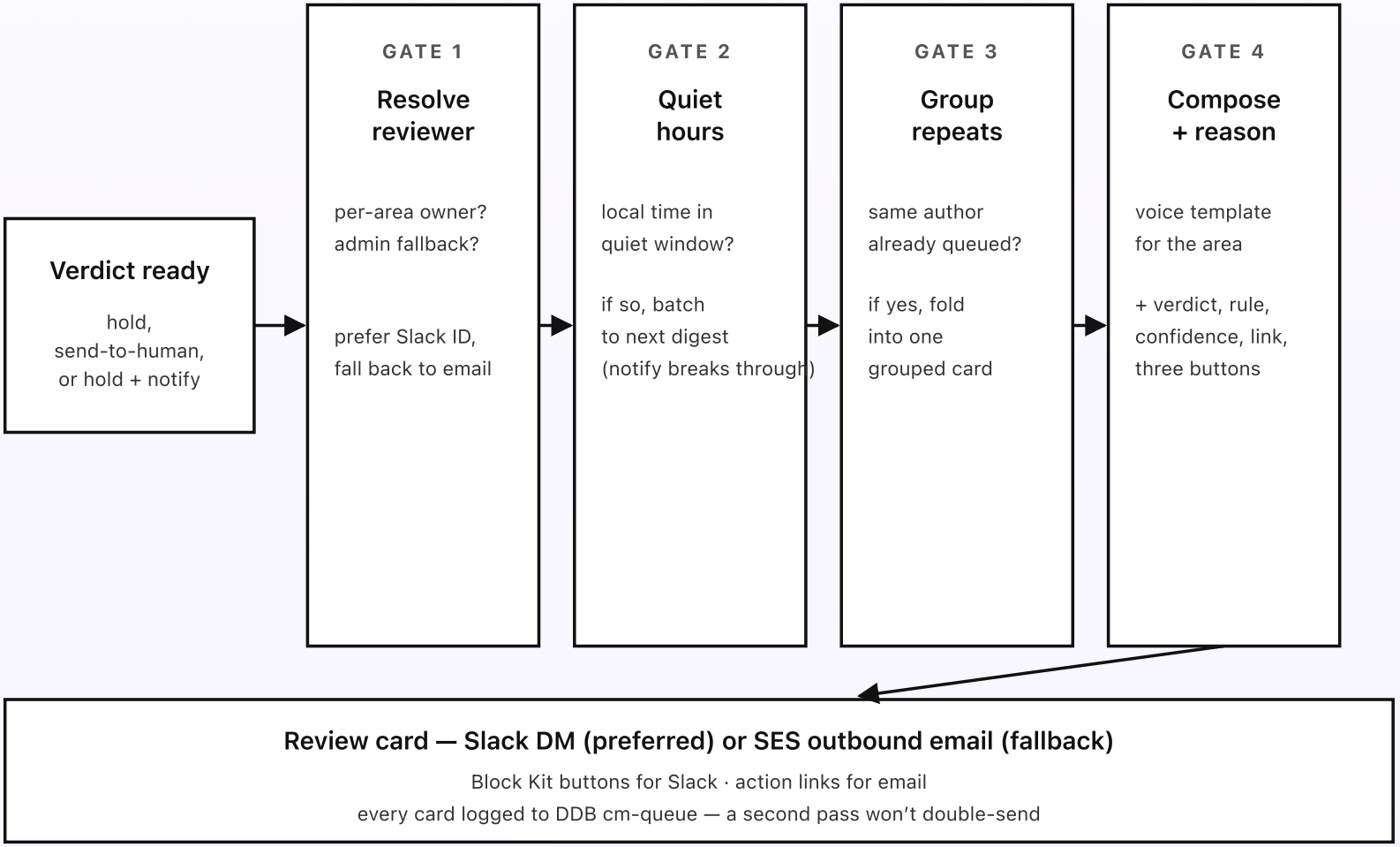
How flagged content reaches a moderator

The checker held an item or marked it borderline. Now something has to get it in front of a person — the right person, on the right channel, at a sane time, with the reason already attached. Get any of those wrong and the review is worse than none: a 2am ping, a card that doesn't say why, fifty separate alerts for one spammer's fifty comments. Four small guardrails sit between the verdict and the card landing.

KEY TAKEAWAYS

- Held and borderline items go into a review queue (an SQS queue with a dead-letter backstop).
- Reviewer resolution: per-area owner beats the configured admin fallback.
- Quiet hours batch normal items into a digest; only “hold + notify” pages at once.
- Repeat offenders are grouped into one card, so one spammer doesn’t mean fifty pings.
- Every card ships with the item, the verdict, the confidence, the rule, and three buttons.

Four guardrails on every review card



Every gate is a deterministic check — no model calls, no surprise pings at midnight.

Fig 4. Four guardrails between the verdict and the review card. Resolve the reviewer. Honor quiet hours. Group repeat offenders. Compose with the full reason. Then ship via Slack or email and log the card so the queue doesn't double-send.

The review queue

Held and borderline items don't go straight to a person; they go into a queue first. The queue is an SQS queue — a simple, durable line of work that holds items until something processes them, so nothing is lost if a delivery hiccups. Behind it sits a dead-letter queue: if an item somehow fails to send a few times in a row, it drops into that backstop instead of vanishing or looping forever, and the weekly digest flags it. A held item is out of public view the moment the verdict lands; the queue just decides *when* and *how* the card reaches a moderator.

Gate 1: resolve the reviewer

Two places the dispatch looks for who owns an item, in order. First, the per-area reviewer in the rules doc — "all review-page items go to Priya, all community-post items go to Sam." Second, the configured admin fallback, the person who set the system up and gets anything unowned. The fallback should rarely fire; when it does, the weekly digest names every item that hit it so the rules doc can be fixed.

Once the reviewer is known, the dispatch looks up how to reach them. The voice doc maps each reviewer to a Slack member ID if one is set, otherwise to an email address. Slack is preferred — a private message with action buttons is faster to act on than an email link — and email is the fallback so nobody is left out.

Gate 2: quiet hours and batching

Items arrive at all hours, but a moderator shouldn't. Gate 2 reads the rules doc's quiet-hours setting (default 8pm to 8am, configurable). During quiet hours, a normal hold or borderline item isn't pinged — it's batched into the next digest, a single grouped message that goes out when business hours start. Since a held item is already out of public view, nothing is harmed by waiting a few hours for a person to confirm.

The one exception is a *hold + notify* — the severe call from Part 3. Those break through quiet hours and page the on-call reviewer at once, because a credible threat or a doxxing post is the rare case where waiting until morning is the wrong trade. Everything else respects the window.

Gate 3: group repeat offenders

Spam comes in bursts. One bot can drop fifty near-identical comments in a minute. Fifty separate review cards would bury the moderator and make the real borderline items impossible to find. Gate 3 checks whether the same author already has open items in the queue; if so, the new item is folded into one grouped card — "this author has 50 held comments, all matching the same blocked-link rule" — with a single set of buttons that can publish or remove the whole group at once. One spammer, one card.

Gate 4: compose with the full reason, then ship

The voice doc has one review-card template per area. The dispatch fills it with the original text, the verdict, the confidence, the exact rule the checker cited, and a link to where the item lives. It attaches three buttons — *Publish*, *Remove*, *Edit & publish* — and ships the card via Slack's Block Kit (the format Slack uses for messages with buttons). For email fallback, the same fields are wrapped in a small HTML email whose buttons are action links that hit a Function URL.

Every card — Slack or email, single or grouped — writes a row to `cm-queue` in DynamoDB marking that it was sent. A second pass over the queue reads that row and knows not to send the same card twice.

Why the guardrails exist

None of these gates are clever. They're the small care a thoughtful person would take if they were handing off the flags themselves — send it to whoever actually owns this area, don't ping at midnight, don't fire fifty alerts for one bot, and include enough that the reviewer doesn't have to go dig for the reason. Putting them in code as four small gates makes them part of the design, not something you're hoping the writer of any one alert remembers.

Next post: what happens when the moderator taps a button — publish, remove, or edit-and-publish — how the item comes down (or goes up), and how every overturn teaches the system the better call.

PART 5 OF 7

JUNE 17, 2026 PART 5 OF 7 · CONTENT MODERATOR SERIES ~5 MIN READ

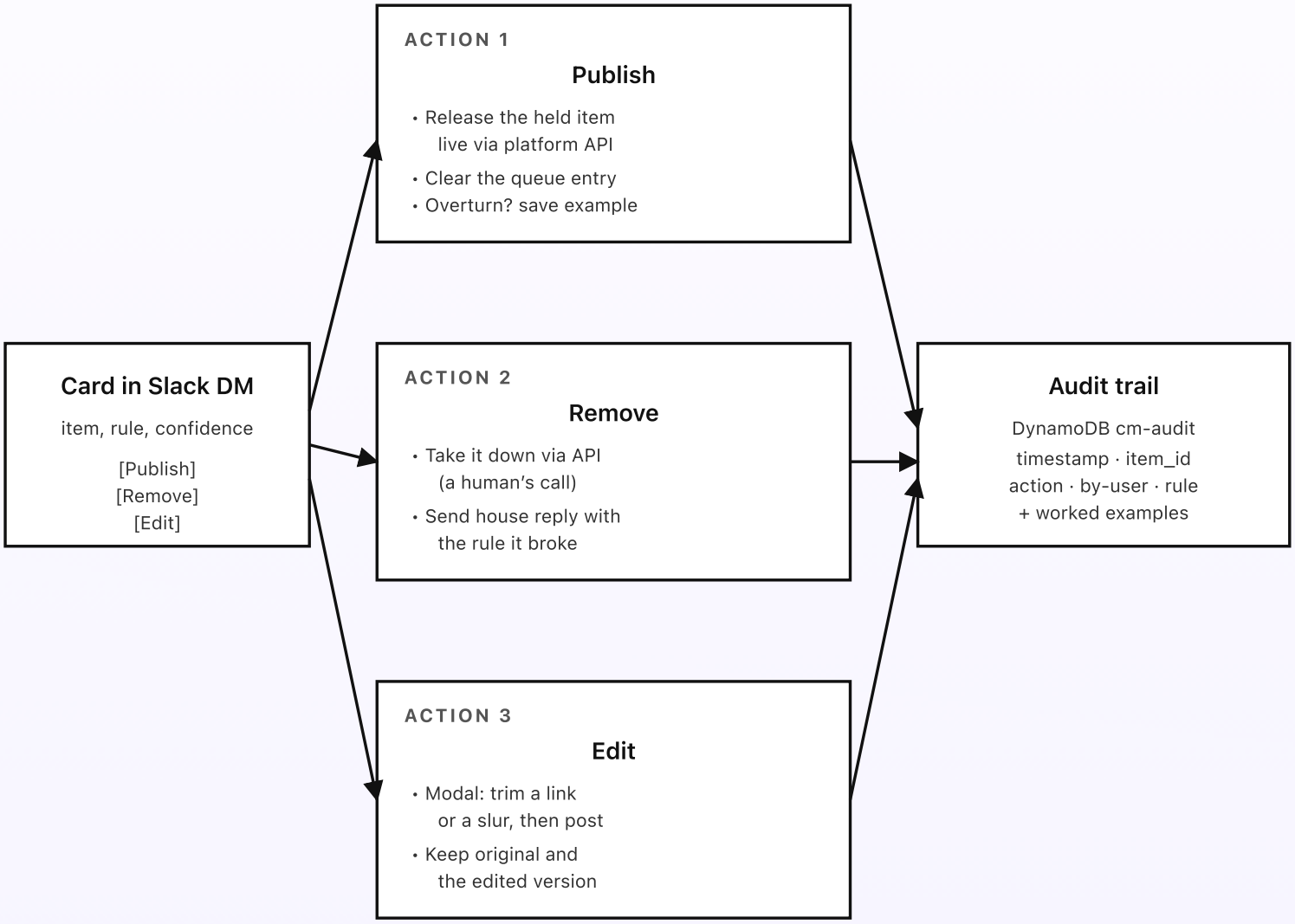
How the moderator makes the final call

A review card lands in Sam’s Slack at 8:03am. A held comment, a reason, the exact rule it may break, three buttons. What happens when Sam taps one? The honest answer is “it depends which button.” This post walks through the three things a moderator can do — publish, remove, or edit-and-publish — how the item, the queue, and the audit trail stay in sync, and how a moderator’s correction teaches the system the better call for next time.

KEY TAKEAWAYS

- Three actions per card: *publish* (release a held item), *remove* (take it down), *edit-and-publish*.
- Removal is the only action that takes content down — and it is always a human's call.
- Each action updates the item, clears it from the queue, and writes an audit row.
- An overturn — publishing a held item, or removing a passed one — becomes a worked example.
- Those examples are fed back into the model prompt so the next similar item gets the better call.

Three actions on the review card



Removal is the only action that takes content down — and it is always a person's decision.

Fig 5. Three actions per card, three different effects. Publish releases a held item. Remove takes it down with the reason logged. Edit-and-publish cleans and posts. Every action writes to the audit trail, and every overturn becomes a worked example.

Action 1: publish (the “this is fine”)

Sam reads the held comment — “this is the dumbest thing I’ve read all week” — and decides it’s rude but within the line. Sam taps *Publish*. The button submits to a Function URL Lambda. Three things happen, in order. First, the held item is released so it goes public on the platform through the platform’s API. Second, the item is cleared from the `cm-queue` so it won’t resurface in a later digest. Third, a `published` row is written to `cm-audit` with Sam, the timestamp, and the rule the checker had cited.

Because the system had held this item and a human published it, this counts as an *overturn* — the system was more cautious than the house actually wants. That correction is saved (more on that below), so the next comment that reads like this one is more likely to pass on its own.

Action 2: remove (the only way content comes down)

The spam comment with the blocked link is the easy case. Sam taps *Remove*. The Function URL Lambda takes the item down through the platform API, sends the poster the polite house reply from the voice doc — “Your comment was removed because it broke our rule on promotional links from unknown sites” — clears the queue entry, and writes a `removed` row to `cm-audit` with the rule and any note Sam added.

This is the one action in the whole system that actually takes content down, and it is always a person tapping the button. The system never reaches this state on its own; the strongest thing it can do unattended is hold an item out of view, which is reversible. Removal is not reversible in the same way, so it stays a human decision — with the reason recorded, so anyone asking “why did this come down?” six months later gets a straight answer.

Action 3: edit-and-publish (the salvage)

Some items are mostly fine with one bad part — a genuinely useful comment that happens to end with a spam link, or a good point wrapped around one slur. Removing the whole thing loses the good part; publishing it as-is breaks a rule. *Edit* opens a Slack modal with the item text pre-filled. Sam trims the link or the slur, hits Save, and the cleaned version is published. Both the original and the edited text are kept — the original in S3, the edit in the item record — so the change is never a mystery and can be undone.

Edit is used sparingly and only on your own surfaces; on platforms that don’t allow editing a member’s words, this button is simply hidden and the moderator chooses publish or remove instead.

Every action is logged, every overturn teaches

The `cm-audit` table records every publish, remove, and edit with the moderator, the timestamp, the rule, and a before-and-after snapshot. That alone makes the system auditable. But the audit trail does double duty: whenever a moderator *overturns* a call — publishes something the system held, or removes something it

had passed — that item, the verdict the system gave, and the human's decision are appended to a small set of worked examples.

Those examples are exactly what the checker in Part 3 loads alongside the house rules and feeds into the model prompt. So the learning loop is simple and contained: the model doesn't retrain, and the rules doc stays the source of truth, but the next borderline item that looks like a past correction gets nudged toward the call your moderators actually made. The set is capped and curated — the most recent and most representative overturns per area — so it sharpens the edges without ballooning the prompt or letting one odd call skew everything.

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 cheap rule pass keeps the model bill small.

PART 6 OF 7

JUNE 17, 2026 PART 6 OF 7 · CONTENT MODERATOR SERIES ~3 MIN READ

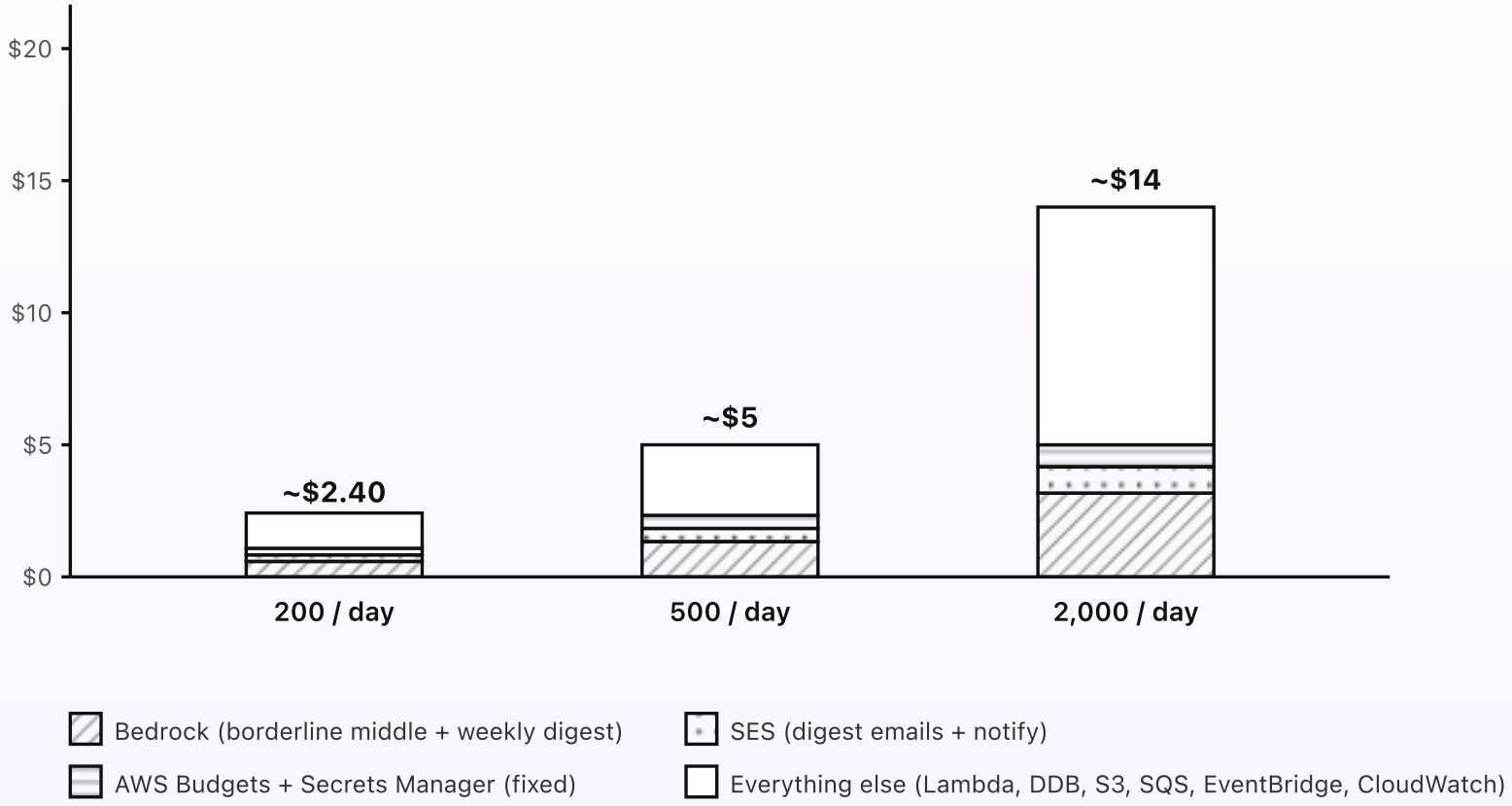
What the content moderator costs

The moderator is one of the cheaper systems in this whole series. Each item runs through a fast rule pass that calls no model, gets one record written, and most of the time stops right there. Only the borderline middle costs a Bedrock call, and Bedrock fires again just once a week for the digest. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (around 200 items a day).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The rule pass settles most items — no model calls on the easy ones.
- Bedrock fires only on the borderline middle and the weekly digest.
- At 500 items a day the bill is around $5. At 2,000 a day it's around $14.

Cost at three volumes



The rule pass keeps the model bill small — most items never reach Bedrock at all.

Fig 6. Monthly cost at three daily-item volumes. Bedrock is a small but visible band because it fires only on the borderline middle and the weekly digest. The dominant cost is the everything-else bucket: the rule pass and storage that touch every item.

Where the dollars actually go

Lambda runtime (the bulk). Every item runs the intake and the rule pass — clean the text, write one record, check the lists. That's a few milliseconds each. The checker Lambda runs only for borderline items, the dispatch Lambda for held and flagged ones, the Function URL Lambda for each moderator action, and the digest Lambda once a week. Add it all up and at 200 items a day the Lambda total is well under a dollar; at 2,000 a day it's a couple of dollars.

DynamoDB on-demand. Four small tables: `cm-items`, `cm-queue`, `cm-audit`, and the worked-examples store. One write per item plus a few reads per borderline item and per action. Pennies a month at SMB volume.

S3 + storage. The raw webhook payloads and the originals of edited items. A few hundred KB to a few MB at SMB volume. Effectively free.

SQS. The review queue plus its dead-letter backstop. The first million requests a month are free; an SMB stays well inside that. Pennies at most.

EventBridge. The verdict events and the weekly-digest schedule. A handful of events per held item. Pennies.

SES. Outbound for the digest emails and any email-fallback cards: $0.10 per thousand sent. A few cents a year at this scale.

Bedrock (only on the borderline middle). The rule pass settles most items with no model. Only the genuinely ambiguous ones reach Haiku 4.5: a few hundred input tokens (the item plus the rules and a few examples) and a short JSON verdict out, so a small fraction of a cent per call. The share of items that need the model is what drives this slice — tune the rule pass and the threshold and it

moves. The weekly digest is one larger call summarizing the week's holds, removals, and overturns; a couple of cents.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the inbound webhook and the action buttons.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Each Lambda runs only when an item or a click arrives.
- **A model on every item.** The rule pass handles the easy majority for free; Bedrock reads only the hard middle.
- **A Knowledge Base.** The house rules are a short doc fed straight into the prompt — no embeddings, no vector store needed.

How the cost scales

Lambda runtime and DynamoDB grow roughly with item count, because every item runs the rule pass and gets a record. Bedrock grows with the *borderline* count — a fraction of the total — so it rises more slowly than volume. SES and the fixed services barely move. So the bill at 5,000 items a day is around $35; at 10,000 a day it's around $70. Past those volumes you'd tighten the rule pass so fewer items reach the model, but that's a tuning knob, not a redesign.

Set an AWS Budgets alarm at $25/month so anything unusual — a spam flood, a runaway loop — pages you before the bill matters. The moderator'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 review queue, and the EventBridge config.

PART 7 OF 7

JUNE 17, 2026 PART 7 OF 7 · CONTENT MODERATOR SERIES ~8 MIN READ

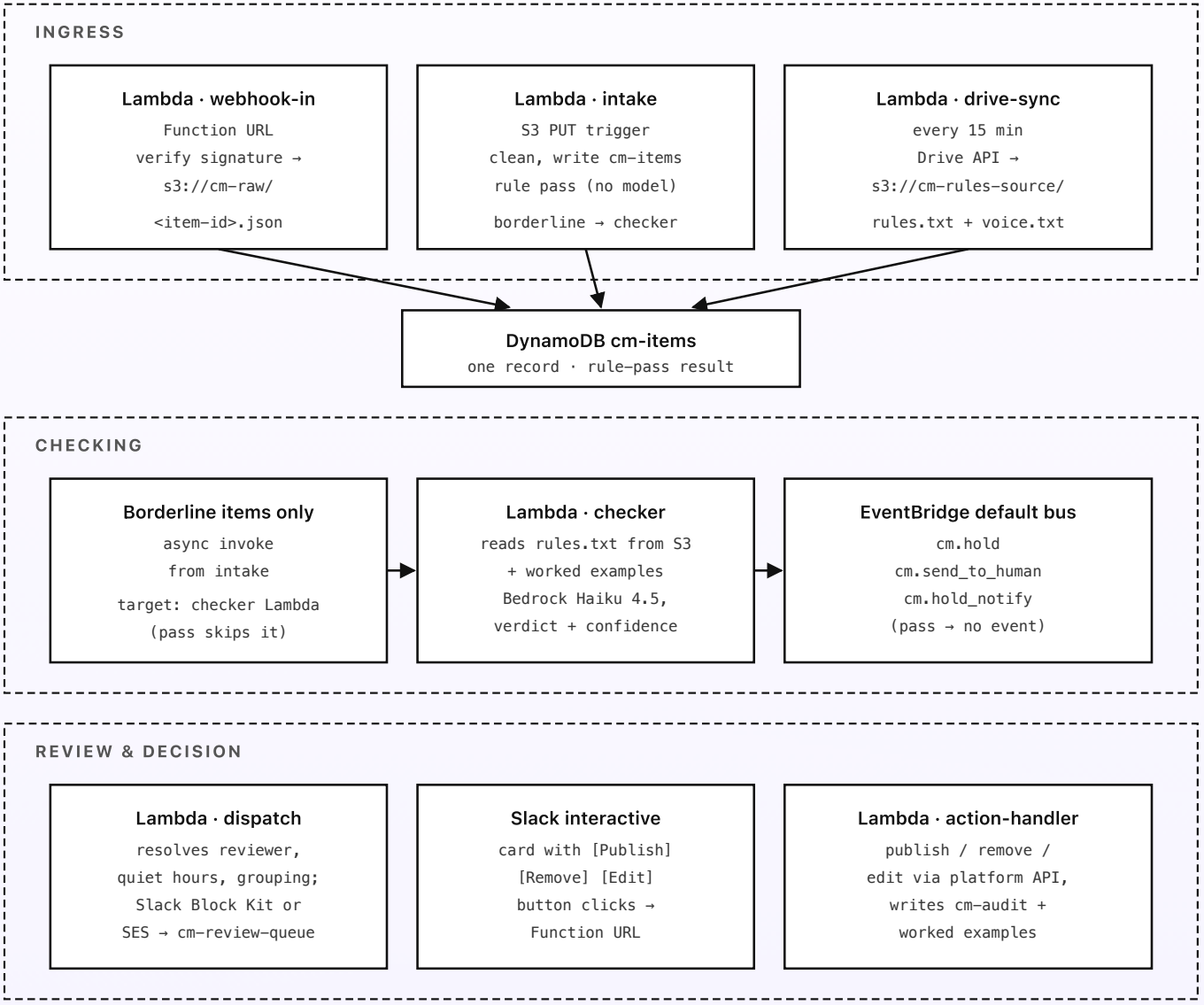
Engineering reference: the content moderator architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SQS review queue, the EventBridge 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). Lambda Function URLs, Bedrock Global cross-Region inference, SQS, and EventBridge are all available there. A second region for resilience isn't worth the extra setup at SMB volume — the failure mode for an SMB is a flagged comment waiting an extra hour for review, not a regional outage. One AWS account dedicated to the moderator (separate from your other workloads) keeps the IAM blast radius small and lets one AWS Budgets alarm cover the whole system.

Topology



Nothing is auto-deleted — and every decision is logged to cm-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (webhook in, clean, rule pass), checking (the borderline middle gets a model verdict and emits events), review and decision (the card reaches a moderator and the human's call is recorded). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

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

- **`webhook-in`** — Lambda Function URL, `AuthType: NONE`; verifies each platform's HMAC signature (secrets per platform in Secrets Manager under `cm/webhook/<platform>`) before doing anything. Writes the raw payload to `s3://cm-raw/<item-id>.json` and returns 200 fast so the platform doesn't retry. All real work is deferred to the S3 PUT trigger. Memory: 256 MB. Timeout: 10 s.
- **`intake`** — S3 PUT trigger on `s3://cm-raw/`. Strips HTML, normalizes text, extracts author/links/length/area, and upserts one record to `cm-items` keyed by `item_id` (idempotent on platform retries). Runs the deterministic rule pass against the allow list, banned-word list, and blocked-domain list loaded from `s3://cm-rules-source/`. On `pass` it marks the item published; on `hold` it emits `cm.hold`; on `borderline` it async-invokes `checker`. Memory: 256 MB. Timeout: 30 s. *No Bedrock calls.*

- **`checker`** — async-invoked by `intake` for borderline items only. Reads `rules.txt` from `s3://cm-rules-source/` and the worked-examples set for the area from `cm-examples`. Calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) with a strict JSON-only contract: `{verdict, confidence, rule}`. Applies the per-area confidence threshold from the rules doc, then emits `cm.hold`, `cm.send_to_human`, or `cm.hold_notify` — or marks the item published on a confident pass. Memory: 512 MB. Timeout: 30 s.
- **`dispatch`** — EventBridge rule on the three move events. Resolves the reviewer (per-area, then admin fallback), checks quiet hours, groups repeat offenders by author, formats the card from `voice.txt`, and sends via the Slack `chat.postMessage` Web API (Block Kit) or SES `SendRawEmail`. Enqueues the card reference to the `cm-review-queue` SQS queue and writes a `cm-queue` row so a re-drive won't double-send. A `cm.hold_notify` bypasses quiet-hours batching. Memory: 256 MB. Timeout: 30 s.
- **`action-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies the Slack signing secret on the request body. Triggered by Slack interactive button clicks (Publish/Remove/Edit) and by email-link clicks. Calls the originating platform's API to publish, remove, or post an edited version; writes the decision to `cm-audit`; on an overturn, appends a worked example to `cm-examples`; clears the `cm-queue` entry. Memory: 256 MB. Timeout: 15 s.
- **`drive-sync`** — EventBridge Scheduler target, every 15 minutes. Uses the Google Drive API (service-account credentials in Secrets Manager under `cm/drive/sa`) to export the rules and voice docs as plain text and write to `s3://cm-rules-source/` only if they changed since the last sync. Memory: 256 MB. Timeout: 30 s.

- `digest` — EventBridge Scheduler target, weekly Monday 9am in `TZ_NAME`. Reads the past week's `cm-audit` and `cm-queue`; calls Bedrock Haiku 4.5 once to write a short narrative summarizing holds, removals, overturns, and any dead-letter items; emails it via SES to the configured stakeholder list and posts a summary to a configured Slack channel. Memory: 512 MB. Timeout: 60 s.

Storage

- **DynamoDB** · `cm-items` — one row per item. PK `item_id`; attributes: `platform`, `area`, `author`, `text`, `links`, `rule_pass`, `state` (published/held/removed/edited), `raw_s3_key`. On-demand.
- **DynamoDB** · `cm-queue` — one row per dispatched card. PK `(item_id, card_id)`; attributes: `reviewer`, `sent_via` (slack/email), `verdict`, `rule`, `group_key`. On-demand. Marks that a card was sent so re-drives don't duplicate.
- **DynamoDB** · `cm-audit` — one row per write action of any kind. PK `(item_id, ts)`; attributes: `action` (publish/remove/edit), `by_user`, `rule`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB** · `cm-examples` — curated worked examples from moderator overturns. PK `area`; sort key `ts`; attributes: `text`, `system_verdict`, `human_decision`, `rule`. Capped per area (most recent N) by a small compaction step in `digest`. On-demand.
- **S3** · `cm-raw` — raw inbound webhook payloads. Versioning enabled. Lifecycle to Glacier at 30 days; expiry at 2 years.

- **S3 · `cm-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled, so a bad Drive edit rolls back in one click.
- **S3 · `cm-originals`** — originals of edited items, kept so any edit-and-publish is reversible and auditable.

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: `checker` for the borderline verdict, and `digest` for the weekly narrative. A heavier reasoning path on Claude Sonnet 4.6 isn't justified here — the verdict is a short, well-scoped classification, and Haiku 4.5 with worked examples handles it cheaply.
- **Embeddings.** Not used. The house rules are a short doc fed straight into the prompt; deterministic lists plus a few worked examples beat vector retrieval at this scale. No Knowledge Base, no S3 Vectors. (If a customer's rules ever grew past a single prompt, Amazon Titan Text Embeddings V2 at 1024 dimensions into Amazon S3 Vectors would be the path — not needed at SMB volume.)
- **Quotas.** Default account quotas are more than enough at SMB volume. The rule pass keeps most items off Bedrock entirely.

SQS review queue

- **`cm-review-queue`** — standard SQS queue holding card references awaiting a moderator. Visibility timeout 5 min; the dispatch path is the producer, the Slack/SES send is the consumer side.

- `cm-review-dlq` — dead-letter queue, `maxReceiveCount: 5`. Anything that fails to send repeatedly lands here instead of looping or vanishing; the weekly `digest` reads and reports DLQ depth.
- **Grouping** — the dispatch computes a `group_key` of `(author, rule)` and folds new items for an existing open key into one card, so a burst of identical spam is one review, not fifty.

EventBridge config

- `cm-move-rule` — rule on the default bus matching `cm.hold`, `cm.send_to_human`, `cm.hold_notify`. Target: `dispatch` Lambda.
- `cm-drive-sync` — Scheduler `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `cm-weekly-digest` — Scheduler `cron(0 9 ? * 2#1 *)` (Monday 9am, weekly cadence) in `TZ_NAME`. Target: `digest` Lambda.
- **Notify path** — a `cm.hold_notify` event is matched by the same `cm-move-rule`; `dispatch` reads the event detail and skips quiet-hours batching for it.

Platform webhooks and APIs

- Each platform (community page, comment plugin, review source) is configured to POST new-content webhooks to the `webhook-in` Function URL with a shared signing secret.
- Per-platform API credentials for the publish/remove/edit calls live in Secrets Manager under `cm/platform/<platform>`. The `action-handler` dispatches to the right client by the item's `platform` attribute.

- Platforms that don't support editing a member's content have the Edit button suppressed at card-compose time; only Publish and Remove are shown.

IAM (least privilege per Lambda)

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

- **webhook-in role:** `s3:PutObject` on `cm-raw`; `secretsmanager:GetSecretValue` on the per-platform webhook secrets. Nothing else.
- **intake role:** `s3:GetObject` on `cm-raw` and `cm-rules-source`; `dynamodb:PutItem` on `cm-items`; `events:PutEvents` on the default bus; `lambda:InvokeFunction` on `checker`. *No* `bedrock:*`.
- **checker role:** `s3:GetObject` on `cm-rules-source`; `dynamodb:Query` on `cm-examples`; `bedrock:InvokeModel` on the Haiku ARN; `events:PutEvents` on the default bus.
- **dispatch role:** `sqs:SendMessage` on `cm-review-queue`; `secretsmanager:GetSecretValue` on the Slack bot token; `ses:SendRawEmail` from the verified sender identity; `dynamodb:PutItem` + `Query` on `cm-queue`; outbound network to `slack.com`.
- **action-handler role:** `dynamodb:PutItem` on `cm-audit` and `cm-examples`; `dynamodb:UpdateItem` on `cm-items`; `secretsmanager:GetSecretValue` on the Slack signing secret and the per-platform API secrets; `s3:PutObject` on `cm-originals`; outbound network to the platform API hosts.
- **drive-sync and digest roles:** drive-sync gets `secretsmanager:GetSecretValue` on the Google service-account secret and

`s3:PutObject` on `cm-rules-source`; digest gets `dynamodb:Query` on `cm-audit` / `cm-queue`, `sqs:GetQueueAttributes` on the DLQ, `bedrock:InvokeModel` on the Haiku ARN, and `ses:SendRawEmail`.

Slack interactive flow

Cards are posted via the `chat.postMessage` Web API with Block Kit blocks containing the action buttons (Publish/Remove/Edit). 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, parses the `action_id` (`publish`, `remove`, `edit`), opens a modal for Edit, and processes the decision on submit. Publish and Remove are one-tap.

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

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:** `webhook-in` 5xx rate > 1% in 5 min (dropped inbound content is the worst failure); `cm-review-dlq` depth > 0; `action-handler` signature-verification failures > 5/hour (might mean the Slack secret rotated).
- **X-Ray:** off by default. Not worth the cost at SMB volume.

- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `cm-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Google service-account credentials for the Drive API live in Secrets Manager under `cm/drive/sa`. Slack bot token and signing secret under `cm/slack/*`. Per-platform webhook signing secrets under `cm/webhook/*` and per-platform API credentials under `cm/platform/*`. The configured timezone, quiet-hours window, per-area confidence thresholds, and admin fallback reviewer live in Parameter Store under `/cm/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 AWS keys — running AWS SAM to ship the stack. The opinionated bits: turn on S3 versioning for `cm-raw`, `cm-rules-source`, and `cm-originals`; keep the dead-letter queue and its alarm in the same stack as the review queue so they ship together; and pin the EventBridge Scheduler timezone so the weekly digest doesn't silently start running in UTC after a CI rotation. Total deployable surface: seven Lambdas, four DDB tables, three S3 buckets, one SQS queue plus its DLQ, one EventBridge rule on the default bus (plus 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.