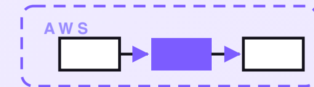
7-PART SERIES · FREE COMPANION



Social inbox unifier

A serverless system that pulls every Instagram, Facebook, and WhatsApp DM into one shared queue so nothing is missed. It labels each message by topic and urgency, removes duplicates, and routes each to the right teammate — so a small team answers from one place instead of five apps. A human writes every reply; the system only gathers, labels, and routes. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

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

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/social-inbox-unifier

CONTENTS

Social inbox unifier

01 A social inbox unifier on AWS for a few dollars a month

02 How a DM reaches the unified inbox

03 How a DM gets labeled and deduped

04 How a DM finds the right teammate

05 How a DM gets answered

06 What the social inbox unifier costs

07 Engineering reference: the social inbox unifier architecture

PART 1 OF 7

JUNE 12, 2026 PART 1 OF 7 · SOCIAL INBOX UNIFIER SERIES ~5 MIN READ

A social inbox unifier on AWS for a few dollars a month

A small business gets messaged in more places than anyone can keep open at once. The Instagram DM asking if you ship overseas. The Facebook message about a refund. The WhatsApp from a regular who wants to book again. The comment that turned into a private message. Five apps, five sets of notifications, and a real chance that the message that mattered most got buried under the one that didn't. This post walks through the design of a small system that pulls all of it into one shared queue, labels each message so the urgent ones rise to the top, drops duplicates, and hands each one to the right teammate — while a human still writes every reply.

KEY TAKEAWAYS

- Every platform's DMs flow into one shared queue through small per-platform connectors.
- One cheap model call labels each message by topic, urgency, and language — nothing more.
- Duplicates are dropped by fingerprint, so a re-sent webhook never opens a second thread.
- Each message is routed to the right teammate by topic, working hours, and fair load-sharing.
- A human writes every reply. The system gathers, labels, and routes — it never answers.
- Designed on AWS for about $3/month at typical small-business volume.

The whole system on one page

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

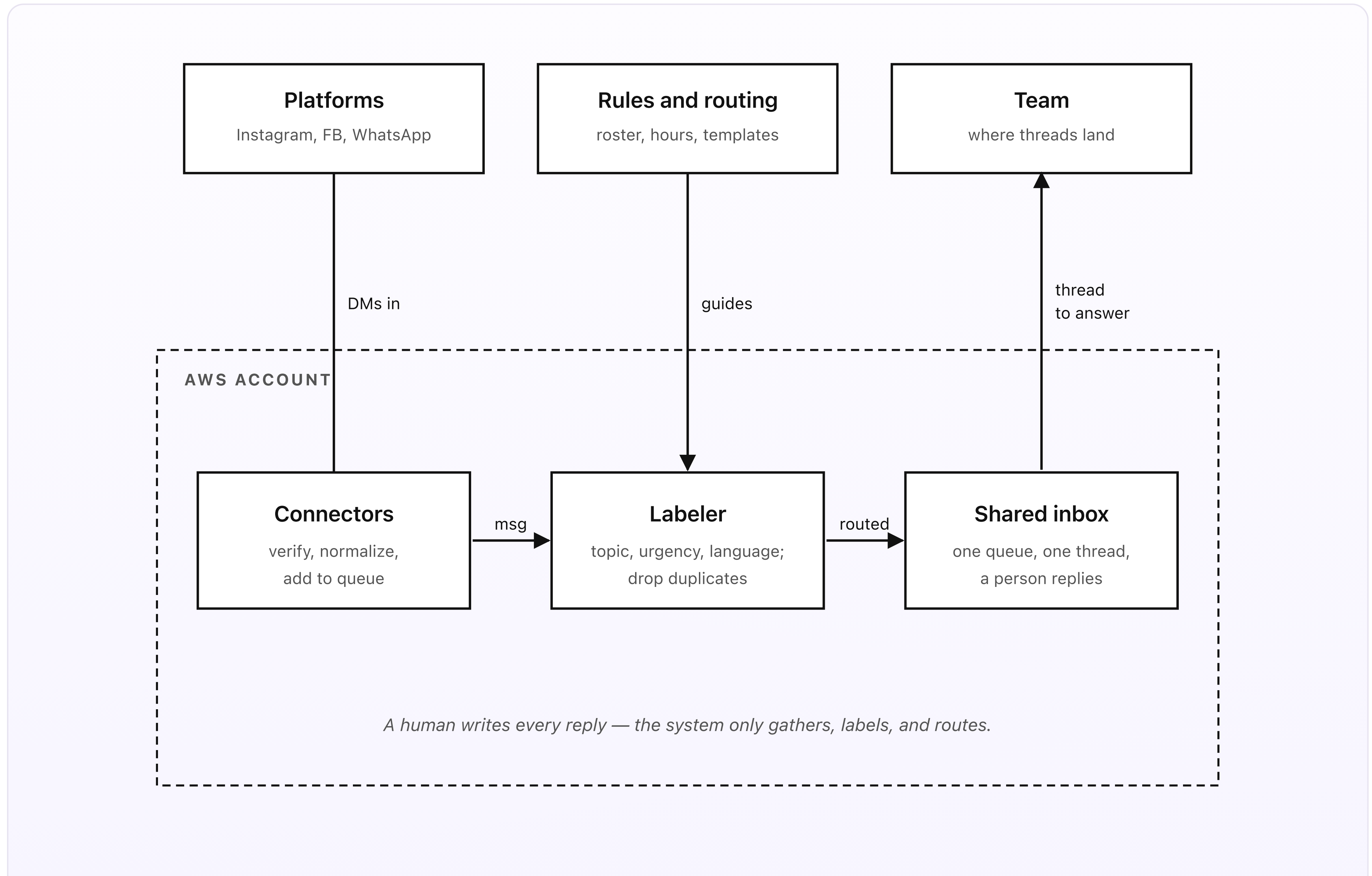

Platforms
Instagram, FB, WhatsApp
Rules and routing
roster, hours, templates
Team
where threads land
DMs in
guides
thread
to answer
AWS ACCOUNT
Connectors
verify, normalize,
add to queue
msg
Labeler
topic, urgency, language;
drop duplicates
routed
Shared inbox
one queue, one thread,
a person replies
A human writes every reply — the system only gathers, labels, and routes.

Fig 1. Three sources outside, three pieces inside AWS. Messages flow in from each platform's webhook. The Labeler reads each one once and drops duplicates. The Shared inbox puts the right thread in front of the right person, who writes the reply.

What you set up once (the outside)

- **Platforms.** Connect each channel you use — an Instagram account, a Facebook Page, a WhatsApp business number — so each one sends new messages to your system. Most of these platforms do this the same way: you give them a web address (a "webhook" — just a URL they call whenever something new happens), and from then on every incoming DM gets pushed to that address within seconds. You connect each platform once and forget it. Adding a new channel later is one more connector, covered in Part 2.
- **Rules and routing.** A small set of config the team can edit without a deploy. It holds the routing rules ("refund questions go to billing; shipping questions go to ops"), the team roster with each person's topics and working hours, the urgency targets (how fast each level should be answered), and the message templates a teammate can pick from to start a reply. None of these write the reply for you — they just save a person from typing the same opening line for the fiftieth time.
- **Team.** The people who answer. Each teammate has a seat in the shared inbox and a short profile: which topics they handle, their working hours, and a backup teammate for when they're off. Threads land in their queue already labeled and sorted, so they open the most urgent one first instead of scrolling five apps to find it.

What runs on every message (the inside)

- **The connectors.** One small piece of code per platform. When a platform calls the webhook, the connector first checks the call is genuine (each platform signs its calls; the connector verifies the signature so a stranger can't inject fake messages). Then it turns that platform's particular format into one common shape — sender, platform, text, timestamp, conversation id — and drops it on a single work queue. After that point, the rest of the system doesn't care which app the message came from.
- **The labeler.** Pulls each message off the queue. First it builds a fingerprint and checks whether this exact message already arrived; if so, it drops the duplicate and links it to the existing thread. If it's new, one Bedrock Haiku 4.5 call reads the text and returns a small label set: topic, urgency (urgent, normal, or low), language, and a one-line summary. That's the only model call in the whole system, and it never writes a reply — it only describes the message so the routing and sorting can be smart.
- **The shared inbox.** Takes the labeled message and decides whose queue it belongs in, then shows it as one clean thread. A teammate opens the thread, reads the full back-and-forth, and types the reply. On send, the reply goes back out through the same platform the customer used. The thread can be reassigned or closed. A daily digest summarizes what came in, what's still open, and anything that sat too long.

In plain words

A customer DMs your Instagram at 9:14am: "Hi, my order #4821 still hasn't shipped and I leave for a trip Friday — can you check?" The connector verifies the call and drops it on the queue. The labeler reads it: topic *shipping*, urgency *urgent*

(there's a deadline), language *English*. The routing sends it to Sam in ops, who's on shift, with the thread already open and the order number highlighted. Sam checks the system, sees the order shipped overnight, and types a reply with the tracking link. It goes back out as an Instagram DM under your brand. The same customer also messaged your Facebook Page with the same question ten minutes later — the fingerprint check spots the duplicate and links it to Sam's thread instead of opening a second one, so nobody answers the same person twice.

The cost of running this is about $3 a month at SMB volume. The cost of *not* running it is the DM that sat unread for two days because it landed in the one app nobody had open — and the customer who quietly went somewhere else.

DESIGN RULES THAT SHAPED EVERY DECISION

- A human writes every reply. The system gathers, labels, and routes — it never auto-answers.
- One common message shape. Past the connector, no piece of the system cares which app it came from.
- One cheap model call per message, for labels only. Routing and dedupe are plain code.
- Duplicates are dropped by fingerprint, so one conversation is ever only one thread.
- The rules live in config. Changing routing, hours, or a template doesn't need a deploy.
- Every action is logged. You can see who answered what, and when, months later.

Why this shape

Most small teams handle social messages one of three ways: each person watches one or two apps, somebody checks all the apps a few times a day, or a shared phone gets passed around. The first way means messages get missed whenever the watcher is busy. The second is slow by design — an urgent question waits hours for the next sweep. The third is a coordination mess: two people answer the same DM, or each assumes the other did, and nobody does.

The setup above leaves the customer-facing channels exactly where they are — people keep messaging your Instagram and your Facebook like always — but adds a small system that *watches every channel at once* and puts the one queue in front of the team. Each message arrives labeled, so the urgent one rises to the top. Each is routed, so the right person sees it without anyone triaging by hand. Duplicates collapse into one thread. And critically, the system stops there: it never writes or sends a reply. A person reads, decides, and answers in their own words. The machine just makes sure the message reached them.

The next four posts walk through each piece in turn: how a DM reaches the unified inbox, how it gets labeled and deduped, how it finds the right teammate, and how it gets answered by a human. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 12, 2026 PART 2 OF 7 · SOCIAL INBOX UNIFIER SERIES ~4 MIN READ

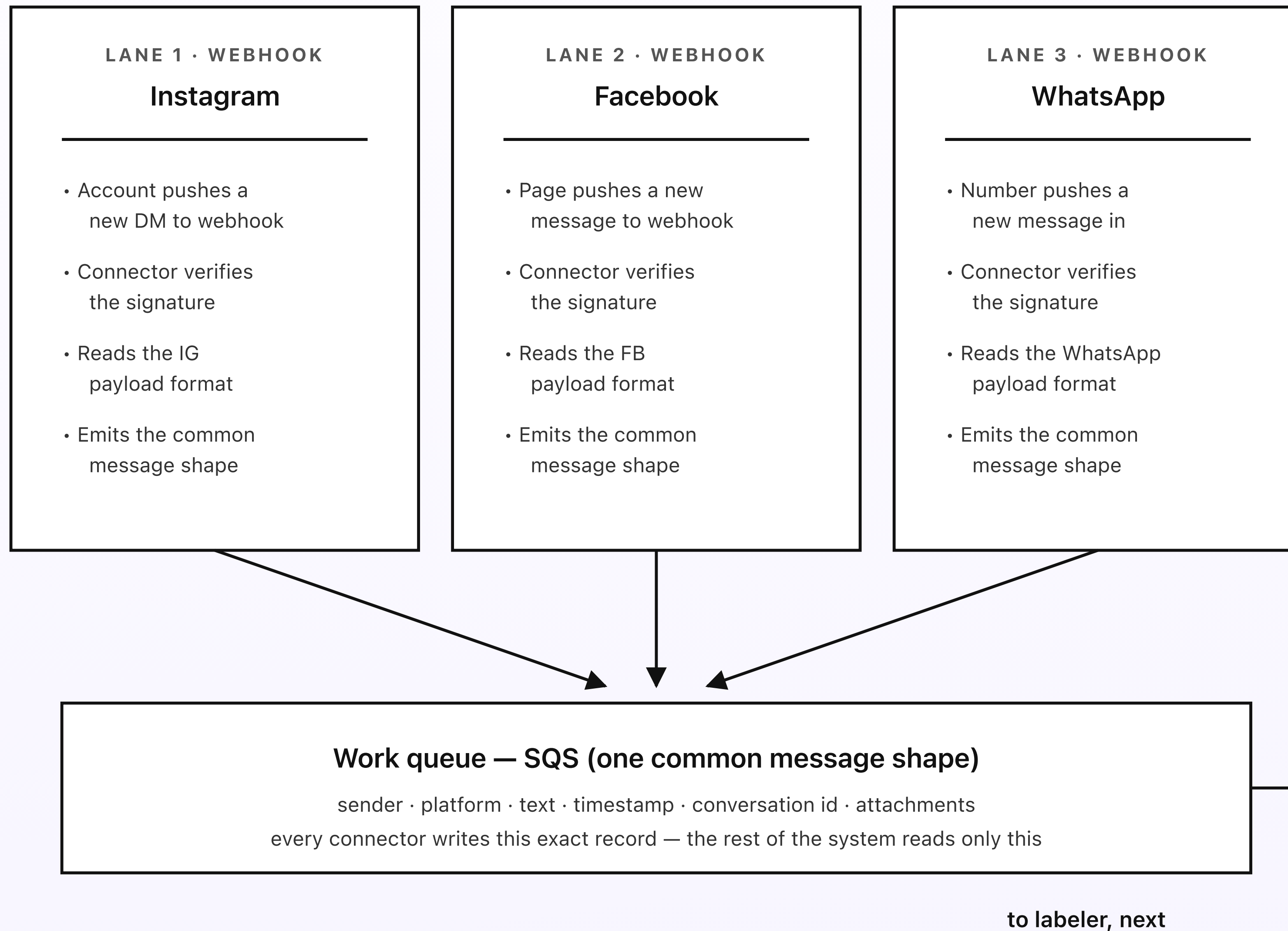
How a DM reaches the unified inbox

The inbox can only unify what reaches it. So the first job is getting every platform's messages into one place, in one shape, the moment they arrive. Each platform does the pushing itself: you give it a web address, and from then on it calls that address every time a new DM comes in. A small connector per platform catches the call, checks it's genuine, and turns whatever odd format that platform uses into one common message everyone downstream understands. Three platforms in, one queue out.

KEY TAKEAWAYS

- Each platform pushes new messages to a webhook — a URL it calls within seconds of a DM arriving.
- One small connector per platform verifies the call's signature so fake messages can't get in.
- Every connector turns its platform's format into one common shape: sender, platform, text, time, thread.
- Each normalized message lands on a single work queue (SQS) for the labeler to pick up.
- Adding a new channel later is just one more connector that writes the same common shape.

Three connectors into one queue



Every platform speaks its own format at the door — past the connector, the system sees one shape.

Fig 2. Three connectors converge on one work queue. Each platform pushes its own format to its own webhook; each connector verifies and normalizes, then writes the same common record. Past the queue, no piece of the system needs to know which app a message came from.

What a webhook is, in one paragraph

A webhook is just a phone number you give a platform so it can call you when something happens, instead of you calling it over and over to ask. You register a web address with Instagram, Facebook, and WhatsApp once. From then on, the moment a customer sends a DM, that platform makes a small web request to your address with the message attached. There's no polling, no app left open, no delay — the message shows up at your door within a second or two of being sent. In this system, each platform's web address is a **Lambda Function URL** — a direct, no-frills address for a small piece of code, with no API Gateway in front of it.

The connector's first job: make sure the call is genuine

An address that anything on the internet can call is an address anything on the internet can lie to. So every platform signs its calls — it attaches a short code, computed from a shared secret only you and the platform know, that proves the call really came from them. The connector's first move is to recompute that code and compare. If it doesn't match, the call is dropped on the spot and nothing enters the queue. The shared secret for each platform lives in Secrets Manager, never in the code. This one check is what stops a stranger from injecting fake DMs into your team's queue.

The connector's second job: turn five formats into one

Every platform describes a message a little differently. Instagram nests the text one way, Facebook another, WhatsApp a third; one calls the customer a “sender id,” another a “from,” another a phone number. If the rest of the system had to know all of those quirks, every downstream piece would carry three sets of special cases forever. Instead, each connector flattens its platform’s payload into one common record: `sender`, `platform`, `text`, `timestamp`, `conversation_id`, and any attachments. From the queue onward, the labeler, the router, and the inbox all read that one shape. Adding a fourth channel — a website chat, say — means writing one more connector that emits the same record, and nothing downstream changes.

Why a queue sits in the middle

The connector doesn’t label or route the message itself. It just drops the normalized record on a work queue — an SQS queue — and replies to the platform right away so the platform knows the message was received. This matters for two reasons. First, the platform wants a fast answer; if the connector tried to do the labeling and routing before replying, a slow moment could make the platform think the call failed and re-send it. Second, the queue smooths out bursts: if forty messages arrive in one minute during a sale, they line up in the queue and get worked through steadily instead of overwhelming anything. If a message ever fails to process, it lands in a dead-letter queue — a side queue for messages that need a second look — so nothing is silently lost.

Next post: how the labeler picks each message off the queue, reads it once with a model to get its topic and urgency, and drops the duplicates before they ever open a second thread.

PART 3 OF 7

JUNE 12, 2026 PART 3 OF 7 · SOCIAL INBOX UNIFIER SERIES ~5 MIN READ

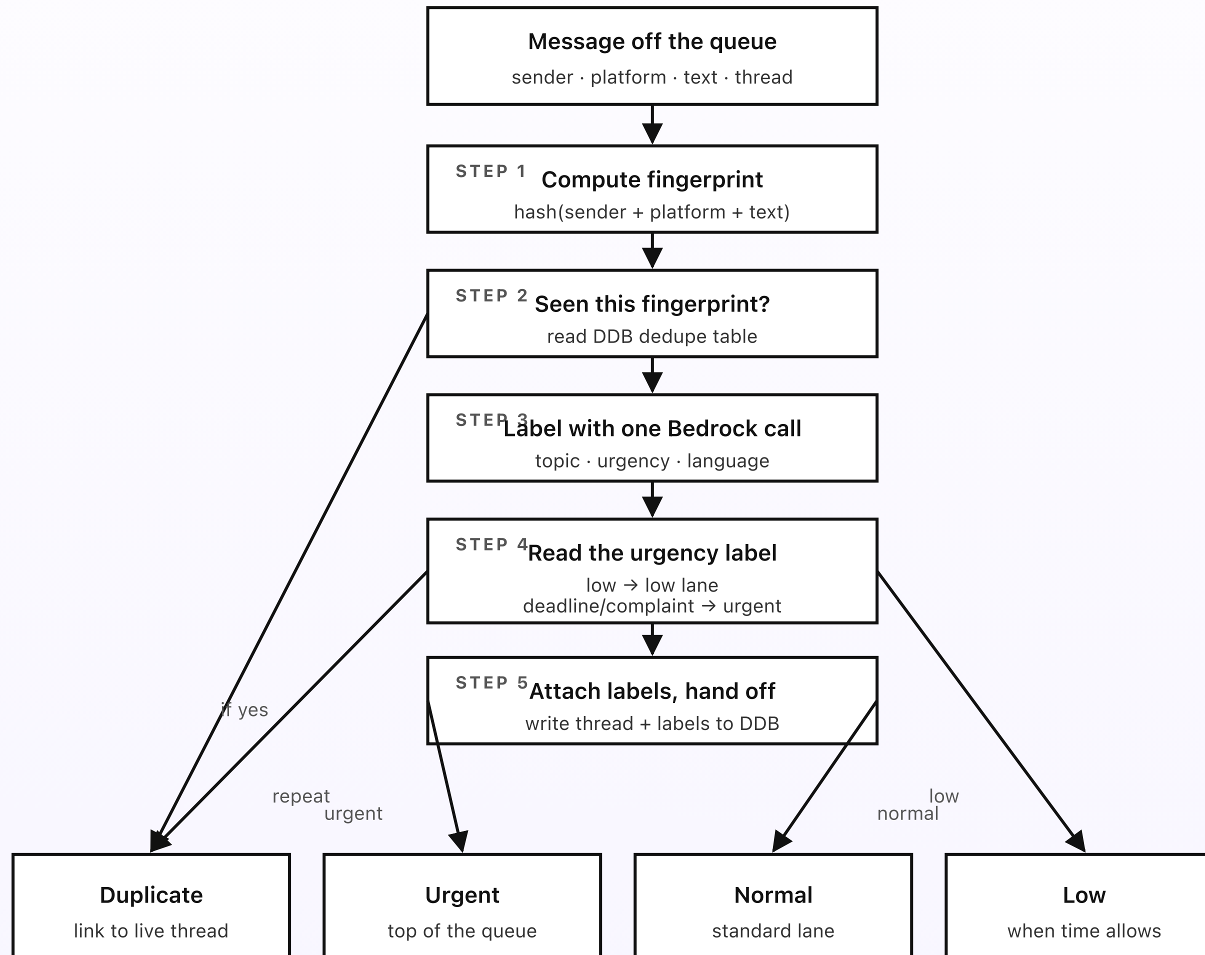
How a DM gets labeled and deduped

A message is sitting on the work queue in the common shape. Two things have to happen before it's ready for a teammate: drop it if it's a copy of one already here, and tag it with a topic, an urgency, and a language so the inbox can sort and route it. The dedupe is plain code — a fingerprint and a quick lookup. The tagging is one short model call that reads the message and hands back a few labels. The model reads the message; it never answers it.

KEY TAKEAWAYS

- Every message gets a fingerprint from its sender, platform, and a hash of the text.
- A short-lived DynamoDB table remembers recent fingerprints; a repeat is dropped and linked, not re-opened.
- One Bedrock Haiku 4.5 call returns topic, urgency, language, and a one-line summary — nothing else.
- The model is asked for labels only. It never drafts or sends a reply.
- Labeled, deduped messages move on to routing; duplicates just attach to the existing thread.

The decision flow, per message



The model returns labels only — it never writes or sends a reply. A person does that, later.

Fig 3. The labeler's flow, per message off the queue. Fingerprint first, so a duplicate never opens a second thread; then one model call for labels; then sort by urgency and hand off to routing. The model describes the message — it never answers it.

Dedupe first, because a copy is worse than nothing

Duplicates are common, and they come from two everyday situations. The first is the platform re-sending the same webhook — if a connector is slow to reply for a moment, the platform assumes the call failed and tries again, so the exact same message arrives twice. The second is a customer who, getting no instant answer, messages you again on a different channel: the same person, the same question, on Instagram and on Facebook five minutes apart. Both should land as *one* thread, not two, because two threads means two teammates might both reply, or each assumes the other will.

So before anything else, the labeler builds a **fingerprint**: a short code computed from the sender, the platform, and a hash of the message text. (A hash is just a fixed-length stamp of the text — same text in, same stamp out.) It checks that fingerprint against a small DynamoDB table that remembers recent fingerprints for a short window — long enough to catch re-sends and same-day cross-channel repeats, short enough that a genuinely new question next month is never mistaken for an old one. If the fingerprint is already there, the message is a duplicate: it's attached to the existing thread as "also messaged on Facebook" and goes no further. If it's new, the fingerprint is recorded and the message continues.

One model call, for labels only

A new message then gets exactly one Bedrock Haiku 4.5 call. The prompt is short and strict: “Read this message. Return JSON only with four fields — `topic` (one of the configured topics), `urgency` (urgent, normal, or low), `language`, and a one-line `summary`. Do not write a reply. Do not suggest a reply.” A few hundred input tokens, a few dozen out. That’s the entire AI footprint of the system on the hot path.

The labels do real work downstream. `topic` drives routing — a refund question goes to billing, a shipping question goes to ops. `urgency` drives the sort order, so the message with a deadline rises above the one that just says “thanks.” `language` lets the inbox flag a Spanish message for a teammate who handles Spanish, or show a translation hint. The one-line `summary` is what a teammate sees in the queue list before opening the thread, so they can pick the right one to work first.

Why the model only labels — and never answers

This is the line the system never crosses. The model is allowed to *describe* a message; it is never allowed to *respond* to one. There are two reasons, and they reinforce each other. The first is trust: a customer-facing reply in your brand’s voice is something a person should own, word for word. A wrong auto-reply — a refund promised that policy doesn’t allow, a tone that misreads an upset customer — is far more damaging than a message that waited ten extra minutes for a human. The second is cost and simplicity: labeling is a tiny, predictable call, while drafting and checking replies would be a much larger, riskier loop. Keeping the model on labels makes the whole system cheap, fast, and easy to reason about.

What if a label is wrong

Labels are a starting point, not a verdict. The shared inbox shows the topic and urgency as editable tags. If the model called something “normal” that a teammate sees is clearly urgent, they bump it with one click, and the thread re-sorts and can re-route. Those corrections are logged. Over time they’re a useful signal about which topics the labeling tends to misjudge — the kind of thing you tune in the prompt or the topic list, not by adding more model calls. The point is that a human is always in a position to overrule the machine, on every message.

Next post: how a labeled, deduped message finds the right teammate — routing by topic, working hours, fair load-sharing, and a backup for when the right person is off.

PART 4 OF 7

JUNE 12, 2026 PART 4 OF 7 · SOCIAL INBOX UNIFIER SERIES ~5 MIN READ

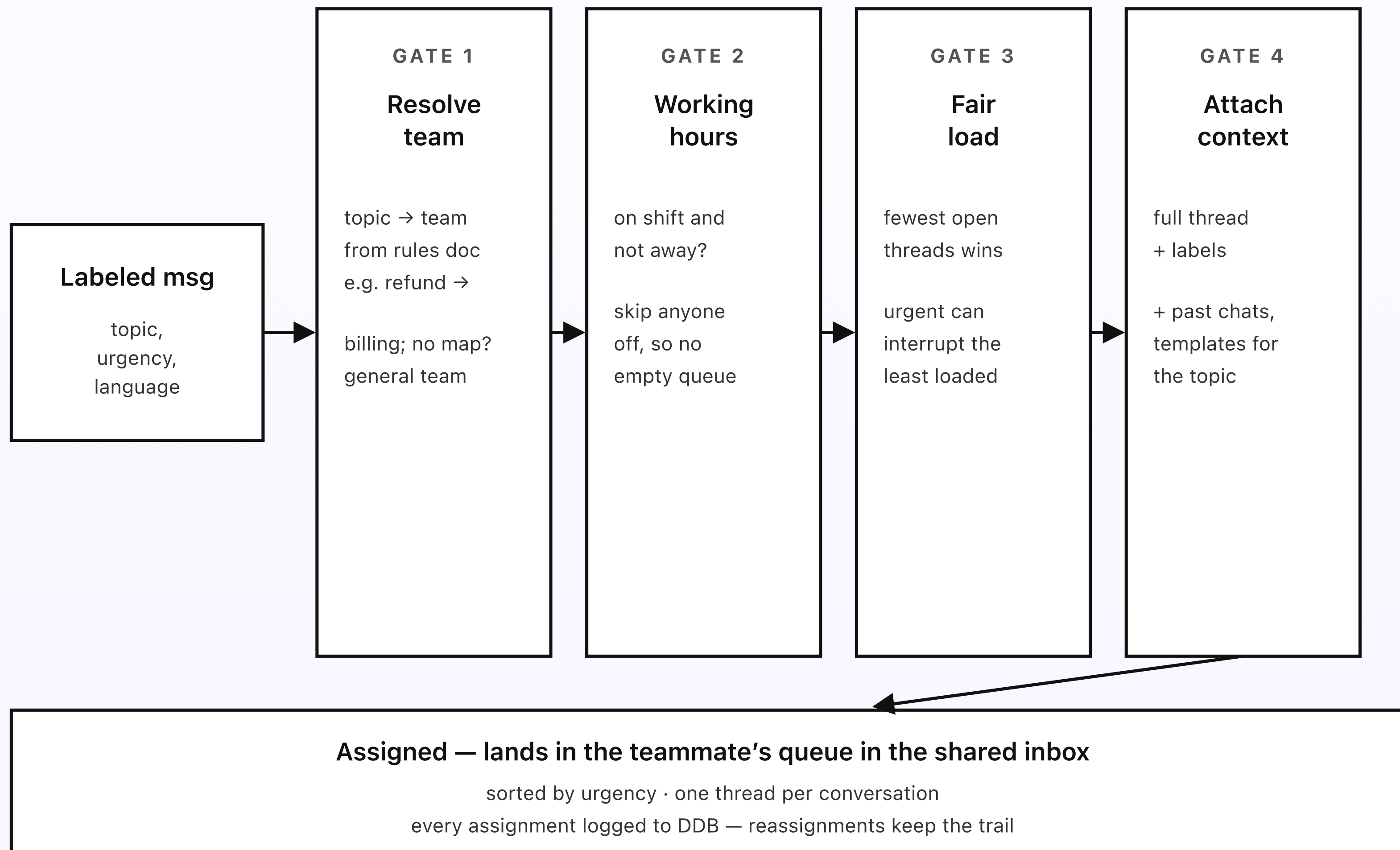
How a DM finds the right teammate

The message is labeled with a topic and an urgency, and it's not a duplicate. Now the system has to put it in front of one person — the right person, who's actually working, who isn't already buried, with enough context to answer without digging. Get any of those wrong and the message either sits in the queue of someone who's off, or piles onto the one teammate who never says no. Four small guardrails sit between the labels and the message landing in a specific queue.

KEY TAKEAWAYS

- Topic resolution: the message's topic maps to the team that owns it in the rules doc.
- Working hours: a teammate who's off-shift is skipped, so messages don't sit in an empty queue.
- Fair load-sharing: among eligible teammates, the one with the fewest open threads gets it.
- Every assignment carries the full thread, the labels, and the customer history — no digging.
- If nobody's eligible or a thread sits too long, it routes to a backup and posts in a team channel.

Four guardrails on every assignment



No one eligible, or a thread sits too long? It routes to a backup and posts in a team channel.

Fig 4. Four guardrails between the labels and the right queue. Resolve the team. Skip anyone off-shift. Spread the load fairly. Attach the full context. Then assign and sort by urgency — and back everything up so nothing sits unseen.

Gate 1: resolve the team from the topic

Routing starts with the topic the labeler gave the message. The rules doc has a short table that maps each topic to the team that owns it: refund and billing questions to the billing folks, shipping and order questions to ops, partnership and press to the owner, everything-else to a general team. That table is config, so the team can change who owns "wholesale" without anyone touching code. If a message comes in with a topic that isn't mapped — rare, but it happens with a brand-new kind of question — it falls back to the general team rather than getting stuck. The general team's job partly is to spot those and add a mapping for next time.

Gate 2: skip anyone who's off

Assigning a message to someone who clocked out three hours ago is the same as not assigning it at all — worse, because it *looks* handled. So Gate 2 narrows the team down to the people actually working. Each teammate has working hours in their profile and a status they can flip to "away" for a long lunch or a sick day. Anyone off-shift or away is skipped. If a whole team is off — say a refund comes in at 2am and billing only works days — the message waits in the team's shared lane and is the first thing they see in the morning, and if its urgency target would be blown by waiting, the backup path in the note takes over.

Gate 3: spread the load fairly

Among the teammates who are on-shift and own the topic, the system picks the one with the *fewest open threads* right now. This is the small piece of fairness that keeps a shared inbox from becoming one person's burden. Without it, messages tend to pile on whoever is fastest or most senior, who then burns out while others sit idle. The count of open threads per teammate is kept in DynamoDB and updated as threads open and close, so the choice is always based on the current moment, not a fixed rotation. An urgent message is allowed to jump in: it goes to the least-loaded eligible person immediately and sorts to the top of their queue, rather than waiting its turn behind normal-priority threads.

Gate 4: attach the context, then assign

The last gate makes sure the teammate doesn't have to go hunting for anything. It bundles the full thread (every message in the conversation so far, across whatever platforms it touched), the labels, the customer's recent history with you ("messaged twice last month about the same order"), and a few reply templates for the topic that the teammate can start from if they want. All of that travels with the message into the assigned queue, so opening the thread shows everything in one view. The teammate reads, decides, and writes — which is Part 5.

Every assignment writes a row to DynamoDB: which thread, which teammate, when, and why (the topic and urgency that drove it). Reassignments later write their own rows, so the trail of who held a thread and when stays complete.

Why the guardrails exist

None of these gates are clever. They’re the small care a good shift lead would take if they were handing out messages by hand — give it to the team that knows the topic, don’t give it to someone who left, don’t dump everything on one person, and hand it over with everything they need to answer. Writing them as four plain checks in code means the team doesn’t depend on a shift lead being awake and paying attention at 9pm on a Saturday. And because none of it involves a model, the routing is fast and predictable: the same message, with the same team on shift, always lands the same way.

Next post: what happens when a teammate opens the thread — how a human writes the reply, how it goes back out through the right platform, and how a thread gets reassigned or closed.

PART 5 OF 7

JUNE 12, 2026 PART 5 OF 7 · SOCIAL INBOX UNIFIER SERIES ~5 MIN READ

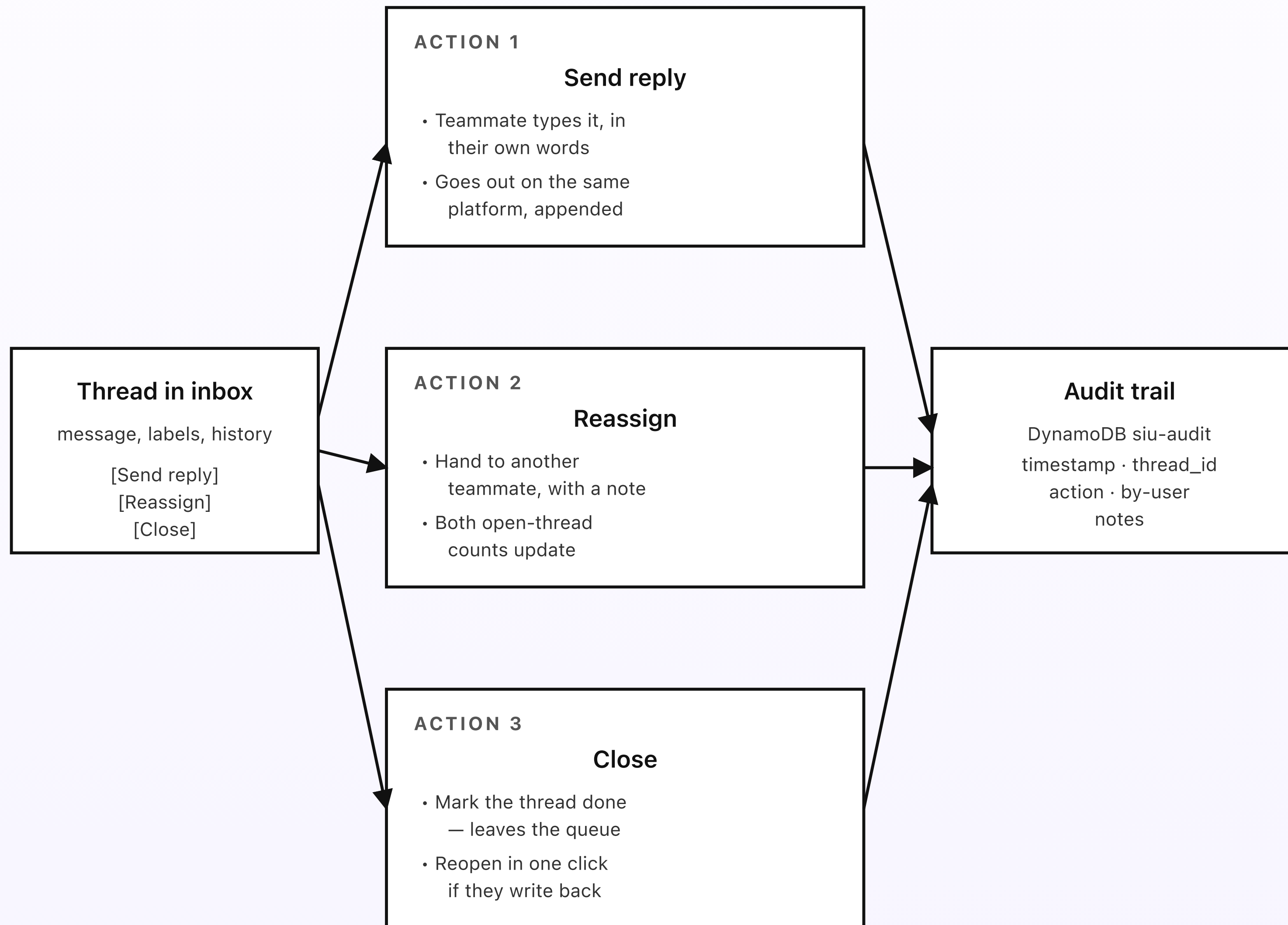
How a DM gets answered

The thread is in Sam's queue at the top, labeled *shipping · urgent*, with the order number highlighted and the full back-and-forth on screen. What happens next? A person reads it and decides. This post walks through the three things a teammate can do on a thread — send a reply, reassign it, or close it — and how the customer's platform, the queue, and the audit trail all stay in sync. The one thing the system never does is write or send the reply itself.

KEY TAKEAWAYS

- Three actions per thread: *send reply* (human-written), *reassign* (hand to another teammate), *close* (mark done).
- Every reply is typed by a person and goes back out through the same platform the customer used.
- Templates can pre-fill an opening line, but the person edits and owns every word before sending.
- Reassign moves the thread to a new queue and updates both teammates' open-thread counts.
- Every action writes an audit row — who did what, when — so a thread's history is never a mystery.

Three actions on a thread



Every reply is human-written and goes out on the customer's own platform — a person answers.

Fig 5. Three actions per thread, three different effects. Send a human-written reply back out on the customer's platform. Reassign to another teammate. Or close. Every action writes to the audit trail.

Action 1: send a reply (the most common)

Sam reads the thread, checks the order, and sees it shipped overnight. He types: "Good news — your order #4821 went out last night, here's your tracking link. It should arrive Thursday, ahead of your trip. Safe travels!" He could have started from a shipping template that pre-fills the opening and the tracking-link placeholder, but he edited it to fit this customer and this moment. He hits Send.

The Send button submits to a Function URL Lambda. Two things happen, in order. First, the Lambda posts the reply back out through the *same platform* the customer used — an Instagram DM goes back as an Instagram reply, a WhatsApp message goes back as a WhatsApp reply — using that platform's send API and the credentials in Secrets Manager. Second, the reply is appended to the thread so the next teammate to open it sees the full conversation, and an `action: replied` row is written to the audit table. The customer experiences a normal, on-brand reply on the app they already use. They never know there's a unified inbox behind it.

The system writes none of those words. Templates can offer a starting point, but a person edits and owns every reply before it goes out. There is no "auto-reply" button, by design.

Action 2: reassign (the hand-off)

Sometimes the message landed with the right team but the wrong person, or it turns out to need someone else. The shipping question is actually a refund in disguise. The customer is a VIP the owner wants to handle personally. Sam is going off-shift and wants to hand a live thread to whoever's taking over.

Reassign lets him pass the thread to another teammate or team, with an optional note on why ("customer is upset, please handle gently"). On save, the thread moves to the new owner's queue at the right urgency, Sam's open-thread count drops by one and the new owner's rises by one (so the fair-load math from Part 4 stays accurate), and an `action: reassigned` row is written with both names. The customer sees nothing — no "you're being transferred" message goes out unless a teammate chooses to send one. The hand-off is internal.

Action 3: close (the "done")

When the conversation is finished — the question answered, the issue resolved — the teammate closes the thread. It leaves the active queue so nobody works it again, the owner's open-thread count drops, and an `action: closed` row is written. The thread isn't deleted: it stays in the customer's history, so the next time that person messages you, whoever picks it up sees the whole past relationship.

Closing isn't final in a scary way. If the customer writes back after a thread is closed — "actually, one more thing" — the new message's fingerprint links to the same conversation, and the thread reopens in the queue automatically, with all its history intact. Nobody starts from scratch.

Every action is logged, every thread has a history

The audit table records every reply, reassign, and close with the teammate who did it, the timestamp, and a note. That trail is what makes a shared inbox trustworthy for a team. When a customer says “someone promised me a refund last week,” anyone can open the thread and see exactly who replied, what they said, and when. When a thread bounced between three people, the reassign notes explain why. The audit trail is the team’s shared memory — especially useful months later, or after the person who handled it has moved on.

And the rule holds at every step: the machine gathered the message, labeled it, routed it, and kept the record — but a human read it and a human answered it, in their own words, on the customer’s own platform.

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 only model call — the per-message label — stays cheap.

PART 6 OF 7

JUNE 12, 2026 PART 6 OF 7 · SOCIAL INBOX UNIFIER SERIES ~3 MIN READ

What the social inbox unifier costs

This is one of the cheaper systems in the series. Most of it is plain code: a webhook catches a message, a connector normalizes it, a queue holds it, a few rows go to DynamoDB, and a thread shows up in the inbox. The only model call is one short label per new message. So the bill tracks how many messages you get — not how many apps you connect — and at typical SMB volume it's a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $3/month at typical SMB volume (about 1,500 messages a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The biggest single line is the per-message label call to Bedrock — and even that is a fraction of a cent each.
- The webhooks, queue, dedupe, and routing are plain code — pennies in total.
- At 5,000 messages a month the bill is around $7. At 10,000 it's around $12.

Cost at three volumes

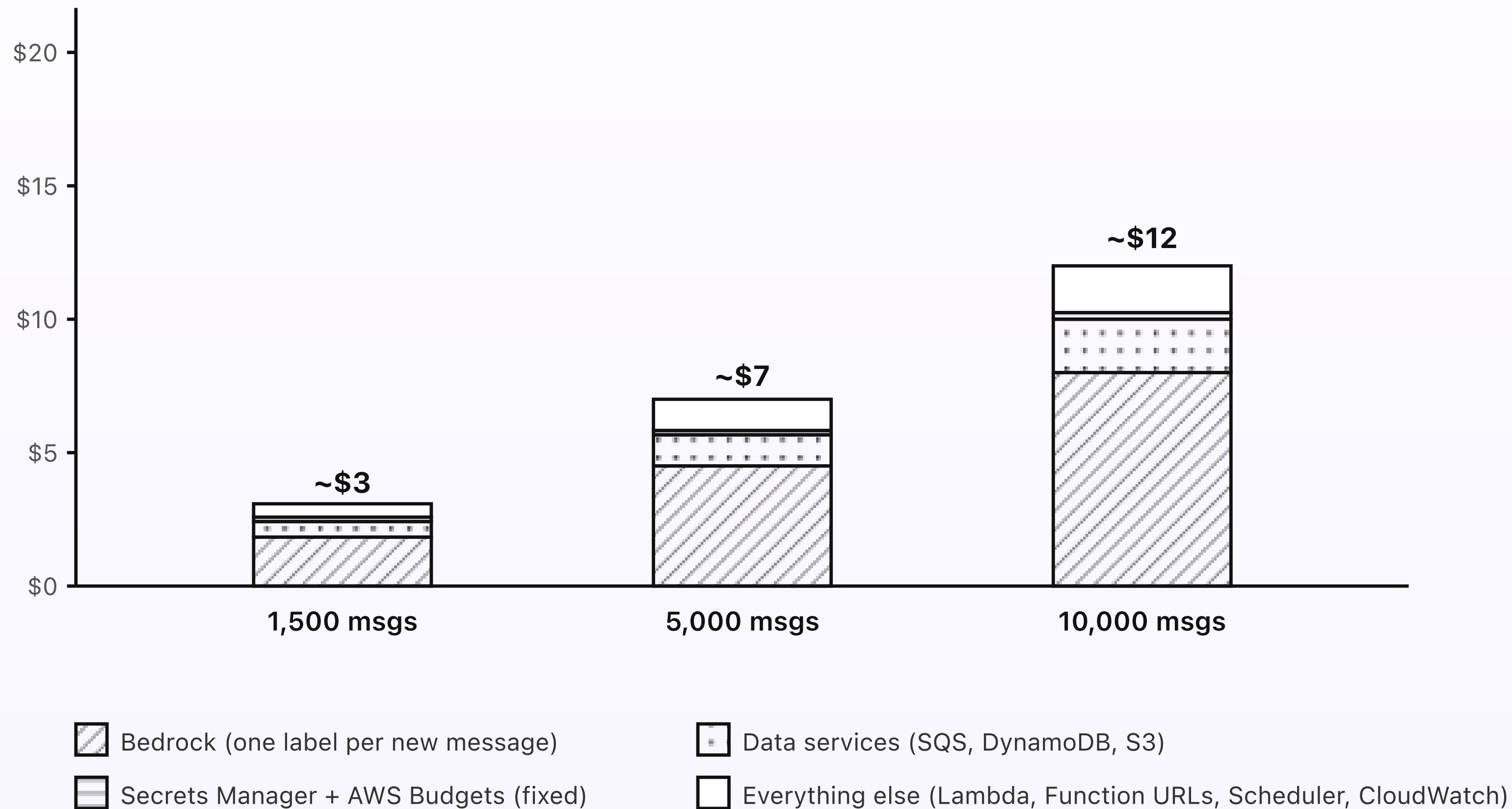


Fig 6. Monthly cost at three message volumes. The Bedrock label call is the dominant slice and grows with message count; the data services and everything-else buckets grow gently, and the fixed costs stay flat.

Where the dollars actually go

Bedrock label call (the bulk). Each new message gets one Haiku 4.5 call — a few hundred input tokens (the message text plus a short instruction) and a few dozen output tokens (the labels). That's a fraction of a cent per message. Multiply by your real new-message count and you get the largest single line on the bill: roughly $1.80 at 1,500 messages, $8 at 10,000. Duplicates don't get a call — the fingerprint check from Part 3 drops them first — so you only pay to label genuinely new messages.

Data services. SQS holds each message briefly: the first million requests a month are free, and an SMB rarely leaves that tier. DynamoDB on-demand backs the dedupe table, the thread state, the per-teammate counts, and the audit trail — a handful of small reads and writes per message. S3 (versioned) keeps message snapshots and any attachments. Together: pennies to a dollar at these volumes.

Lambda + Function URLs (everything else). The connectors, the labeler, the router, and the reply endpoint are all small, short Lambda runs on arm64. The webhooks and the reply endpoint are Lambda Function URLs — no API Gateway in the path. Even at 10,000 messages a month, the compute total stays under a dollar.

EventBridge Scheduler. One daily-digest rule, plus any deferred re-routes when a thread sits too long. A few invocations a day. Pennies.

Secrets Manager + AWS Budgets (fixed). A few stored secrets (one set of platform credentials per channel, plus the Slack token) and one budget alarm. A small fixed amount that doesn't move with volume — the only part of the bill that isn't basically free.

What doesn’t cost money

- **API Gateway.** Replaced by Lambda Function URLs for the webhooks and the reply endpoint.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Everything runs only when a message arrives or a teammate acts.
- **A second, bigger model.** No reply is ever drafted by a model, so there’s no costly generation step — just the one tiny label call.
- **A Knowledge Base.** The system labels and routes; it doesn’t answer from documents. No embeddings, no vector store.

How the cost scales

Almost everything grows with message count, because the work is per-message: one label call, a few DynamoDB rows, a short Lambda run. The fixed costs stay flat no matter how many channels you connect — connecting a fourth platform adds a connector, not a recurring bill. So the cost at 25,000 messages a month is around $28; at 50,000 it’s around $55. Past those volumes you’d look at batching the label calls or caching labels for near-identical messages, but those are tunings for high-traffic accounts — not redesigns.

Set an AWS Budgets alarm at $20/month so anything unusual pages you before the bill matters. The 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 webhook setup, and the SQS config.

PART 7 OF 7

JUNE 12, 2026 PART 7 OF 7 · SOCIAL INBOX UNIFIER SERIES ~8 MIN READ

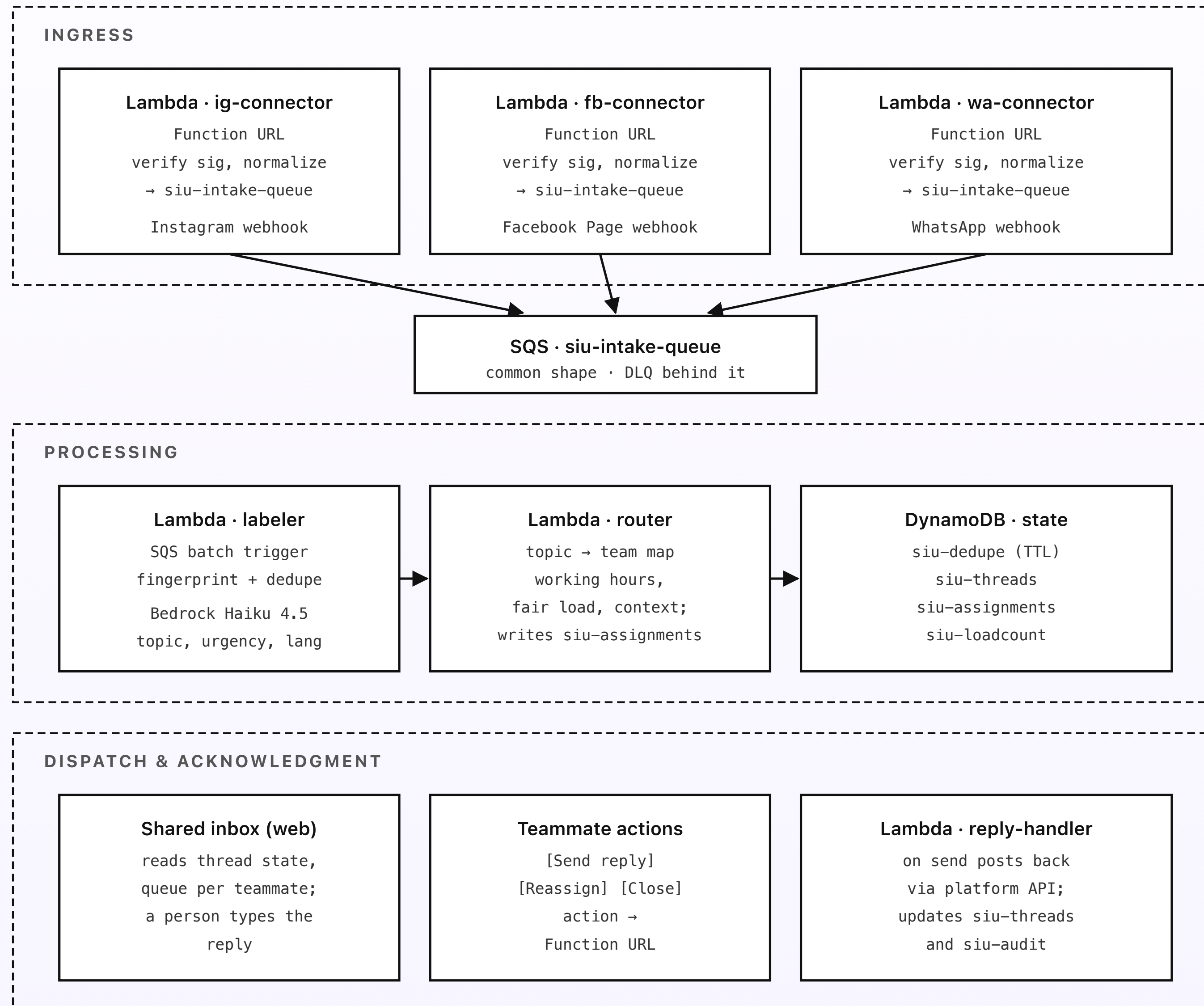
Engineering reference: the social inbox unifier architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the platform webhook setup, the SQS config, the DynamoDB schemas, and the reply flow. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). Bedrock cross-Region inference, SQS, Lambda Function URLs, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is a delayed reply, not a regional outage. One AWS account dedicated to the unifier (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



A human writes every reply — and every interaction is logged to siu-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three connectors into one queue), processing (label, dedupe, route), dispatch and acknowledgment (a person replies and the action is recorded). Every Lambda is event- or request-driven; nothing is synchronous-chained.

Lambda functions

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

- **`ig-connector`** / **`fb-connector`** / **`wa-connector`** — one Lambda per platform, each fronted by a Lambda Function URL (`AuthType: NONE`; signature verified in-handler). On the platform's webhook call, verifies the platform signature against the secret in Secrets Manager (Instagram/Facebook use the `X-Hub-Signature-256` HMAC; WhatsApp uses its app-secret HMAC), parses the platform payload, normalizes to the common message shape, and `SendMessage` to `siu-intake-queue`. Returns 200 immediately so the platform doesn't re-deliver. Each also answers the platform's GET verification handshake on setup. Memory: 256 MB. Timeout: 10 s.
- **`labeler`** — SQS batch trigger on `siu-intake-queue` (batch size 10, partial-batch-response enabled). For each record, computes `fingerprint = sha256(sender|platform|normalized_text)` and does a conditional `PutItem` on `siu-dedupe`; if the item already exists, links the message to the live thread and skips. For new messages, invokes 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 prompt returning `topic`, `urgency`, `language`, `summary`. Writes/updates the thread in `siu-threads`, then invokes `router` (or emits to it via the same handler). Memory: 512 MB. Timeout: 30 s.

- **`router`** — resolves the owning team from the topic→team map in `/siu/config/routing` (Parameter Store), filters candidate teammates by working hours and status from `siu-roster`, picks the least-loaded eligible teammate from `siu-loadcount`, attaches thread/labels/customer-history context, and writes the assignment to `siu-assignments` with an atomic increment of the teammate's open-thread count. Urgent messages bypass the round and go straight to the least-loaded eligible teammate. No Bedrock. Memory: 256 MB. Timeout: 15 s.
- **`reply-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies the shared-inbox session token. Triggered by teammate actions (Send reply / Reassign / Close). On *send*, posts the human-written reply via the originating platform's send API (IG/FB Send API, WhatsApp Cloud API; tokens in Secrets Manager), appends it to `siu-threads`, and writes `action: replied` to `siu-audit`. On *reassign*, updates `siu-assignments` and both teammates' counts. On *close*, marks the thread closed and decrements the count. Memory: 256 MB. Timeout: 15 s. *Never composes reply text — the body comes from the teammate.*
- **`reopen-watcher`** — invoked from the `labeler` path when an incoming fingerprint links to a closed thread; flips the thread back to open, re-runs routing, and writes `action: reopened` to `siu-audit`. Lightweight; folded into the labeler in small deployments. Memory: 256 MB.

- `sla-sweeper` — EventBridge Scheduler target, every 5 minutes. Scans `siu-assignments` for threads past their urgency target (urgent: minutes; normal: hours; from `/siu/config/sla`); re-routes stale threads to the topic's backup teammate and posts a notice to the team Slack channel. No reply is sent. Memory: 256 MB. Timeout: 30 s.
- `digest` — EventBridge Scheduler target, daily at 6pm local. Reads `siu-threads` and `siu-audit` for the day; sends a summary (messages in, answered, still open, anything that breached its target) to a configured Slack channel. No Bedrock; the message is a plain summary table. Memory: 256 MB.

Storage

- **SQS** · `siu-intake-queue` — the work queue holding normalized messages. Visibility timeout 60 s; redrive policy to `siu-intake-dlq` after 5 receives. The DLQ has a CloudWatch alarm on depth > 0.
- **DynamoDB** · `siu-dedupe` — PK `fingerprint`; attributes: `thread_id`, `first_seen`. TTL of a few days so old fingerprints expire and a genuinely new message next month isn't treated as a repeat. On-demand.
- **DynamoDB** · `siu-threads` — PK `thread_id`; attributes: `customer_id`, `platform`, `status` (open/closed), `topic`, `urgency`, `language`, `messages` (ordered list), `assignee`. On-demand. GSI on `(customer_id)` for history lookups.
- **DynamoDB** · `siu-assignments` — PK `thread_id`; sort key `assigned_at`; attributes: `assignee`, `team`, `reason`, `sla_due`. GSI on `(assignee, status)` to render each teammate's queue. On-demand.

- **DynamoDB** · **`siu-loadcount`** — PK `assignee`; attribute: `open_threads` (atomic counter). On-demand.
- **DynamoDB** · **`siu-roster`** — PK `assignee`; attributes: `topics`, `working_hours`, `status`, `backup`. On-demand. Editable from the inbox admin screen.
- **DynamoDB** · **`siu-audit`** — one row per write action of any kind. PK `(thread_id, ts)`; attributes: `action`, `by_user`, `before`, `after`, `notes`. On-demand. No TTL — this is the long-term audit trail.
- **S3** · **`siu-message-snapshots`** — raw normalized message snapshots and any inbound attachments (images, voice notes). Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 2 years.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. One callsite: `labeler`, for the per-message label set. No callsite ever generates customer-facing text.
- **Heavier model.** Not used on the hot path. Claude Sonnet 4.6 (`anthropic.claude-sonnet-4-6-...`) is wired only as an optional offline reviewer for tuning the topic taxonomy from a sample of mislabeled threads — run by hand, not in production traffic.
- **Embeddings.** Not used. The system labels and routes; it does not answer from documents. No Knowledge Base, no S3 Vectors.

- **Quotas.** Default account quotas are more than enough at SMB volume. One small call per new message; duplicates never reach the model.

Platform webhooks

- **Instagram / Facebook.** App subscribed to the `messages` webhook field on the Page/IG account. Callback URL is the connector's Function URL; verify token stored in Secrets Manager for the GET handshake; payloads HMAC-signed with the app secret (`X-Hub-Signature-256`), verified in-handler.
- **WhatsApp.** Cloud API webhook subscribed to `messages`; callback URL is the `wa-connector` Function URL; app-secret HMAC verified in-handler; phone-number ID and access token in Secrets Manager.
- **Reply path.** Outbound uses each platform's send API with the same stored tokens. Note WhatsApp's 24-hour customer-care window: replies outside it require an approved template, which the inbox surfaces to the teammate rather than silently failing.

IAM (least privilege per Lambda)

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

- **connector roles:** `sqs:SendMessage` on `siu-intake-queue`; `secretsmanager:GetSecretValue` on that platform's secret only. *No* DynamoDB, *no* `bedrock:*`.
- **labeler role:** `sqs:ReceiveMessage` + `DeleteMessage` on the queue; `dynamodb:PutItem` (conditional) on `siu-dedupe`; `dynamodb:PutItem` +

`UpdateItem` on `siu-threads`; `bedrock:InvokeModel` on the Haiku ARN; `lambda:InvokeFunction` on `router`.

- **router role:** `dynamodb:GetItem` on `siu-roster` and `siu-loadcount`; `dynamodb:PutItem` + `UpdateItem` on `siu-assignments` and `siu-loadcount`; `ssm:GetParameter` on `/siu/config/*`. *No* `bedrock:*`.
- **reply-handler role:** `dynamodb:UpdateItem` on `siu-threads`, `siu-assignments`, `siu-loadcount`; `dynamodb:PutItem` on `siu-audit`; `secretsmanager:GetSecretValue` on the platform send-token secrets; outbound network to `graph.facebook.com` and the WhatsApp Cloud API host.
- **sla-sweeper / digest roles:** `dynamodb:Query` on `siu-assignments` / `siu-threads`; `secretsmanager:GetSecretValue` on the Slack token; `ssm:GetParameter` on `/siu/config/sla`.

SQS and EventBridge Scheduler config

- **`siu-intake-queue`** — standard queue; visibility timeout 60 s; `maxReceiveCount` 5 to `siu-intake-dlq`. Labeler event-source mapping batch size 10, max batching window 5 s, partial-batch-response on.
- **`siu-sla-sweep`** — `rate(5 minutes)`. Target: `sla-sweeper` Lambda.
- **`siu-daily-digest`** — `cron(0 18 * * ? *)` in `TZ_NAME`. Target: `digest` Lambda.
- **One-off rules** — not needed on the hot path; re-routes are handled by the periodic sweeper rather than per-thread schedules, which keeps the Scheduler surface tiny.

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:** `siu-intake-dlq` depth > 0 (a message failed to process); labeler error rate > 1% in 24h; connector signature-verification failures > 5/hour (might mean a platform secret rotated); reply-handler send failures > 0 (a reply didn't reach the customer).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $20/month threshold, alarm at 80% and 100%, posts to SNS topic `siu-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Platform credentials live in Secrets Manager: `siu/ig/*`, `siu/fb/*`, `siu/wa/*` (app secret, verify token, page/phone-number ID, send token per platform). The Slack token for digests and SLA notices is under `siu/slack/bot-token`. The topic→team routing map, the urgency SLA targets, quiet/working-hours defaults, and the dedupe window all live in Parameter Store under `/siu/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 connector Function URLs and their secrets as a separate stack (a rotated platform secret shouldn't force a full redeploy), turn on S3 versioning for `siu-message-snapshots`, set the dedupe TTL deliberately (too short re-opens duplicates, too long can swallow a real new message), and keep the topic taxonomy in Parameter Store so changing it never needs a deploy. Total deployable surface: around eight Lambdas, one SQS queue plus its DLQ, seven DynamoDB tables, one S3 bucket, two EventBridge 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.