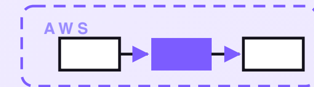
7-PART SERIES · FREE COMPANION



Ticket router

A serverless router that reads each new support ticket, works out the topic and how urgent it is, tags it, and routes it to the right team or person — flagging anything angry or urgent to jump the queue. It only sorts and routes; it never answers the customer itself. A misroute is one click for a human to fix, and the router learns the correction. 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/ticket-router

CONTENTS

Ticket router

01 A ticket router on AWS for a few dollars a month
02 How a ticket gets read
03 How a ticket gets routed
04 How a ticket reaches the right team
05 How a misroute gets corrected
06 What the ticket router costs
07 Engineering reference: the ticket router architecture

PART 1 OF 7

MAY 30, 2026 PART 1 OF 7 · TICKET ROUTER SERIES ~5 MIN READ

A ticket router on AWS for a few dollars a month

A small support team gets more tickets than anyone can sort by hand. The billing question that should go to finance, not the help desk. The angry email from a customer whose site is down, sitting behind forty routine how-do-I questions. The feature request that nobody on support can answer. The same customer who emails, fills in the web form, and pings chat about the same problem. Sorting all of that — reading each one, deciding what it's about, judging how urgent it is, and getting it to the right person — eats the morning before a single reply goes out. This post walks through the design of a small router that reads each new ticket, tags it, and sends it to the right team — and flags anything angry or urgent so it jumps the queue. It never answers the customer itself; a person always does that.

KEY TAKEAWAYS

- Three sources for tickets: an email inbox, a web form on your site, and a chat lane — all land as one ticket.
- Every ticket ends in one of four moves: route, priority, escalate, or hold for a human to look at.
- One small model call per ticket reads the topic, the urgency, and the tone. Nothing else uses a model.
- The router only sorts and routes. It never writes a reply to the customer. A person always 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.

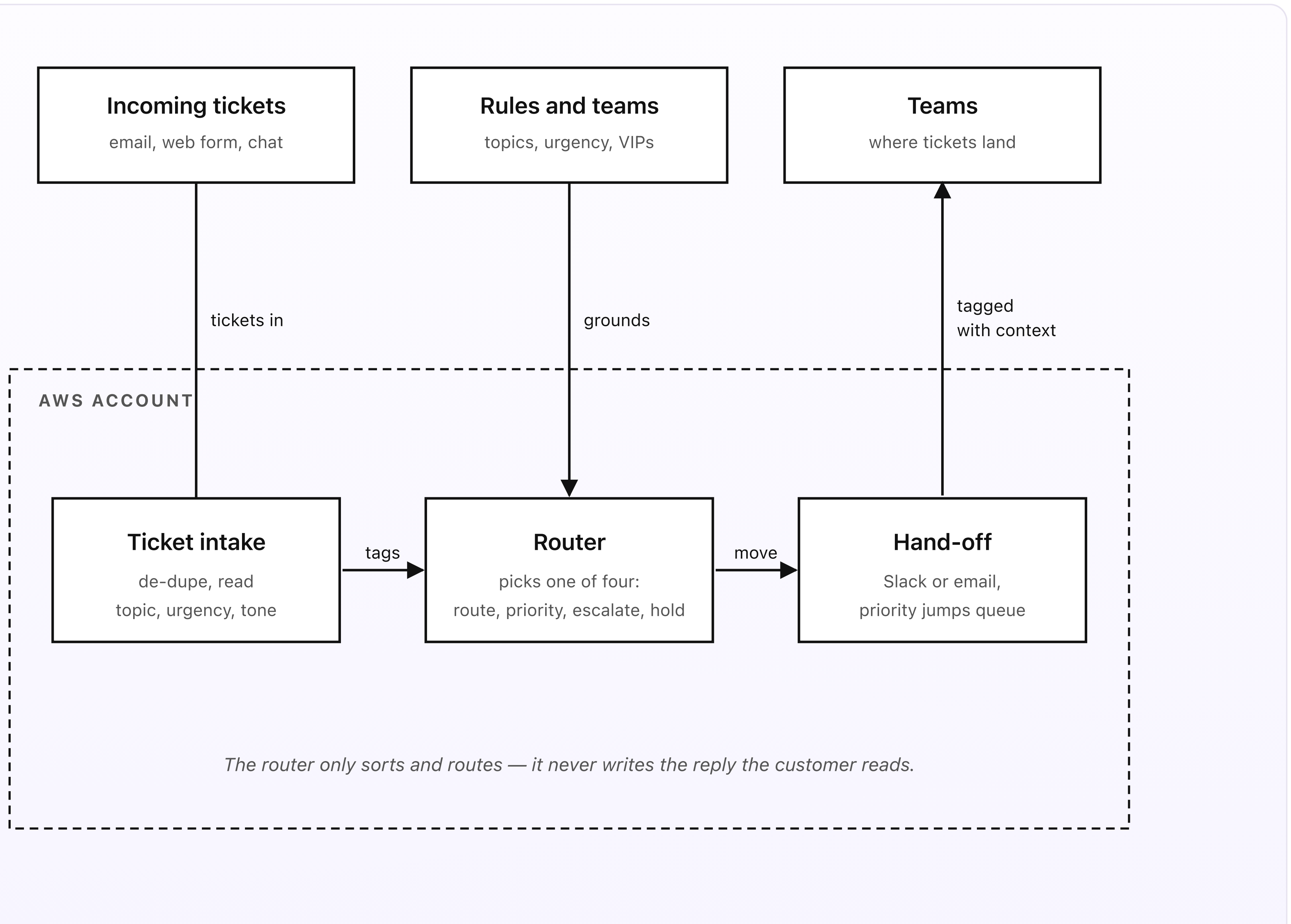

Incoming tickets
email, web form, chat
Rules and teams
topics, urgency, VIPs
Teams
where tickets land
tickets in
grounds
tagged
with context
AWS ACCOUNT
Ticket intake
de-dupe, read
topic, urgency, tone
tags
Router
picks one of four:
route, priority, escalate, hold
move
Hand-off
Slack or email,
priority jumps queue
The router only sorts and routes — it never writes the reply the customer reads.

Fig 1. Three sources outside, three pieces inside AWS. Tickets flow in from an email inbox, a web form, and a chat lane. The Router reads each one and picks one of four moves. The Hand-off sends the right ticket to the right team, with the urgent ones jumping the queue.

What you set up once (the outside)

- **Incoming tickets.** Three ways a ticket can arrive, all covered in Part 2 — an email inbox (forward your support address into a dedicated one and every email becomes a ticket), a web form on your site (the “Contact support” form posts straight to a small endpoint), and a chat lane (your chat tool drops finished conversations in as tickets). Whichever way it comes in, it ends up as one ticket with the customer, the subject, and the message body.
- **A rules folder.** One Google Sheet and one short Google Doc in a Drive folder. The *sheet* maps each topic to a team or person — “billing → finance, login problems → support, feature requests → product.” The *doc* covers the words that mean urgent (outage, down, refund, deadline), the VIP customer list (whose tickets always jump the queue), and the priority rules. The doc also holds a handful of labelled examples — a few real tickets with the right topic next to each — that help the router pick well.
- **Teams.** The people who actually answer. Each team has a Slack channel (so the ticket lands where the team already works) or, if Slack isn’t set up for them, a shared inbox. Tickets land with the customer name, the topic, the urgency, the tone, the original message, and a one-click “Reassign” button if the router got it wrong.

What runs on every ticket (the inside)

- **The ticket intake.** Three sources feed one queue. Each new ticket is written once, given an id, and checked against the last few tickets from the same customer so three copies of the same problem become one. Then a single Bedrock Haiku 4.5 call reads the ticket and returns three things: the topic (which bucket it belongs in), the urgency (how soon someone has to act), and the tone (is the customer calm, confused, or angry). The model reads; it does not reply.
- **The router.** Takes the topic, urgency, and tone and turns them into a move. It reads the rules sheet to find the team for that topic, checks the VIP list and the priority rules in the doc, and picks one of four moves. *Route*: normal ticket — send it to the right team's normal queue. *Priority*: urgent or from a VIP — send it to the same team but put it at the top and ping the lead. *Escalate*: angry and urgent, or a VIP outage — flag it loudly so a manager sees it right away. *Hold*: the topic is unclear or the ticket is half-empty — drop it in a small triage lane for a human to sort, rather than guess. The move logic is plain Python; the only model call already happened on intake.
- **The hand-off.** Sends the tagged ticket to the team the router chose. Slack messages go to the team channel; email goes through SES outbound to a shared inbox. Priority and escalate tickets go to the top of the list and ping the lead. Every hand-off writes a row in DynamoDB so you can see where each ticket went and how long it waited. A weekly digest shows volume by topic and how often the router was corrected; a monthly summary writes a short paragraph: busiest topics, slowest queues, and the corrections that taught the router something.

In plain words

A customer emails at 8:54am: “Our store has been down for an hour, we’re losing sales, please help NOW.” The intake writes the ticket and reads it: topic is *outage*, urgency is *high*, tone is *angry*. The customer is on the VIP list. The router picks *escalate*: the ticket lands at the top of the engineering channel and pings the on-call lead in Slack, with the full message attached. Meanwhile a second customer asks “how do I change my billing address?” — topic *billing*, urgency *low*, tone *calm* — and that one routes quietly to the finance queue. Nobody had to read either ticket to sort it. And in both cases, a person writes the actual reply; the router never speaks to the customer.

The cost of running this is about $3 a month at SMB volume. The cost of *not* running it is the outage email that sat unread behind forty routine questions, the billing query that bounced between three people before reaching finance, and the angry customer who waited because nobody spotted that they were angry.

DESIGN RULES THAT SHAPED EVERY DECISION

- The router sorts and routes; it never answers the customer. The reply stays fully in human hands.
- Four moves, always. Route, priority, escalate, hold. There is no fifth.
- When the topic is unclear, the router holds for a human instead of guessing. A wrong route is worse than a held one.
- Angry, urgent, and VIP tickets jump the queue. The team sees them first, with full context.
- The rules live in Drive. Adding a topic, changing a team, or tuning the urgency words doesn't need a deploy.
- Every route and every correction is logged. A misroute is one click to fix, and the router learns from it.

Why this shape

Most teams sort tickets one of three ways: a shared inbox where whoever's free grabs the next one, a single "support" queue that a person triages by hand each morning, or a pile of rules in the help-desk tool that nobody maintains. The shared inbox works until it's busy — and the busiest mornings are exactly when the urgent ticket gets buried. The hand-triage works until the person doing it is out. And the rules-in-the-tool approach breaks the first time a customer phrases something in a way the keyword rule didn't expect.

The setup above keeps the rules in a sheet the team already edits, but adds a small system that *reads* each ticket the way a person would — getting the gist, not just matching keywords — and acts on it the same second it arrives. Urgent tickets surface immediately. Unclear ones go to a human instead of the wrong team. And every correction makes the next sort a little better. The router is invisible on the easy tickets; it earns its place on the messy ones and the urgent ones.

The next four posts walk through each piece in turn: how a ticket gets read, how a ticket gets routed, how a ticket reaches the right team, and how a misroute gets corrected. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 30, 2026 PART 2 OF 7 · TICKET ROUTER SERIES ~4 MIN READ

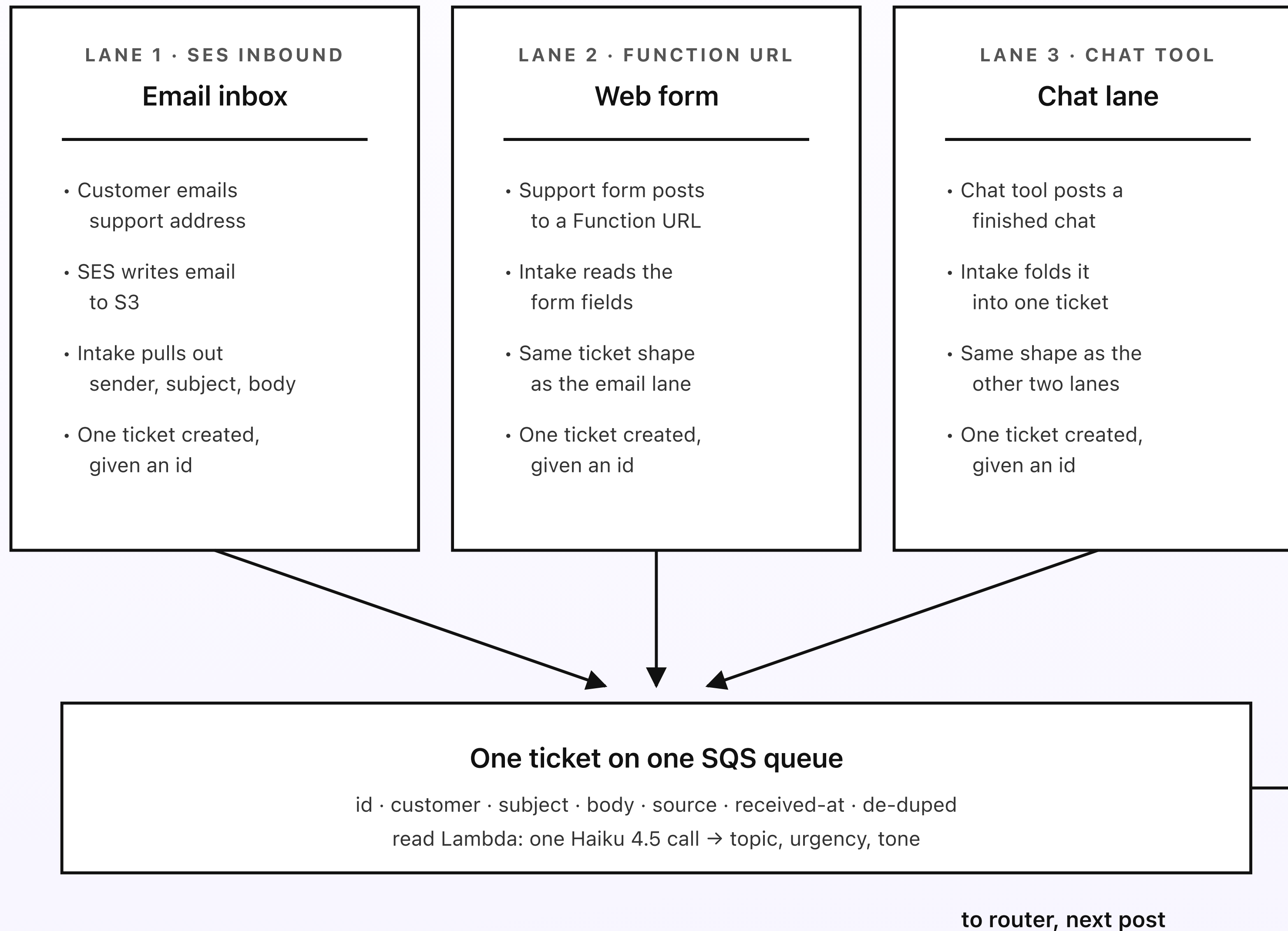
How a ticket gets read

The router can only sort what reaches it. So the first job is turning every way a customer can ask for help into one clean ticket. There are three ways a ticket gets in: a customer emails your support address, fills in the form on your site, or sends a chat message. The first is the most common. The other two exist because customers reach for whatever's in front of them — and the same person will often try all three about the same problem. Once a ticket is in, one small model call reads it: what's it about, how urgent is it, and how does the customer sound?

KEY TAKEAWAYS

- Three intake lanes feed one queue: an email inbox, a web form, and a chat lane.
- Each new ticket is written once, given an id, and checked against recent tickets so duplicates merge.
- One Bedrock Haiku 4.5 call reads each ticket and returns topic, urgency, and tone.
- The model only reads. It never writes a reply — that stays with a person.
- A burst of tickets queues up in SQS so nothing is dropped when a busy hour hits.

Three lanes into one queue



Every lane becomes one ticket on one queue — and the model only reads it, never the reply.

Fig 2. Three lanes converge on one queue. Email, web form, and chat all become the same shape of ticket. A read Lambda takes each one off the queue and makes a single model call to tag the topic, urgency, and tone.

Lane 1: the email inbox (the lane most teams live in)

Set up a dedicated inbound address — something like `support@your-company.com` — via Amazon SES. SES is Amazon's email service; here it's receiving mail, not sending it. When a customer emails that address, SES writes the raw email to `s3://tr-raw-mail/`. The S3 PUT triggers an intake Lambda. The Lambda reads the email, pulls out the sender, the subject, and the body (and strips the long quoted history off replies so the router reads the new message, not the whole thread), and creates one ticket with an id.

This lane covers the bulk of real support. People email. They reply to their own ticket. They forward something a colleague sent. The intake treats a reply to an existing ticket as part of that ticket, not a brand-new one, by matching the email's thread headers against tickets already in flight.

Lane 2: the web form

The "Contact support" form on your site posts straight to a Lambda Function URL — a plain web address that runs a Lambda, with no API Gateway in front of it. The form sends the customer's name, email, a subject, and a message. The intake reads those fields and creates a ticket in exactly the same shape as the email lane, so everything downstream treats the two identically. A simple shared secret on the form post keeps random internet traffic from creating junk tickets.

The form lane is worth having because a customer on your pricing page who hits a problem will use the form in front of them rather than hunt for an email address. Catching them there means the ticket arrives with the page they were on already attached.

Lane 3: the chat lane

If you run a chat widget, finished conversations can become tickets too. When a chat ends without resolution — the visitor left, or the chat tool couldn't answer — the tool posts the conversation to another Function URL. The intake folds the back-and-forth into one ticket body so the router reads it like any other message. This is the most optional of the three lanes; a team without chat loses nothing.

Reading the ticket: one small model call

However a ticket arrived, it lands on an SQS queue. SQS is a simple waiting line for work; if a hundred tickets arrive in a minute, they line up instead of overwhelming anything. A read Lambda takes tickets off the queue one at a time and makes a single Bedrock Haiku 4.5 call. The prompt is short: "Read this support ticket. Return JSON only. Give the topic from this list, an urgency of low, medium, or high, and a tone of calm, confused, or angry. If you are unsure of the topic, say unsure." The list of topics comes from the rules sheet, and a handful of labelled examples are included so the model has a few real tickets to pattern-match against.

That one call is the only place a model touches a ticket. It reads — topic, urgency, tone — and writes those three tags back onto the ticket. It does not draft a reply, suggest an answer, or message the customer. Everything after this point is plain Python working from those three tags, which is what the next post covers.

Why everything funnels to one queue

Three lanes in, but only one queue the router reads from. That's deliberate. If each lane routed on its own, "why did this ticket go there?" would mean checking three code paths. Funneling everything to one queue means there is exactly one ticket per request, in one shape, read by one model call, sorted by one set of rules. The lanes are first-class for getting tickets in, but they all become the same ticket on the way.

Next post: how the router takes the topic, urgency, and tone and turns them into one of four moves.

PART 3 OF 7

MAY 30, 2026 PART 3 OF 7 · TICKET ROUTER SERIES ~5 MIN READ

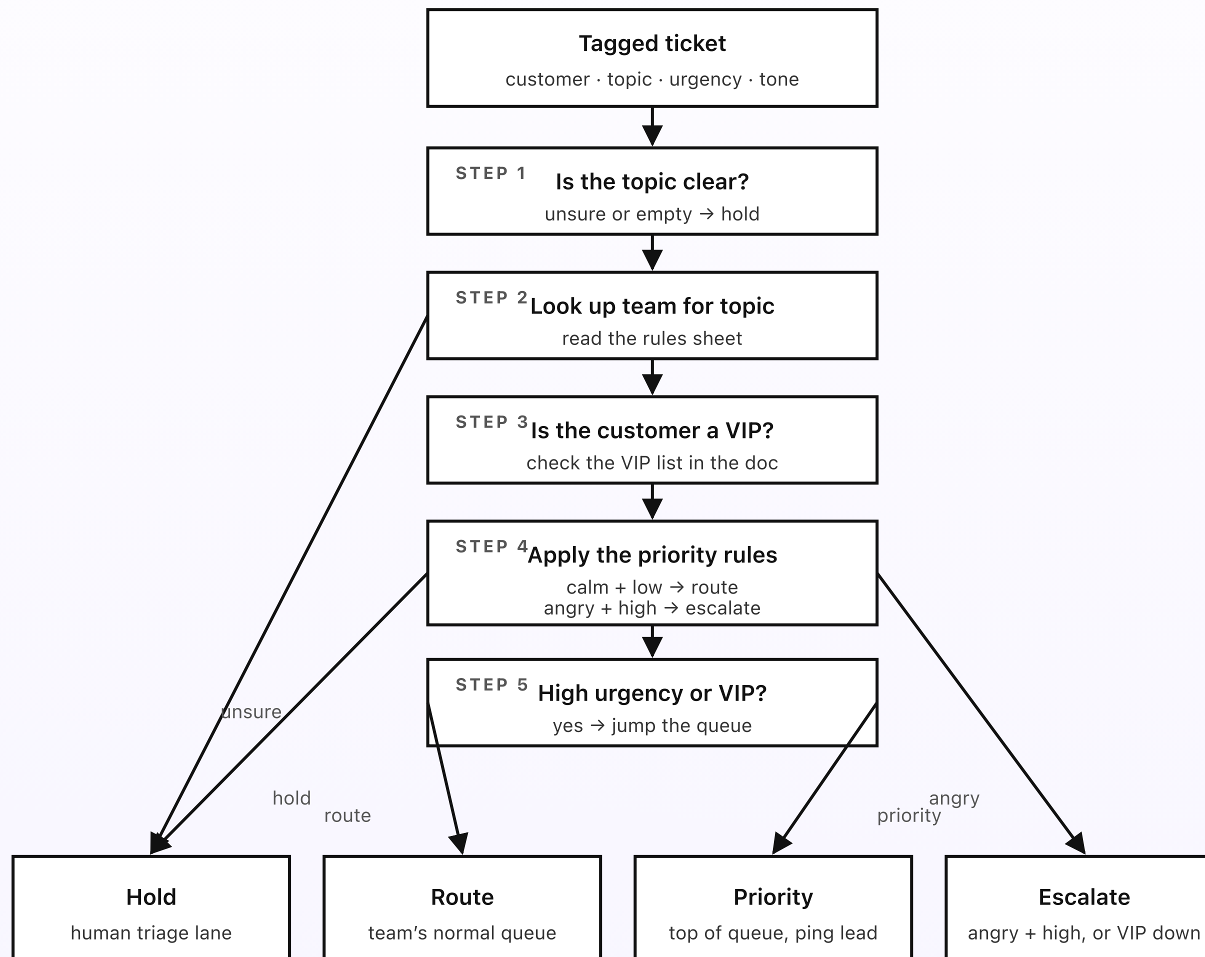
How a ticket gets routed

The read step handed the router three tags: a topic, an urgency, and a tone. Now the router has to turn those into a decision — which team, and how loud. It reads one ticket at a time, looks up the team for the topic, checks the VIP list and the priority rules, and picks one of four moves. The whole decision is plain Python. The model call already happened on intake. Every rule lives in the Drive folder, where a manager can edit it without a deploy.

KEY TAKEAWAYS

- The router runs once per ticket, right after the read step tags it.
- The topic-to-team map lives in the rules sheet; the urgency words and VIP list live in the rules doc.
- Four moves per ticket: route, priority, escalate, or hold.
- An unclear topic or a half-empty ticket goes to *hold* — a human triage lane, never a guess.
- The router never calls a model. The decision is entirely from the rules.

The decision flow, per ticket



The rules doc holds every rule — change one and the next ticket uses the new value.

Fig 3. The router's decision tree, per ticket. Five steps decide which of four moves applies and which team it goes to. The rules hold every threshold; the router only enforces them.

Topic to team: the map is in the sheet

The rules sheet has one row per topic. Each row names the topic and the team that owns it: "billing → finance, login problems → support, bug report → engineering, feature request → product, sales question → sales." When the read step tags a ticket with a topic, the router looks up that row and gets the team. If a topic has no row — someone added a new topic to the model's list but not the sheet — the router treats it as unclear and holds the ticket rather than dropping it somewhere arbitrary.

Per-customer overrides exist too. The VIP list can pin a named customer to a named person ("Acme always goes to Dana"), so a key account's tickets skip the general queue and land with the person who knows them. That override beats the topic map for that customer.

Four moves, always

Every ticket lands in exactly one of four buckets. The names are plain on purpose.

- **Route.** A normal ticket — clear topic, calm tone, nothing urgent. Send it to the chosen team's normal queue. Most tickets, most days, route. Nobody is pinged; it just shows up in the right place.

- **Priority.** The urgency is high, or the customer is a VIP, but the tone is still civil. Send it to the same team, but put it at the top of their list and ping the team lead so it isn't missed. A payment that failed, a deadline mentioned, a key account asking — these are priority.
- **Escalate.** Angry and urgent together, or a VIP reporting an outage. Flag it loudly — the ticket goes to the top and a manager is pinged directly, not just the team lead. This is the small set of tickets where being a minute late is genuinely costly, so the router makes noise on purpose.
- **Hold.** The topic came back *unsure*, or the ticket is nearly empty ("help" with no detail), or it looks like spam. Don't guess. Drop it in a small triage lane where a human glances at it and sends it on. A wrong route costs more than a held ticket, because a wrong route looks handled when it isn't.

Why the routing decision uses no model

The router could call a model again to pick the team. It doesn't. Two reasons. First, the routing decision should be the one part a manager can fully predict — if the sheet says billing goes to finance, it goes to finance, every time. A model in that loop would add variance the team can't reason about, and "why did this go there?" would have no clean answer. Second, the reading is already done; the topic, urgency, and tone are in hand. Turning three tags into a team is a lookup and a few `if` statements, not a job for a model.

So Bedrock fires exactly once per ticket — on the read step in Part 2. The router itself is plain Python that reads a sheet, checks a list, and picks a move. That keeps the routing cheap, fast, and explainable.

What the router writes down

Every decision is recorded. The router writes a row to the `tr-routes` DynamoDB table: `(ticket_id, topic, urgency, tone, team, move, decided_at)`. That row is what the hand-off step reads to know where to send the ticket, and what the weekly digest counts to show volume by topic and team. It's also the "before" picture if a human reassigns the ticket later — which is exactly what Part 5 covers.

Next post: how the chosen move actually reaches the right team — the channel, the queue position, the context attached, and the guardrails on every hand-off.

PART 4 OF 7

MAY 30, 2026 PART 4 OF 7 · TICKET ROUTER SERIES ~5 MIN READ

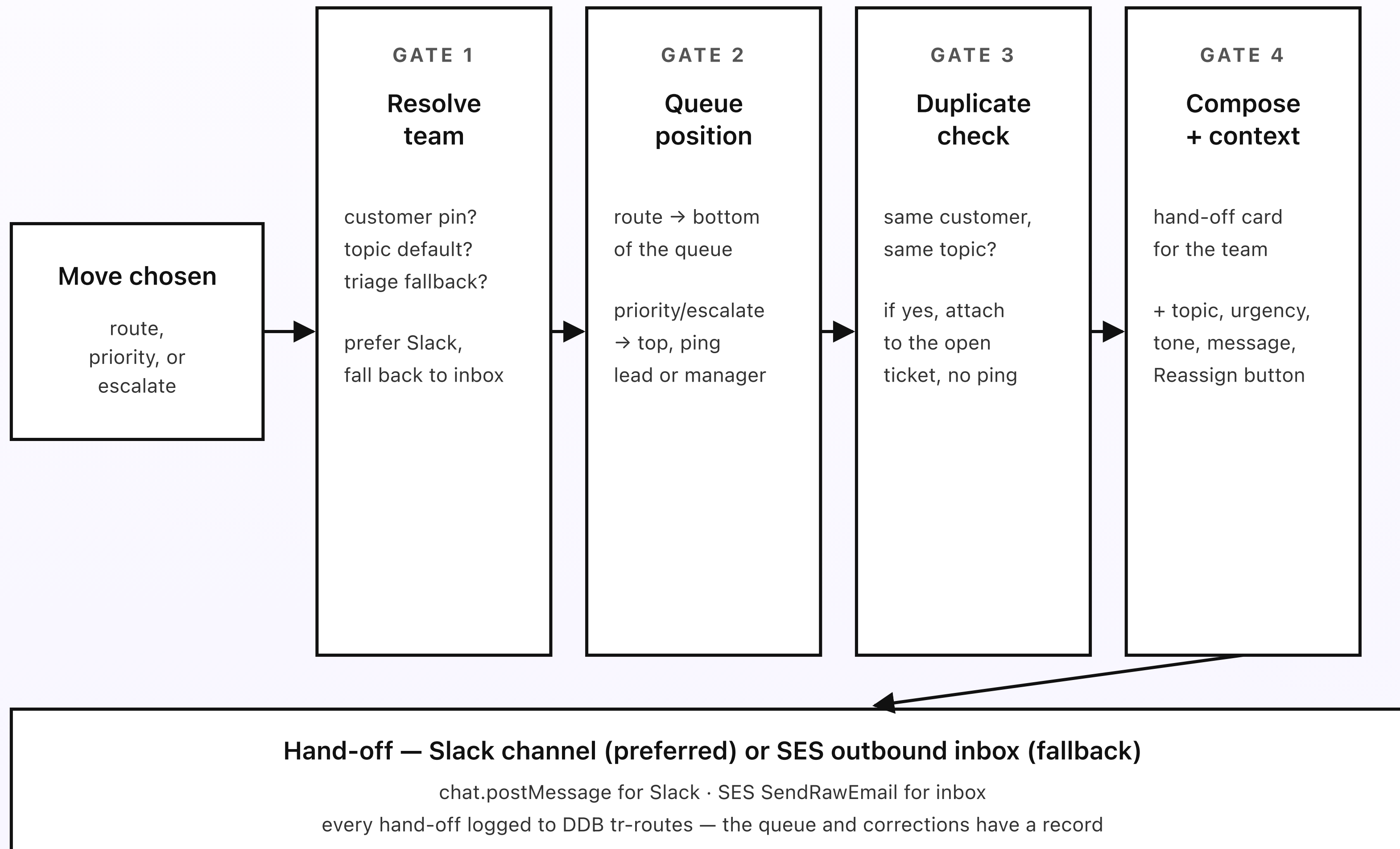
How a ticket reaches the right team

The router picked a move — route, priority, or escalate. Now the hand-off has to figure out which channel, at what queue position, with what context attached, and without pinging the same team twice for one problem. Get any of those wrong and the hand-off is worse than none: an urgent ticket buried in a normal queue, a duplicate that fires three Slack pings, a hand-off with no detail that makes someone open four tabs to understand it. Four small guardrails sit between the move and the ticket landing.

KEY TAKEAWAYS

- Team resolution: per-customer override beats per-topic default beats fallback to the general triage lane.
- Slack channels are the default; a shared inbox via email is the fallback if no channel is set.
- Priority and escalate tickets jump the queue and ping the lead or a manager.
- Every hand-off ships with the customer, topic, urgency, tone, the original message, and a Reassign button.
- A duplicate check stops the same problem from pinging a team more than once.

Four guardrails on every hand-off



Every gate is a deterministic check — no model calls, no surprise pings on a busy morning.

Fig 4. Four guardrails between the move and the landed ticket. Resolve the team. Set the queue position. Skip duplicates. Compose with full context. Then ship via Slack or email and log the hand-off so the queue and the corrections both have a record.

Gate 1: resolve the team

Three places the hand-off looks for the team, in order. First, the per-customer override in the VIP list — if a row pins a named customer to a named person, that wins regardless of topic. Second, the per-topic team from the rules sheet ("billing goes to finance"). Third, the general triage lane — the catch-all that a human watches. The triage fallback should rarely fire in steady state; if it does, the weekly digest names every ticket that hit it so the rules sheet can be filled in.

Once the hand-off knows which team, it looks up where the team works. The rules sheet maps each team to a Slack channel if one is set, otherwise to a shared inbox email. Slack is preferred because the team already lives there and a Slack card with a Reassign button is more useful than a forwarded email. Email is the fallback so a team without Slack still gets its tickets.

Gate 2: queue position

The move decides where the ticket lands in the team's list. A *route* move drops the ticket at the bottom of the normal queue — it shows up, quietly, in turn. A *priority* move puts it at the top and pings the team lead, so the team sees it before the routine pile. An *escalate* move also goes to the top, but pings a manager directly, because escalate is the small set of tickets where a minute matters.

The pings are deliberate and rare. A team that gets pinged for every ticket learns to ignore the pings; a team that gets pinged only for priority and escalate learns that a ping means “look now.” Keeping route silent is what makes the priority ping mean something.

Gate 3: duplicate check

Customers repeat themselves. The same person emails, then fills in the form, then pings chat about one outage. Without a guard, that’s three tickets and three pings for one problem. Gate 3 looks at recent tickets from the same customer about the same topic; if this ticket is a repeat of one already handed off and still open, it attaches the new message to that ticket instead of firing a second ping. The team sees “customer followed up” on the existing card, not a fresh duplicate.

The check is conservative on purpose. It only merges when the customer and the topic both match a still-open ticket within a short window. A genuinely new problem from the same customer — different topic, or a ticket already closed — starts its own card. Merging too aggressively would hide real tickets; merging carefully just removes the obvious echoes.

Gate 4: compose with full context, then ship

The hand-off card carries everything the team needs to start without asking a follow-up question: the customer name, the topic the router picked, the urgency, the tone, the original message in full, and the source it came from. It also carries a *Reassign* button — one tap to move the ticket to a different team if the router got it wrong. For Slack, the card is posted via the chat API with the button as a Block Kit

action. For the email fallback, the same fields are wrapped in a short HTML email, and the Reassign button is a link to a Function URL that records the change.

An escalate move adds a line the others don't: why it escalated — "angry tone, outage mentioned, VIP customer" — so the manager sees the reason at a glance instead of having to reread the whole ticket to judge it.

Every hand-off — Slack or email, route or escalate — writes a row to `tr-routes` in DynamoDB. That row is what the queue reads and what a correction later updates.

Why the guardrails exist

None of these gates are clever. They're the small care a thoughtful person would take if they were handing tickets to teams by hand — check who actually owns this, put the urgent ones on top, don't hand the same thing over twice, include enough context that the team can start straight away. Putting them in code as four small sequential gates makes them part of the design, not something you're trusting a busy morning to remember.

Next post: how a misroute gets corrected — the Reassign button, the priority bump, the split, and how each correction teaches the router to do better next time.

PART 5 OF 7

MAY 30, 2026 PART 5 OF 7 · TICKET ROUTER SERIES ~5 MIN READ

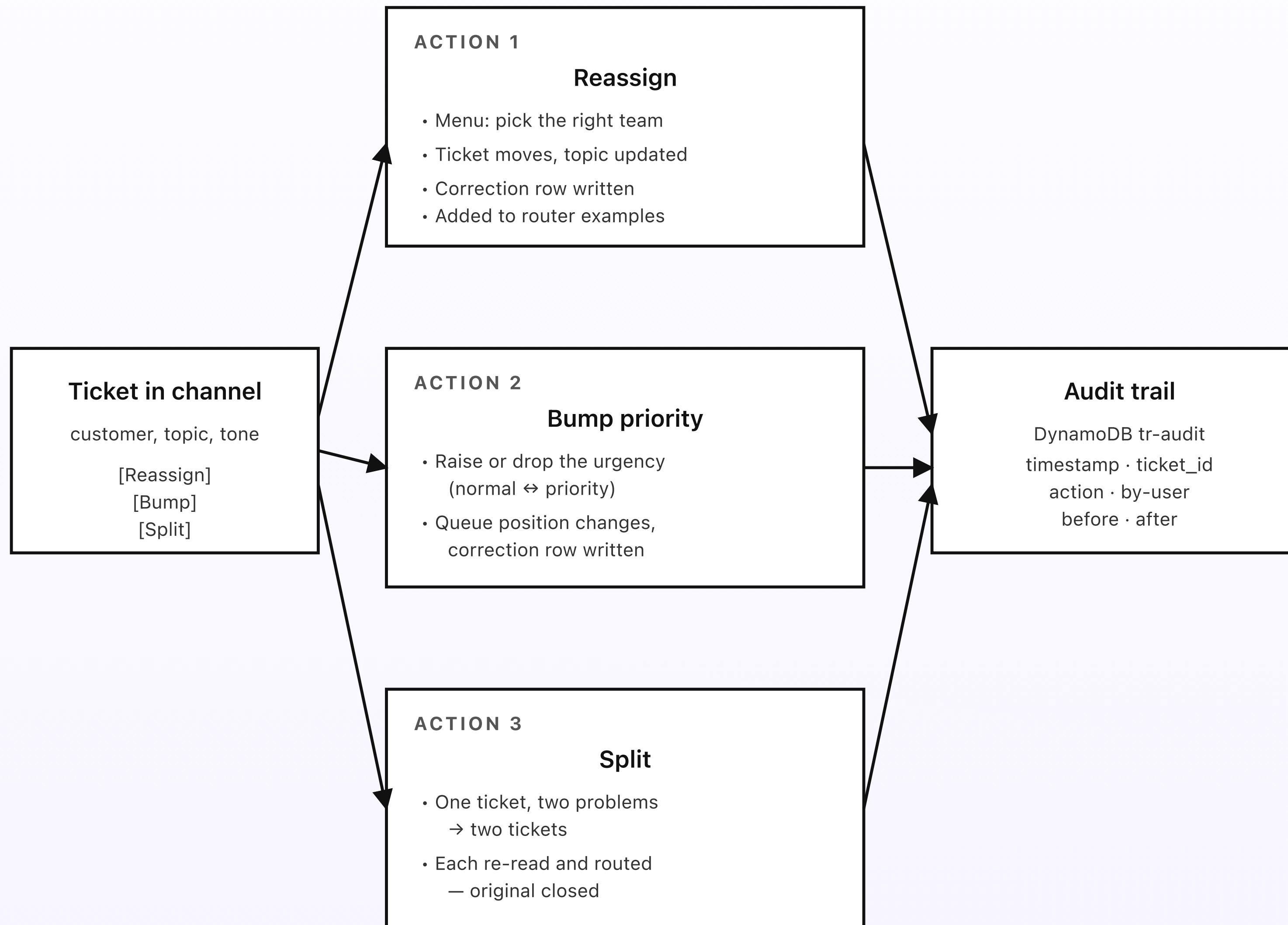
How a misroute gets corrected

A ticket lands in the support channel at 9:12am. It’s really a billing question; the router read it as a login problem. There’s a Reassign button. What happens when someone taps it? The honest answer is “it depends what they meant to fix.” This post walks through the three things a human can do to a routed ticket — reassign, bump priority, split — and how the ticket, the queue, and the router’s own examples all stay in sync, so the same mistake gets less likely next time.

KEY TAKEAWAYS

- Three actions on a routed ticket: *reassign* (move it, teach the router), *bump priority* (raise or drop urgency), *split* (one ticket, two topics).
- Each action updates the ticket, moves it in the queue, and writes an audit row.
- A reassign records the original tag and the corrected tag side by side.
- Corrections feed the router's labelled examples, so the next sort is a little better.
- The Reassign button is a Slack interactive message backed by a Function URL.

Three actions on a routed ticket



A correction doesn't just fix this ticket — it teaches the router, so the same misroute gets less likely.

Fig 5. Three actions on a routed ticket, three different effects. Reassign moves it and teaches the router. Bump raises or drops the urgency. Split turns one ticket into two. Every action writes to the audit trail.

Action 1: reassign (the most common)

Someone in the support channel reads the ticket, sees it's really a billing question, and taps *Reassign*. A small Slack menu opens with the list of teams. They pick *Finance* and confirm.

The confirm submits to a Function URL Lambda. Three things happen, in order. First, the ticket moves: it's removed from the support queue and placed in the finance queue, and the topic on the ticket is updated from *login* to *billing*. Second, a correction row is written to `tr-corrections` with the original topic and team, the corrected topic and team, the customer, and the person who made the change. Third, an `action: reassign` row is written to `tr-audit` with the before-and-after snapshot.

The correction is the part that matters most. That row — "this ticket, which read like login, was actually billing" — gets added to the router's labelled examples, the same handful of real tickets the read step shows the model in Part 2. The next time a ticket arrives that looks like this one, the model has a closer example to match against. The router doesn't retrain; it just gets a better set of examples, refreshed from real corrections.

Action 2: bump priority (raise or drop the urgency)

Sometimes the team is wrong — the router read it right — and sometimes the router misjudged how urgent something was. A ticket that read calm turns out to be a customer about to churn; a ticket that read urgent turns out to be someone who writes everything in capitals. Either way, a human can fix the urgency without touching the team.

Bump opens a small choice: raise to priority, or drop to normal. On confirm, the Function URL Lambda changes the urgency on the ticket and moves it in the queue — up to the top and a ping to the lead if raised, down into the normal pile if dropped — and writes a row to `tr-corrections` noting the urgency change. Like reassigns, repeated urgency corrections on a kind of ticket feed back into the examples, so the router learns that "customers who say X are usually more urgent than they sound."

Action 3: split (one ticket, two problems)

Customers don't always send one problem per email. "My login's broken and also I was double-charged last month" is one ticket with two topics — one for support, one for finance. Routing it to either team leaves the other half stranded.

Split turns the one ticket into two. The Function URL Lambda creates two new tickets from the original, each carrying the relevant part of the message, sends each back through the read step so it gets its own topic, urgency, and tone, and routes each on its own. The original ticket is closed with a note pointing at the two new ones, so the trail is clear. Split is the rarest of the three actions, but it's the right tool for the ticket that genuinely belongs in two places — far better than forcing it into one queue and trusting two teams to coordinate from a single card.

Every action is logged, every action teaches

The `tr-audit` table records every reassign, bump, and split with the user, the timestamp, and a snapshot of the ticket before and after. If a reassign was itself a mistake — someone moved it to the wrong team in a hurry — the next person can reassign again; the trail shows the full chain, so nothing is lost. The `tr-corrections` table is the narrower, teaching record: just the cases where the router's tag and a human's tag disagreed. That table is what the weekly digest counts to show the correction rate, and what feeds the examples.

This is the quiet payoff of keeping a human in the loop on every misroute instead of trusting the router blindly. Each correction is cheap — one tap — and each one makes the system a little more right. A router that learns from its mistakes beats one that's clever on day one and never improves.

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 one small model call per ticket is the only real cost.

PART 6 OF 7

MAY 30, 2026 PART 6 OF 7 · TICKET ROUTER SERIES ~3 MIN READ

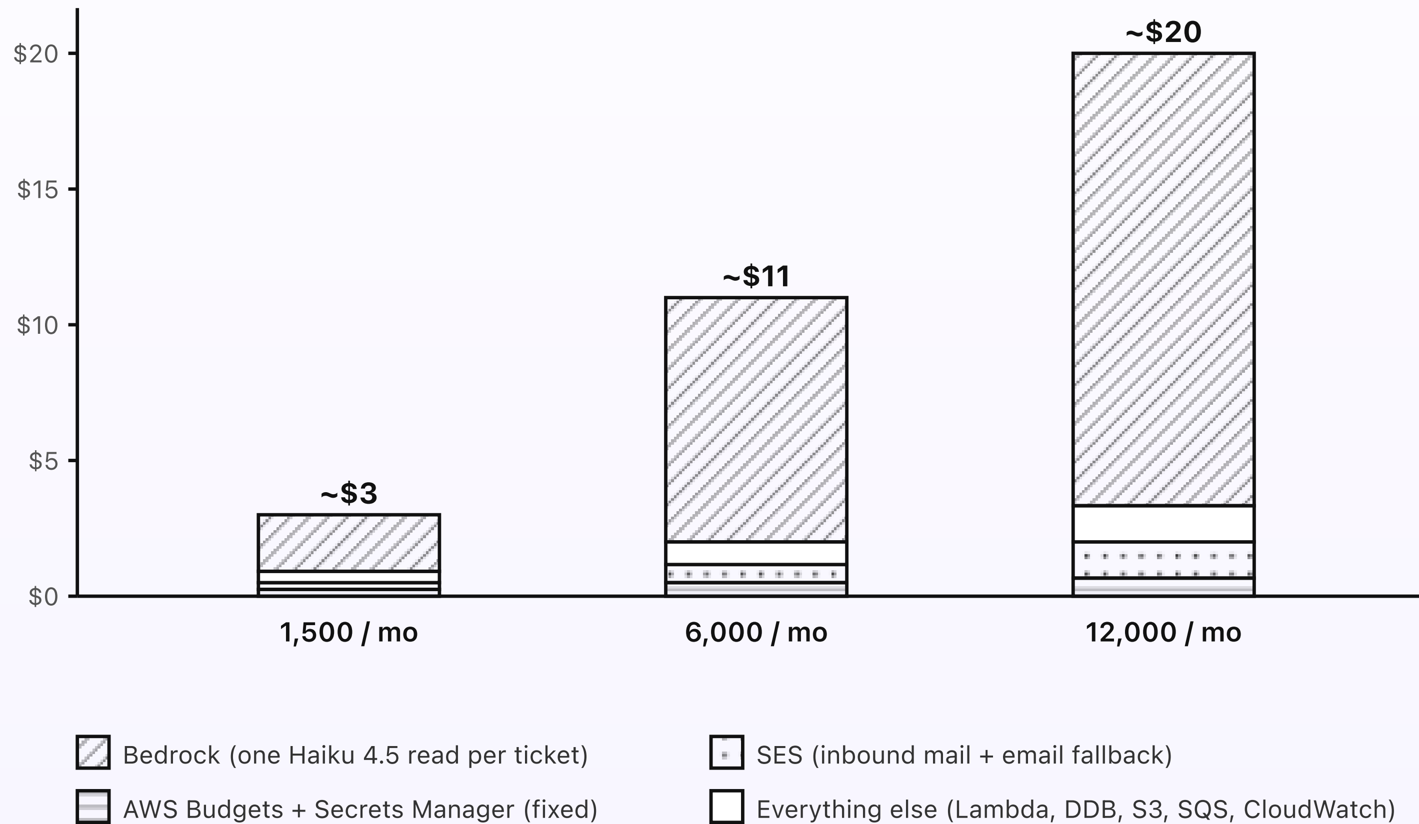
What the ticket router costs

The router is a cheap system, but it has one cost the watcher in the last series didn't: it reads every ticket with a model. That one small Bedrock Haiku 4.5 call per ticket is the only part of the bill that grows with volume. Everything else — the Lambdas, the queue, the rules lookup, the hand-off — is pennies. At typical SMB volume the bill is a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $3/month at typical SMB volume (around 1,500 tickets a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The dominant cost is one Haiku 4.5 read per ticket — a fraction of a cent each.
- The router decision, the queue, and the hand-off use no model — they're pennies.
- At 6,000 tickets a month the bill is around $11. At 12,000 it's around $20.

Cost at three volumes



The per-ticket read is the dominant cost — and even that is a fraction of a cent per ticket.

Fig 6. Monthly cost at three ticket volumes. Bedrock is the dominant slice because the router reads every ticket once; the fixed services and the everything-else bucket stay small. The cost tracks ticket count, not team size.

Where the dollars actually go

Bedrock (the bulk). One Haiku 4.5 call per ticket reads the topic, urgency, and tone. The input is the ticket body plus a short prompt and a few labelled examples — a few thousand tokens; the output is a tiny JSON object — a few dozen tokens. That's a fraction of a cent per ticket. At 1,500 tickets a month it's a couple of dollars; at 12,000 it's the bulk of a $20 bill. This is the one cost that grows directly with how many tickets you get, which is exactly the cost you want to be paying — it only happens when there's real work to sort.

Lambda runtime. The intake, the read trigger, the router, the hand-off, and the correction handler are all small Lambdas that run for a few hundred milliseconds each. Even at 12,000 tickets a month, the Lambda total lands under a dollar. None of them run when there are no tickets.

DynamoDB on-demand. Three small tables: `tr-routes`, `tr-corrections`, `tr-audit`. A handful of writes per ticket and a few reads. Pennies a month at any of these volumes.

SQS. The queue that absorbs bursts. The first million requests a month are free, and an SMB doesn't come close. Effectively free.

S3 + storage. The raw inbound mail and the mirrored rules. A few hundred KB at SMB volume. Effectively free.

SES. Inbound for the email lane: $0.10 per thousand received messages. Outbound for the email-fallback hand-offs: $0.10 per thousand sent. Both are a few cents at this scale.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the web-form and correct-routing endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Nothing runs between tickets.
- **A Knowledge Base.** The router reads each ticket fresh and looks teams up in a sheet — no vector search. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **A second model call.** The routing decision, the queue, and the hand-off are plain Python. Bedrock fires exactly once per ticket, on the read.

How the cost scales

The bill tracks ticket volume almost one-for-one, because the per-ticket read is the dominant cost and everything else is small and flat. So the bill at 24,000 tickets a month is around $40, and at 50,000 it's around $80. Past those volumes you'd look at whether a cheaper first pass — a quick keyword check that routes the obvious tickets without a model call, sending only the ambiguous ones to Haiku — is worth the added complexity. For an SMB it isn't; reading every ticket is simpler and still cheap.

Set an AWS Budgets alarm at $15/month for a typical SMB so anything unusual — a spam flood, a runaway retry — pages you before the bill matters. At higher steady volume, raise the ceiling to match. The router's normal bill stays well under whatever ceiling fits your volume.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, DynamoDB schemas, the SES rule set, and the SQS queue config.

PART 7 OF 7

MAY 30, 2026 PART 7 OF 7 · TICKET ROUTER SERIES ~8 MIN READ

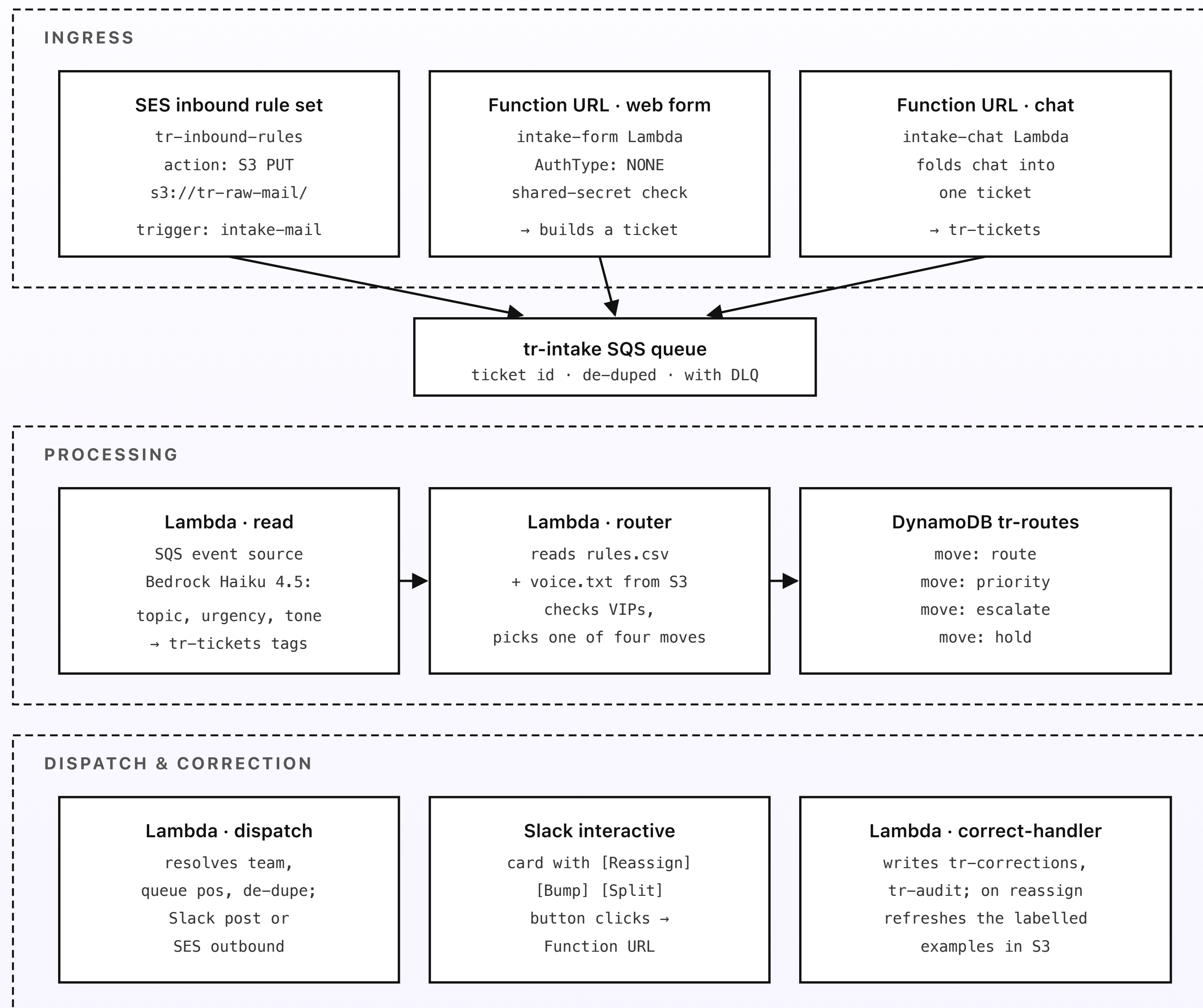
Engineering reference: the ticket router architecture

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

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound, Bedrock cross-Region inference, SQS, and Lambda Function URLs 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 ticket sitting in the wrong queue for an hour, not a regional outage. One AWS account dedicated to the router (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



The router only sorts and routes — and every correction is logged to tr-corrections.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes onto one queue), processing (the read call then the router picking a move), dispatch and correction (the ticket lands and a human's correction is recorded and fed back). Every Lambda is event-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.

- **`intake-mail`** — S3 PUT trigger on `s3://tr-raw-mail/`. Parses the MIME, extracts sender/subject/body, strips quoted reply history, matches thread headers against open tickets in `tr-tickets` to merge replies, writes a new ticket (or appends to an existing one), and sends the ticket id to the `tr-intake` SQS queue. Memory: 256 MB. Timeout: 30 s.
- **`intake-form`** — Lambda Function URL, `AuthType: NONE`, verifies a shared secret (in Secrets Manager under `tr/form/secret`) on the POST body. Builds a ticket from the form fields in the same shape as the mail lane, writes to `tr-tickets`, enqueues the id. Memory: 256 MB. Timeout: 15 s.
- **`intake-chat`** — Lambda Function URL, signature-verified against the chat tool's secret. Folds a finished conversation into one ticket body, writes to `tr-tickets`, enqueues the id. Memory: 256 MB. Timeout: 15 s.
- **`read`** — SQS event source on `tr-intake` (batch size 5, partial-batch responses enabled). For each ticket, calls Bedrock Haiku 4.5

(`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) with the ticket body, the topic list from `rules.csv`, and the labelled examples from `examples.jsonl`; parses the returned JSON (`topic`, `urgency`, `tone`); writes the tags back to `tr-tickets`. On a malformed model response, retries once with a stricter prompt, then tags `topic: unsure` so the router holds it. Memory: 512 MB. Timeout: 30 s. *This is the only Bedrock callsite.*

- **`router`** — invoked by `read` after tagging (or as a second SQS stage). Reads `s3://tr-rules-source/rules.csv` (topic-to-team map) and `voice.txt` (urgency words, VIP list, priority rules). Applies the decision flow from Part 3, picks one of `route`, `priority`, `escalate`, `hold`, and writes a row to `tr-routes`. *No Bedrock calls.* Memory: 256 MB. Timeout: 15 s.
- **`dispatch`** — triggered on new `tr-routes` rows (DynamoDB Streams). Resolves the team, sets the queue position, runs the duplicate check against recent open tickets, formats the hand-off card, and ships via Slack `chat.postMessage` (`tr/slack/bot-token` in Secrets Manager) or SES `SendRawEmail` to the team's shared inbox. Writes the dispatch outcome back to `tr-routes`. Memory: 256 MB. Timeout: 30 s.
- **`correct-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive button clicks (Reassign/Bump/Split) and by email-link clicks. Updates `tr-tickets` and `tr-routes`; writes to `tr-corrections` and `tr-audit`; on reassign or bump, refreshes `examples.jsonl` in `s3://tr-rules-source/` with the corrected label (capped to the most recent N examples per topic). On split, creates two new tickets and re-enqueues both. Memory: 256 MB. Timeout: 15 s.

- `drive-sync` — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Sheets API + Docs API (service-account credentials in Secrets Manager under `tr/drive/sa`) to export the rules sheet and the rules doc, writing `rules.csv` and `voice.txt` to `s3://tr-rules-source/` only if changed since the last sync. Memory: 256 MB. Timeout: 30 s.
- `digest` — EventBridge Scheduler target, weekly Sunday 6pm. Reads `tr-routes` and `tr-corrections` for the past week; posts a summary to a configured Slack channel: volume by topic and team, the correction rate, and the slowest queues. No Bedrock; a plain summary table. Memory: 256 MB.
- `summary` — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `tr-routes`, `tr-corrections`, and `tr-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph narrative (busiest topics, slowest queues, what the corrections taught); emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB** · `tr-tickets` — one row per ticket. PK `ticket_id`; attributes: `customer`, `source` (mail/form/chat), `subject`, `body`, `topic`, `urgency`, `tone`, `status`, `received_at`. GSI on `(customer, topic)` for the duplicate check. On-demand.
- **DynamoDB** · `tr-routes` — one row per routing decision. PK `ticket_id`; attributes: `topic`, `urgency`, `tone`, `team`, `move` (route/priority/escalate/hold), `queue_pos`, `dispatched_via`, `decided_at`. DynamoDB Streams enabled to trigger `dispatch`. On-demand.

- **DynamoDB · `tr-corrections`** — one row per human correction. PK `(ticket_id, ts)`; attributes: `action` (reassign/bump/split), `orig_topic`, `orig_team`, `new_topic`, `new_team`, `by_user`. This table feeds the labelled-example refresh and the correction-rate metric. On-demand.
- **DynamoDB · `tr-audit`** — one row per write action of any kind. PK `(ticket_id, ts)`; attributes: `action`, `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **S3 · `tr-raw-mail`** — raw inbound MIME from the email lane. Lifecycle to Glacier at 30 days; expiry at 1 year.
- **S3 · `tr-rules-source`** — mirrored `rules.csv`, `voice.txt`, and `examples.jsonl`. Versioning enabled so a bad rules edit or example flood can be rolled back in one click.

SQS

- **`tr-intake`** — standard queue between the three intake lanes and the `read` Lambda. Visibility timeout 60 s (6× the read function timeout). Absorbs bursts so a spike of tickets never overruns Bedrock's rate limit.
- **`tr-intake-dlq`** — dead-letter queue, `maxReceiveCount` 3. A ticket that fails the read three times (malformed body, model error) lands here and pages the on-call admin instead of silently disappearing. Redrive back to `tr-intake` once the cause is fixed.

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: `read` for the per-ticket topic/urgency/tone classification, and `summary` for the monthly narrative. Sonnet 4.6 is *not* used — classification is well within Haiku's reach, and a heavier model on the hot path would multiply the dominant cost for no gain.
- **Embeddings.** Not used. Routing is a fresh read plus a sheet lookup; deterministic mapping beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas cover SMB volume comfortably. SQS in front of `read` smooths bursts so the per-ticket calls stay under the on-demand throughput limit.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `support.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `tr-inbound-rules`: one rule with recipient `support@your-company.com` → spam scan → S3 PUT to `s3://tr-raw-mail/<message-id>` → stop. The S3 PUT triggers `intake-mail`.
- SES outbound for the email-fallback hand-offs and the monthly summary: verify a sender identity at `router@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **read role:** `sqs:ReceiveMessage` + `DeleteMessage` on `tr-intake`; `bedrock:InvokeModel` on the Haiku ARN; `s3:GetObject` on the rules and examples keys; `dynamodb:UpdateItem` on `tr-tickets`.
- **router role:** `s3:GetObject` on `rules.csv` and `voice.txt`; `dynamodb:GetItem` on `tr-tickets`; `dynamodb:PutItem` on `tr-routes`. *No* `bedrock:*`.
- **dispatch role:** `dynamodb:GetRecords` on the `tr-routes` stream; `dynamodb:Query` on the `tr-tickets` GSI for the duplicate check; `secretsmanager:GetSecretValue` on the Slack bot token; `ses:SendRawEmail` from the verified sender; outbound network to `slack.com`.
- **correct-handler role:** `dynamodb:UpdateItem` on `tr-tickets` and `tr-routes`; `dynamodb:PutItem` on `tr-corrections` and `tr-audit`; `s3:GetObject` + `PutObject` on `examples.jsonl`; `sqs:SendMessage` on `tr-intake` (for split). Verifies the Slack signing secret on every request.
- **intake-* roles:** `s3:GetObject` on `tr-raw-mail` (mail only); `dynamodb:PutItem` + `Query` on `tr-tickets`; `sqs:SendMessage` on `tr-intake`; the secret for the lane’s shared secret or signature.
- **drive-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on `tr-rules-source`; outbound network to `www.googleapis.com`.

Slack interactive flow

Hand-off cards are posted via the `chat.postMessage` Web API with Block Kit blocks containing the action buttons (Reassign, Bump, Split). Button clicks are sent by Slack to the configured Interactivity request URL, which is the `correct-handler` Function URL. `correct-handler` verifies the Slack signing secret on the inbound request, parses the `action_id` (`reassign` , `bump` , `split`), opens a menu or modal where needed (Reassign opens a team menu; Split opens a two-field modal; Bump is one-tap), and processes the response on submit.

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

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch metric for alerting.
- **Alarms:** `tr-intake-dlq` depth > 0 (a ticket failed to read three times); `read` Bedrock throttle rate > 1% in 24h; dispatch failure rate > 1% in 24h; `correct-handler` signature-verification failures > 5/hour (might mean the Slack secret rotated).
- **Custom metric:** correction rate — `tr-corrections` writes over `tr-routes` writes — tracked weekly. A rising correction rate on a topic means the examples for it need a refresh or the rules sheet needs a new row.
- **X-Ray:** off by default. Not worth the cost at SMB volume.

- **AWS Budgets:** $15/month threshold for a typical SMB, alarm at 80% and 100%, posts to SNS topic `tr-cost-alarm` subscribed to the on-call admin's email and Slack. Raise the ceiling to match higher steady volume.

Config and secrets

Service-account credentials for the Sheets and Docs APIs live in Secrets Manager under `tr/drive/sa`. Slack bot token and signing secret under `tr/slack/*`. The web-form and chat lane secrets under `tr/form/secret` and `tr/chat/secret`. SES sender identity lives in IAM and the verified-domain config. The topic list, the VIP list reference, the priority rules, and the admin fallback team all live in Parameter Store under `/tr/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) running AWS SAM. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for `tr-rules-source` so a bad rules edit or example flood can be rolled back in one click, and keep the DLQ alarm wired before going live so a read failure pages someone instead of vanishing. Total deployable surface: around ten Lambdas, four DDB tables, two S3 buckets, two SQS queues (one DLQ), one SES rule set, and one Budgets alarm.

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