7-PART SERIES · FREE COMPANION

Form intake router

A serverless router that catches every form your website posts — contact, quote request, booking, signup; checks the fields, filters obvious spam, files each submission safely, and sends it to the right team and the right tool. If a downstream tool is down it retries and never drops the lead, and the customer always gets a “we got it” reply. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Form intake router

PART 1 OF 7

JUNE 9, 2026 PART 1 OF 7 · FORM INTAKE ROUTER SERIES ~5 MIN READ

A form intake router on AWS for a few dollars a month

A small business website usually has more forms than anyone keeps track of. The contact form that emails a shared inbox nobody checks on weekends. The quote-request form wired to a plugin that broke after an update and silently stopped sending. The booking form that lands in someone's spam folder. The newsletter signup that goes nowhere. Each one is a way a customer is trying to reach you — and each one is a place a lead can quietly disappear. This post walks through the design of a small router that catches every submission, checks it, files it safely, sends it to the right place, and always tells the customer "we got it."

KEY TAKEAWAYS

- Every form on your site posts to one door, so no submission slips through an old plugin or a dead inbox.
- Every submission is saved and acknowledged *before* any slow downstream work, so the customer always gets a reply.
- A routing table decides where each one goes: a team email, a CRM list or sheet tab, and an auto-reply.
- Deliveries retry on a queue; if a tool stays down, the submission waits safely and is never dropped.
- Designed on AWS for about $2/month at typical small-business volume.

The whole system on one page

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

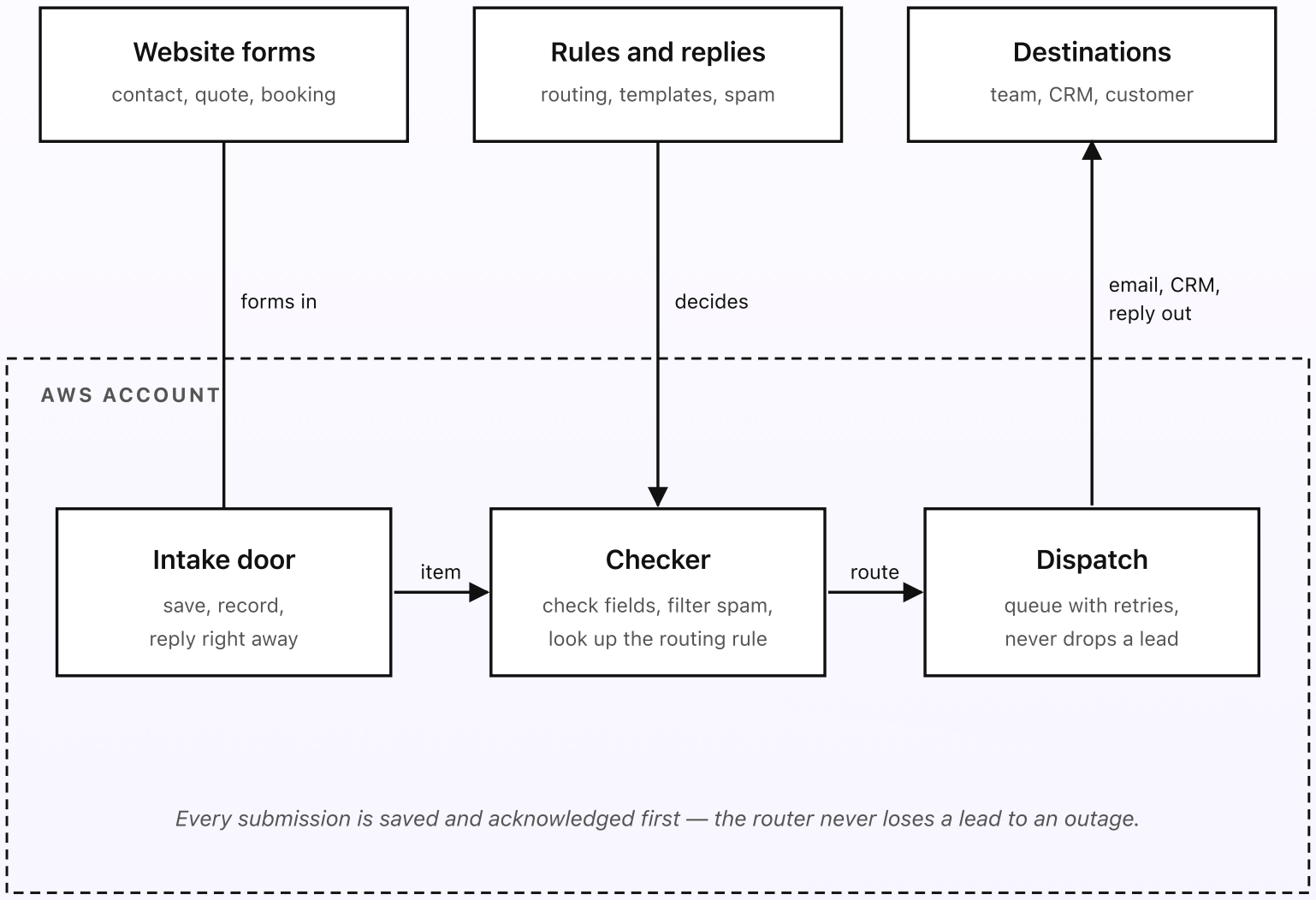

Website forms
contact, quote, booking
Rules and replies
routing, templates, spam
Destinations
team, CRM, customer
forms in
decides
email, CRM,
reply out
AWS ACCOUNT
Intake door
save, record,
reply right away
item
Checker
check fields, filter spam,
look up the routing rule
route
Dispatch
queue with retries,
never drops a lead
Every submission is saved and acknowledged first — the router never loses a lead to an outage.

Fig 1. Forms outside, three pieces inside AWS. Every form posts to one Intake door. The Checker validates and decides where it goes. Dispatch delivers each one from a retrying queue, so a busy downstream tool never costs you a lead.

What you set up once (the outside)

- **Website forms.** Every form on your site — the contact form, the quote-request form, the booking form, the newsletter signup — gets a tiny snippet that posts the submission to one address (a Lambda Function URL, which is just a plain web address AWS gives a small function). You don't replace your form builder or your site; you change where the form sends its data. Each form carries a short `form_id` so the router knows which form it came from. Part 2 covers the snippet in detail.
- **A rules folder.** Two short things in a Google Drive folder. The *routing table* is a Google Sheet with one row per form: which form, which team email should get it, which CRM list or sheet tab the row goes to, and which auto-reply template the customer gets. The *rules and replies* doc holds the actual auto-reply message templates (one per form) and the spam rules — the banned-word list, the rate limit, the honeypot field name. A non-technical owner can change where a form goes, or reword a reply, by editing the sheet. No deploy.
- **Destinations.** The places a checked submission ends up. The right team's email inbox (sales, support, bookings). Your CRM or a Google Sheet that your team already lives in. And the customer's own inbox, which gets a short "we got your message, here's what happens next" reply so they're never left wondering whether the form worked.

What runs on every submission (the inside)

- **The intake door.** The Function URL receives the form post. The very first thing it does is save the raw submission to S3 and write a record to DynamoDB — before any checking, before any sending. Then it replies to the browser so the customer's form shows "thanks, we got it" in well under a second. Saving first means that even if every later step failed, the submission still exists and can be replayed. Nothing a customer typed is ever lost to a crash. Part 2 walks through this.
- **The checker.** Reads the saved submission. Checks the required fields are present and look right (an email that looks like an email, a phone that looks like a phone). Runs the spam rules: a hidden honeypot field a real person never fills in, a minimum time-on-page, a rate limit per address, and a banned-pattern list. Anything that looks borderline gets one cheap Bedrock Haiku 4.5 second-opinion rather than a guess. Then it looks up the routing rule for that `form_id`. The checker is plain Python; the model is the exception, not the rule. Part 3 covers it.
- **Dispatch.** Takes the routing decision and does the deliveries — email the team, write to the CRM, add the row to the sheet — but it does each one as a separate job on an SQS queue (a queue is just a waiting line for jobs). If a delivery fails because a tool is briefly down, the queue retries it a few times with growing gaps. If it still won't go through, the job waits safely in a dead-letter queue (a holding line for jobs that keep failing) and an alert goes out, so a human can replay it once the tool is back. The customer reply and the team email are separate jobs, so a slow CRM never delays the "we got it." Parts 4 and 5 cover this.

In plain words

A prospect fills in your quote-request form at 11pm on a Sunday: name, email, “need 200 branded mugs by month-end, what’s the price?” The intake door saves it and replies in their browser instantly: “Thanks — we’ve got your request and someone from sales will reply within one business day.” They also get that same line by email a moment later. The checker confirms the email is real, sees it’s not spam, and reads the routing table: quote requests go to `sales@` and to the “Quotes” tab of your CRM. Dispatch emails the sales team the full submission and writes the CRM row. Your CRM happens to be mid-maintenance for ten minutes — the write fails, the queue waits, retries, and succeeds on the third try. Monday morning, sales sees the lead in both their inbox and the CRM, and the customer already feels looked after. Nobody had to notice anything went wrong.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the one quote request that vanished into a broken plugin, the booking that went to spam, or the contact form that a customer filled in twice because the first time it looked like nothing happened.

DESIGN RULES THAT SHAPED EVERY DECISION

- Save and acknowledge first. The submission exists and the customer is told before any downstream work runs.
- Never drop a lead. Every delivery retries; anything that keeps failing waits in a dead-letter queue, never deleted.
- Favor the real lead over the blocked one. Spam-flagged submissions are held for review, not silently thrown away.
- One door, one routing table. Adding a form or changing where it goes is a sheet edit, not a deploy.
- Plain Python on the hot path. A model is called only when a check is genuinely borderline.
- Every submission is logged. You can answer "what happened to that lead?" for any submission, any day.

Why this shape

Most small businesses wire forms one of three ways: a form-builder plugin that emails a shared inbox, a third-party form service that charges per submission and locks your leads in their dashboard, or a custom script that worked the day it was written. All three share the same weakness — when the delivery step fails, the customer's submission is just gone, and usually nobody finds out until a prospect calls asking why they never heard back. The plugin update that broke the email. The service that hit its free-tier cap. The script that timed out because the CRM was slow.

The setup above fixes the weakness by separating “we received it” from “we delivered it.” The moment a form posts, the submission is saved and the customer is told — that part can’t fail quietly because it happens first and depends on nothing else. Delivery to your team and your tools happens afterward, on a queue that retries and never deletes. The router is invisible most days; you only notice it on the day a tool would otherwise have eaten a lead, and on that day it simply waits and tries again.

The next four posts walk through each piece in turn: how a form submission gets captured, how it gets checked, how it finds the right tool, and how it gets confirmed and never dropped. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 9, 2026 PART 2 OF 7 · FORM INTAKE ROUTER SERIES ~4 MIN READ

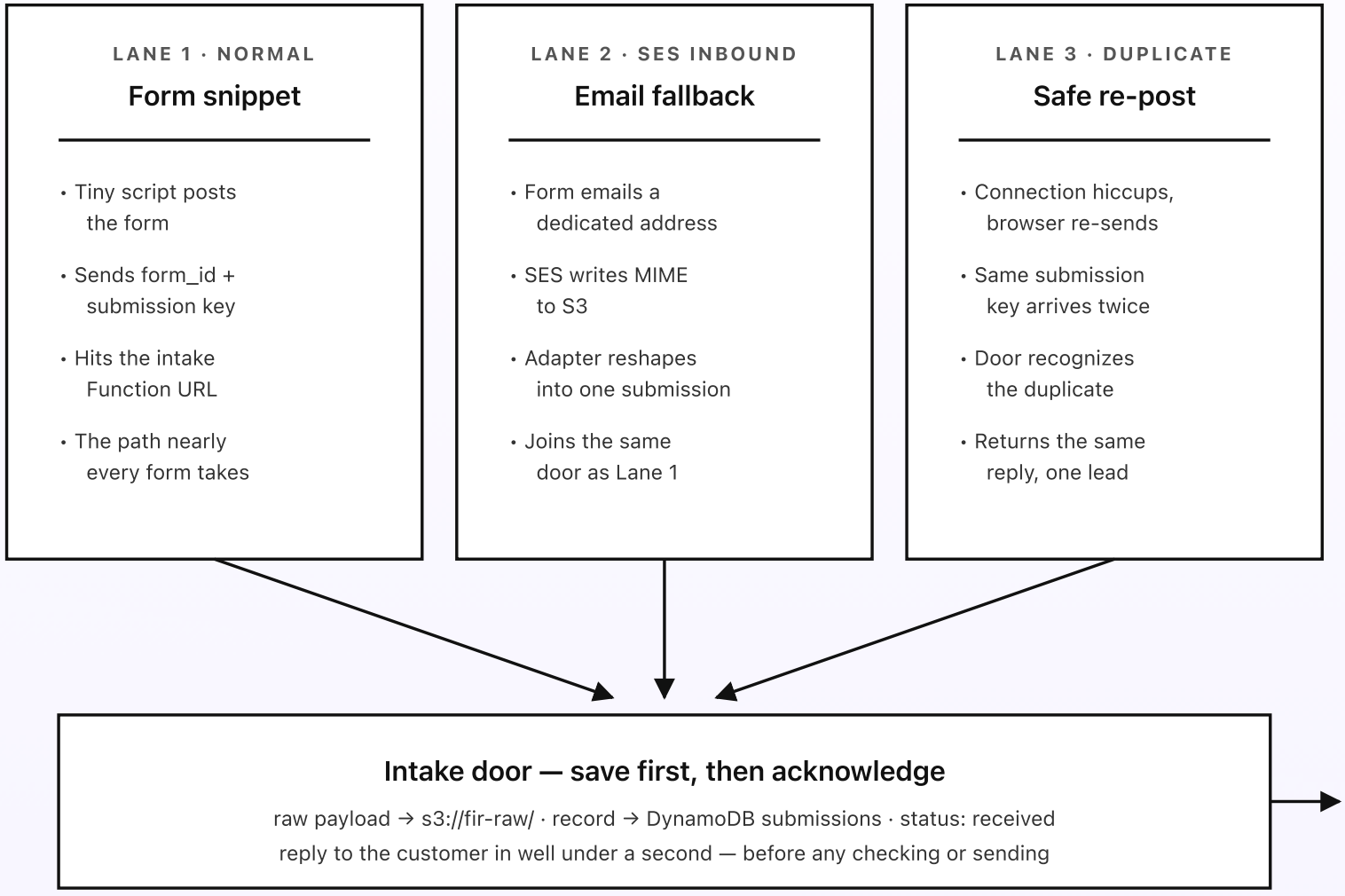
How a form submission gets captured

The router can only route what it actually receives. So the first job is making sure a submission never gets lost between the customer clicking “Send” and the router writing it down. There are three ways a submission reaches the door: the normal way (a tiny snippet posts the form), a fallback (an email lane for forms you can’t add a snippet to), and a re-post lane (the same submission arriving twice because the customer’s connection hiccuped). All three end the same way — saved, recorded, and acknowledged — before anything slow happens.

KEY TAKEAWAYS

- Three capture lanes feed one saved record: the form snippet, an email fallback, and a safe re-post.
- The door saves the raw payload to S3 and writes a record before checking or sending anything.
- The customer gets a “we got it” reply in well under a second, before any downstream work.
- A submission key from the form means a double-click never creates two leads.
- The saved record is the one source of truth; everything later reads from it.

Three lanes into one saved record



The door saves and acknowledges first — capture can't fail quietly because it happens before anything slow.

Fig 2. Three capture lanes converge on one intake door. The door saves the raw payload and writes a record before it acknowledges, and acknowledges before it checks or sends. The submission key keeps a double-click from becoming two leads.

Lane 1: the form snippet (how nearly everything arrives)

Each form on your site gets a tiny snippet — a few lines of JavaScript — that sends the form's fields to one address when the customer clicks Send. That address is a Lambda Function URL: a plain web address AWS gives a small function, with no API Gateway in front of it. The snippet adds two things to the submission: a short `form_id` so the router knows which form it is (`contact`, `quote`, `booking`), and a one-time `submission_key` it generates fresh each time the form is opened. You keep your existing form builder, your existing fields, your existing design; you only change where Send points.

When the post lands, the door does the smallest possible amount of work: it writes the raw body to `s3://fir-raw/`, writes a record to the DynamoDB `submissions` table with status `received`, and returns a small "thanks, we got it" response to the browser. That's it. No spam check, no routing, no email yet — those all happen after the customer has already been told their message is safe. The whole round trip is well under a second because it touches only two fast services.

Lane 2: email fallback (for forms you can't touch)

Sometimes you can't add a snippet — a locked-down site builder, a third-party landing page, an old form you don't want to risk editing. For those, the form is pointed at a dedicated email address instead (most form builders can email a

submission). Amazon SES receives that email, writes the raw message to S3, and a small adapter Lambda turns the email's fields into exactly the same submission shape Lane 1 produces — same `form_id`, a `submission_key` derived from the message id. From the door's point of view there's no difference; an email-lane submission is saved, recorded, and (where an address is present) acknowledged just like a posted one.

This lane is the safety net that means "every form on your site" is actually true, not just "every form we could rewire."

Lane 3: safe re-post (the double-click problem)

Real browsers on real connections re-send. The customer taps Send, the spinner stalls on flaky hotel wifi, they tap again. Without care, that's two leads, two team emails, and a prospect who looks like they asked twice. The `submission_key` generated when the form opened fixes this: the door writes the record using that key as the identifier, with a conditional write that only succeeds the first time. A second arrival with the same key finds the record already there, skips creating anything new, and returns the same "we got it" the first one did. One submission, one lead, one reply — no matter how many times the button is pressed.

This is the plumbing word *idempotency*: doing the same thing twice has the same effect as doing it once. It's a small detail that quietly removes a whole class of "why did this customer come through three times?" confusion.

Why the door saves before it does anything else

The order is the whole point. If the door checked spam first, or tried to email the team first, then a crash or a slow tool during that work would happen *before* the submission was written down — and the lead would be gone with no trace. By saving the raw payload and the record first, the worst case becomes "we have the submission but haven't delivered it yet," which is fully recoverable: a later step (or a human) can replay it from the saved record. The customer is acknowledged off that same saved record, so even a total failure of everything downstream still leaves the prospect feeling heard and the lead sitting safely in storage.

Next post: how the checker reads that saved submission, confirms the fields, filters spam without throwing away real leads, and decides which routing rule applies.

PART 3 OF 7

JUNE 9, 2026 PART 3 OF 7 · FORM INTAKE ROUTER SERIES ~5 MIN READ

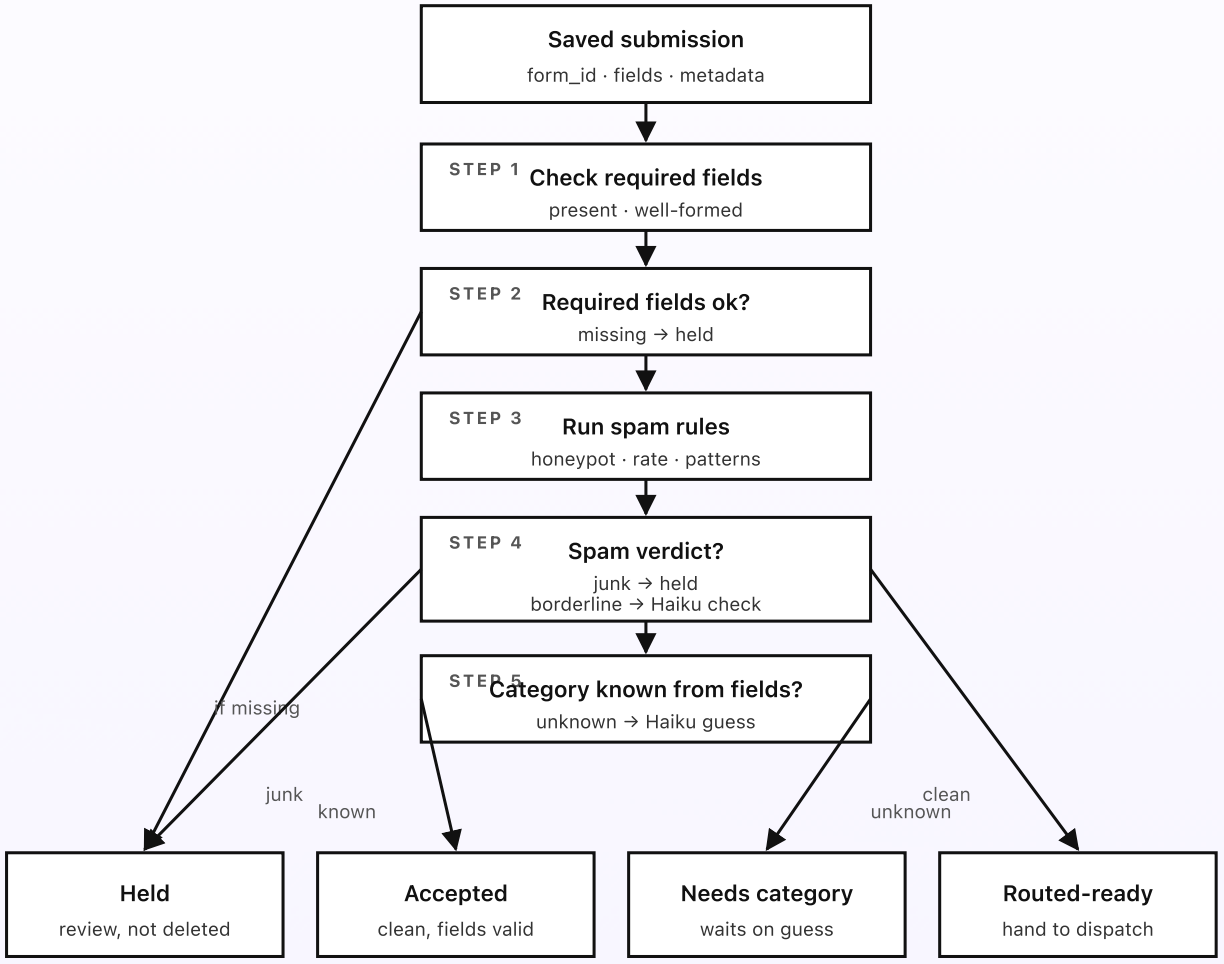
How a form submission gets checked

The submission is saved and the customer already has their reply. Now the checker picks it up and decides three things: are the fields actually usable, is this a real person or spam, and what kind of request is it. The whole decision is plain Python with one small exception. No model on required fields. No model on the obvious-spam rules. A model is called only when a check is genuinely borderline — and even then, just for a second opinion, never to throw a lead away on its own.

KEY TAKEAWAYS

- The checker runs right after capture, reading the saved submission — the customer is already acknowledged.
- Spam rules live in the rules doc — honeypot field, minimum time-on-page, per-address rate limit, banned patterns.
- Four outcomes per submission: held, accepted, needs-category, or routed-ready.
- Anything flagged is filed in a held bucket for one-tap review — never silently deleted.
- Bedrock fires only on borderline spam and unknown categories. Most submissions never touch a model.

The decision flow, per submission



The rules doc holds every spam rule — and a real lead is never deleted, only held for a human.

Fig 3. The checker's decision tree, per submission. Five steps decide which of four outcomes applies. The rules doc holds every spam rule and category map; the checker only enforces them, and held leads are kept for review.

Required fields: usable beats present

The first check is the dull, important one: are the fields the form marked as required actually there and actually usable? A required email that's blank, or that reads `asdf`, or a phone number that's three digits — these aren't leads you can act on. Rather than guess or auto-correct, the checker routes them to *held* with a note saying which field failed. A human glances at the held bucket, and most of the time it's genuinely junk; occasionally it's a real person who fat-fingered their email, and the held note tells the team exactly what to ask for.

Which fields are required, and what "well-formed" means for each, lives in the rules doc per `form_id` — so adding a required field to the booking form is a doc edit, not a code change.

Spam rules: cheap, layered, and biased toward keeping leads

The spam check runs as a stack of cheap deterministic rules, in order, all configured in the rules doc:

- **Honeypot.** A hidden field a human never sees and never fills in, but bots happily complete. If it's filled, the submission is almost certainly a bot — held.

- **Time-on-page.** The snippet records how long the form was open. A submission completed in under a second or two is a script, not a person — held.
- **Rate limit.** More than a handful of submissions from the same address in a short window is a flood — the extras are held.
- **Banned patterns.** A short list in the rules doc — known spam phrases, link-stuffing, gibberish markers. A clear hit is held.

A submission that trips none of these passes as clean. A submission that clearly trips one is held. The interesting case is the in-between — a real-looking message that happens to mention a few risky words, or a slightly fast fill. For those, and only those, the checker makes one cheap Bedrock Haiku 4.5 call: "Here's a website form submission. Is this a genuine inquiry or spam? Answer with a label and one line of reasoning." The model's answer is a second opinion that tips a borderline case, never the sole judge. And crucially, the worst outcome of the whole stack is *held*, not deleted — a wrongly-flagged real lead sits one tap away from being released, while a wrongly-passed spam costs the team ten seconds to ignore. The bias is deliberate: losing a real customer is far worse than seeing a junk one.

The category decision

Most forms answer "what kind of request is this?" by themselves. A `quote` form is a quote; a `booking` form is a booking. The routing table maps the `form_id` straight to a category and the checker is done. The exception is the generic catch-all contact form, where the same form might carry a sales question, a

support problem, or a partnership pitch — and where it should go depends on what the customer actually wrote.

For those, the checker makes one Bedrock Haiku 4.5 call to read the message and propose a category from the short list in the rules doc (`sales`, `support`, `billing`, `other`). The submission sits at *needs-category* for the fraction of a second the call takes, then moves to *routed-ready* with the proposed category attached. If the model is unsure, it's allowed to answer `other`, which routes to a general inbox a human triages — an honest "I'm not sure" beats a confident wrong guess.

Status that makes the path auditable

Every step updates the submission's `status` in DynamoDB: `received` → `checking` → one of `held`, `routed-ready`. Each held submission carries a short `held_reason` (which field, which spam rule). This is what makes "what happened to that lead?" answerable: you can look up any submission and see exactly why it was held, or that it sailed through clean, with timestamps. No black box.

Why the hot path is mostly model-free

The checker could send every submission to a model and ask "is this good, and where should it go?" It doesn't, for two reasons. First, the common case — a valid, clean, clearly-categorized submission — should be utterly predictable and instant; a model in that loop adds variance and latency for no gain. Second, model calls cost money, and the overwhelming majority of submissions are unambiguous, so the call would be wasted on most of them. Bedrock earns its place exactly twice:

tipping a borderline spam call, and reading a generic contact message to guess a category. Everywhere else, plain Python reading a doc is faster, cheaper, and easier to reason about.

Next post: how a routed-ready submission finds the right tool — the routing table, the per-delivery queue, and the retries that mean a busy CRM never costs you a lead.

PART 4 OF 7

JUNE 9, 2026 PART 4 OF 7 · FORM INTAKE ROUTER SERIES ~5 MIN READ

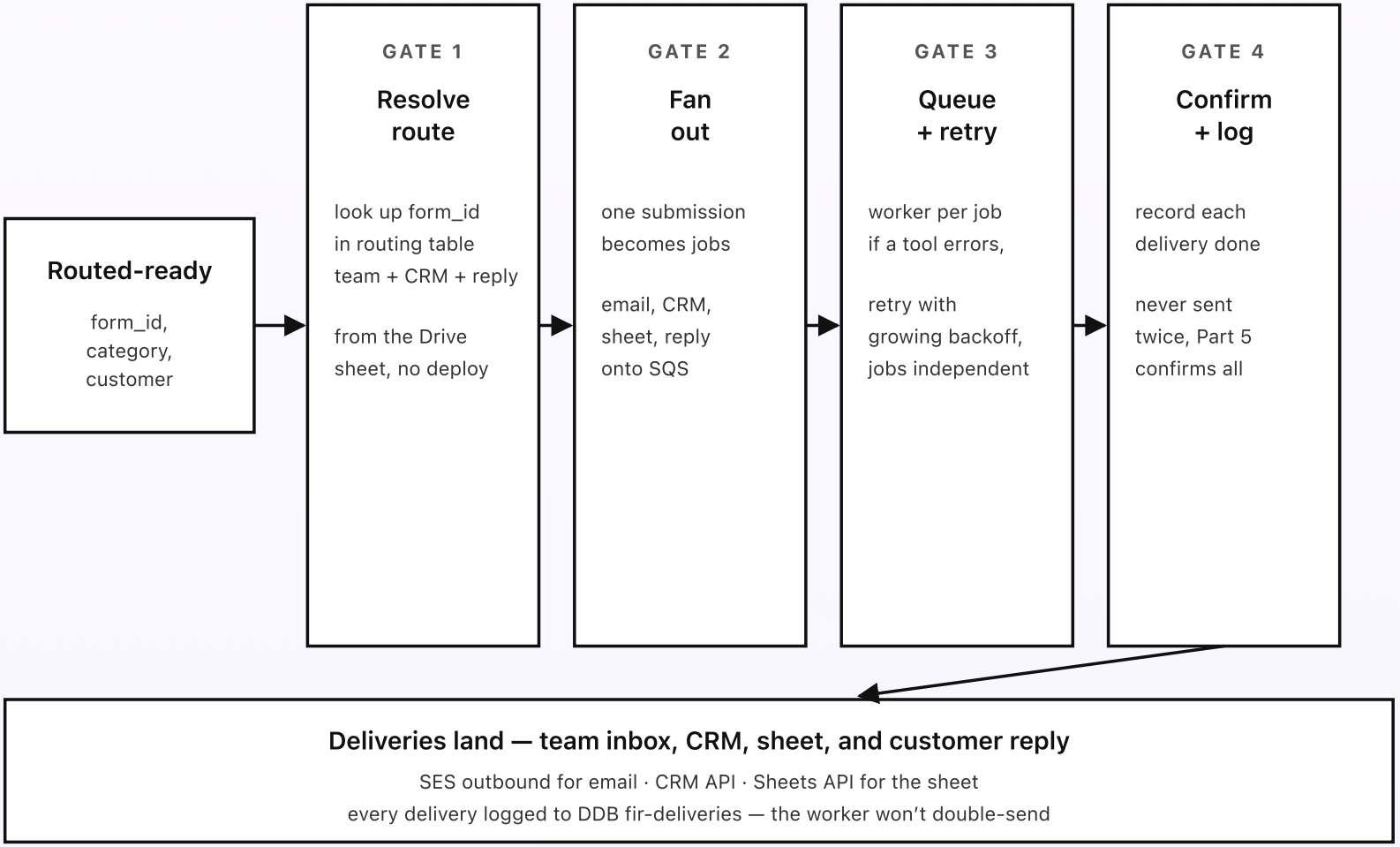
How a form submission finds the right tool

The submission is checked and routed-ready. Now dispatch has to get it to the right places — the team that handles it, the CRM or sheet it belongs in, the customer who's waiting on a reply. Get this wrong and the lead is worse than lost: it goes to the inbox nobody reads, or it vanishes because the CRM was down for ninety seconds. Four small gates sit between "routed-ready" and every delivery actually landing — and each delivery runs on its own so a slow tool can't hold up the rest.

KEY TAKEAWAYS

- The routing table maps each form to a team email, a CRM list or sheet tab, and an auto-reply template.
- Dispatch fans the submission out into separate delivery jobs — team email, CRM write, sheet write, customer reply.
- Each delivery is its own job on an SQS queue with retries and growing backoff.
- A slow CRM never delays the customer reply — the jobs are independent.
- Every delivery is logged so the next step knows it landed and won't double-send.

Four gates between routed-ready and delivered



Each delivery is independent and retried — one tool being down never costs a lead or blocks the others.

Fig 4. Four gates between routed-ready and delivered. Resolve the route. Fan out into separate jobs. Queue each one with retries. Confirm and log. Then the deliveries land — team, CRM, sheet, and customer — without any one slow tool holding up the rest.

Gate 1: resolve the route

Dispatch starts by looking up the routing rule for the submission's `form_id` (and category, for the catch-all contact form) in the routing table. The table is a Google Sheet your team can edit, mirrored to S3 every fifteen minutes by a small `sheet-sync` Lambda — so dispatch reads a fast local copy, never the live sheet, on every submission. Each row says four things: which **team email** gets it, which **CRM list or sheet tab** the row goes into, which **auto-reply template** the customer receives, and an optional **priority flag** for forms that should also page someone.

Because the table is a sheet, the day sales splits into "new business" and "renewals," or you add a careers form, the owner edits a row and the next submission routes the new way. No code change, no deploy, no waiting on an engineer.

Gate 2: fan out into independent jobs

This is the gate that protects you from slow tools. Instead of doing all the deliveries one after another in a single function — where a slow CRM would delay the team email and the customer reply behind it — dispatch turns the one submission into several separate jobs and drops each onto an SQS queue. A queue is just a managed waiting line for jobs. There's a job to email the team, a job

to write the CRM, a job to append the sheet row, and a job to send the customer's auto-reply. Each is picked up and run on its own.

Independence is the whole benefit. The customer's "we got it" reply doesn't wait behind the CRM. The team email doesn't wait behind the sheet. If three deliveries succeed instantly and one is slow, three land instantly and the fourth catches up on its own clock.

Gate 3: queue and retry

Each job is processed by a small worker Lambda that does exactly one delivery. If the downstream tool answers cleanly, the job is done. If it returns an error or times out — the CRM is mid-deploy, the email service hiccupped, the sheet API is briefly rate-limited — the job isn't lost. SQS hands it back after a wait and the worker tries again, with the gap growing each time (a few seconds, then tens of seconds, then minutes). Most transient failures clear within a try or two, and the delivery lands without anyone noticing there was a problem.

This is the heart of "never drops the lead." A normal form-handler that emails inline has exactly one chance: if the send fails, the lead is gone. A queued job has many chances spread over time, and the submission sits safely in storage the whole while. What happens if a tool stays down past every retry is the subject of Part 5 — the dead-letter queue — but the short version is that even then, nothing is deleted.

Gate 4: confirm and log

When a delivery succeeds, the worker writes a row to the `fir-deliveries` table in DynamoDB recording which submission, which delivery (`team_email`, `crm`, `sheet`, `customer_reply`), and when. Two things fall out of this. First, the system can prove a lead was delivered — the audit answer to "did sales actually get this?" is a lookup, not a shrug. Second, because each job checks the log before acting, a job that gets retried after it had actually already succeeded (which queues occasionally do) won't send the team a second copy or write the CRM row twice. The delivery is recorded as done, so the retry is a no-op.

Why the gates exist

None of these gates are exotic. They're the care a thoughtful person would take by hand — look up where this should go, do each errand separately so a slow one doesn't hold up the rest, try again if a delivery bounces, and write down what you've done so you don't do it twice. Putting them in code as four small steps makes them part of the design, not a feature you're trusting a fragile inline send to get right under pressure.

Next post: how a submission gets confirmed end to end — the customer reply, the de-duplicate guard, the retry budget, and the dead-letter queue that catches anything a downstream tool simply refuses to accept.

PART 5 OF 7

JUNE 9, 2026 PART 5 OF 7 · FORM INTAKE ROUTER SERIES ~5 MIN READ

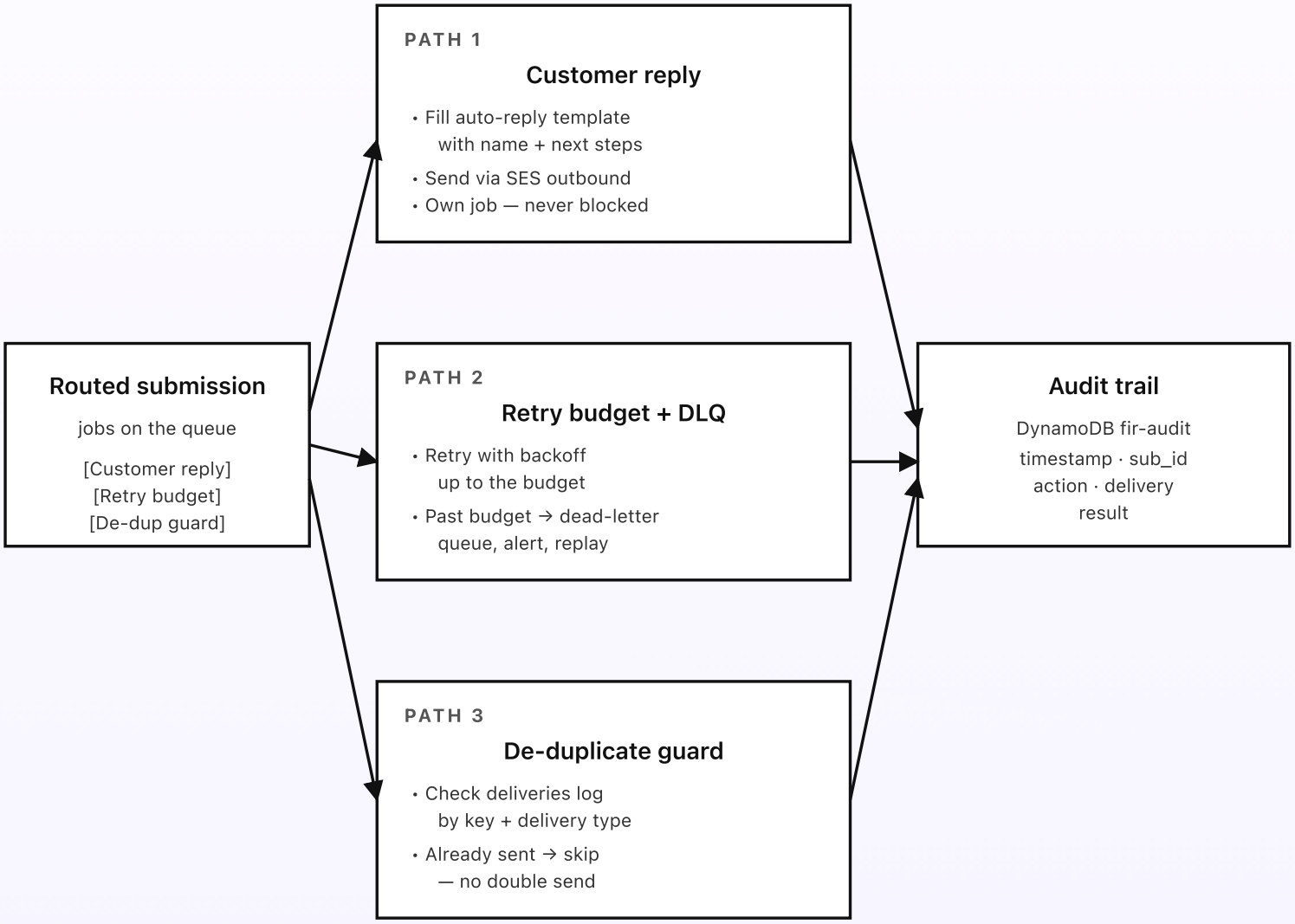
How a form submission gets confirmed

A quote request lands at 11:02pm. The customer's browser shows "thanks, we got it" instantly. But under the hood, "confirmed" means more than one thing: the customer got a real reply, the team and the tools got the lead, nothing was sent twice, and if a tool was down, the submission is waiting safely rather than lost. This post walks through the three things that finish a submission — the customer reply, the retry budget and dead-letter queue, and the de-duplicate guard — and how the record, the deliveries, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Three finishing pieces: the *customer reply*, the *retry budget* with a dead-letter queue, and the *de-duplicate guard*.
- The customer reply is its own queued job, so it lands even if every other delivery is slow.
- A delivery that keeps failing lands in a dead-letter queue — held, alerted, replayable, never deleted.
- The de-duplicate guard means a retried or double-clicked submission never sends twice.
- Every finish writes to the audit trail, so any lead's full story is one lookup away.

Three things that finish a submission



The customer is acknowledged first and nothing is deleted — a down tool only delays a delivery.

Fig 5. Three things finish a submission. The customer reply lands on its own job. The retry budget and dead-letter queue catch anything a tool refuses. The de-duplicate guard stops double sends. Every finish writes to the audit trail.

Path 1: the customer reply (the part they actually see)

The customer already got an instant “thanks, we got it” in their browser the moment they hit Send — that came straight from the intake door in Part 2. Path 1 is the proper follow-up: a real email to their inbox, filled from the auto-reply template the routing table chose for that form. The template lives in the rules doc, so the owner can word it per form — a quote-request reply might say “we’ll send pricing within one business day,” a booking reply might say “we’ll confirm your slot shortly.”

It matters that this is its own queued job. The customer’s reassurance doesn’t wait behind the CRM write or the team email. If those are slow, the customer still hears back on time. The one thing the router will never do is leave a customer wondering whether their message went anywhere — that silence is exactly what makes people fill in a form three times or give up and call a competitor.

Path 2: the retry budget and the dead-letter queue

Part 4 covered the happy retries — a tool hiccups, the job tries again, it lands. Path 2 is what happens when a tool doesn’t come back in time. Each delivery job has a *retry budget*: a set number of attempts over a stretch of minutes. Most outages are shorter than that and the job succeeds inside its budget. But if the CRM is

down for an hour, or someone rotated an API key and nobody updated the secret, the job will exhaust its budget without landing.

When that happens, the job isn't dropped — it's moved to a *dead-letter queue*, a separate holding line for jobs that kept failing. Three things follow. An alert fires (to the admin's email and Slack) so a human knows a delivery type is failing. The submission record is marked so you can see which leads are waiting. And once the tool is fixed, an operator replays the dead-letter queue and every held delivery goes through. The lead was never lost; it was parked safely until the world was ready for it. This is the concrete meaning of "never drops the lead" — not "nothing ever fails," but "a failure is held and recoverable, not deleted."

Path 3: the de-duplicate guard

Queues occasionally hand the same job out twice — it's a normal property of systems that promise to never lose a job, and the safe trade is "at least once" rather than "exactly once." Combined with the double-clicking customer from Part 2, that means the router has to assume any delivery might be attempted more than once. The de-duplicate guard handles it: before any send, the worker checks the `fir_deliveries` log for a row matching this submission key *and* this delivery type. If it's already there, the work was already done — the worker records the duplicate and stops. The team never gets two copies of the same lead; the CRM never gets two rows; the customer never gets two replies.

This is the same idempotency idea from Part 2, applied to the sending side. "Do it again" safely means "do nothing if it's already done."

Every finish is logged, every finish is recoverable

The `fir-audit` table records every finishing action — reply sent, delivery succeeded, delivery dead-lettered, duplicate skipped — with the submission id, the delivery type, the timestamp, and the result. Combined with the `received` / `checking` / `routed-ready` status from earlier, this gives any single lead a complete story: when it arrived, why it was held or passed, where it was routed, which deliveries landed and when, and whether anything had to wait in the dead-letter queue. When a prospect calls asking “did you get my message?” the answer is a lookup, not a guess.

And because the raw submission is still in S3 and the record is still in DynamoDB, anything can be replayed. A delivery that dead-lettered, a routing rule that was wrong and got fixed, a whole batch from an hour the CRM was down — all of it can be re-driven from the saved record. The system’s memory is the only memory anyone needs.

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 queue-and-retry design costs almost nothing extra.

PART 6 OF 7

JUNE 9, 2026 PART 6 OF 7 · FORM INTAKE ROUTER SERIES ~3 MIN READ

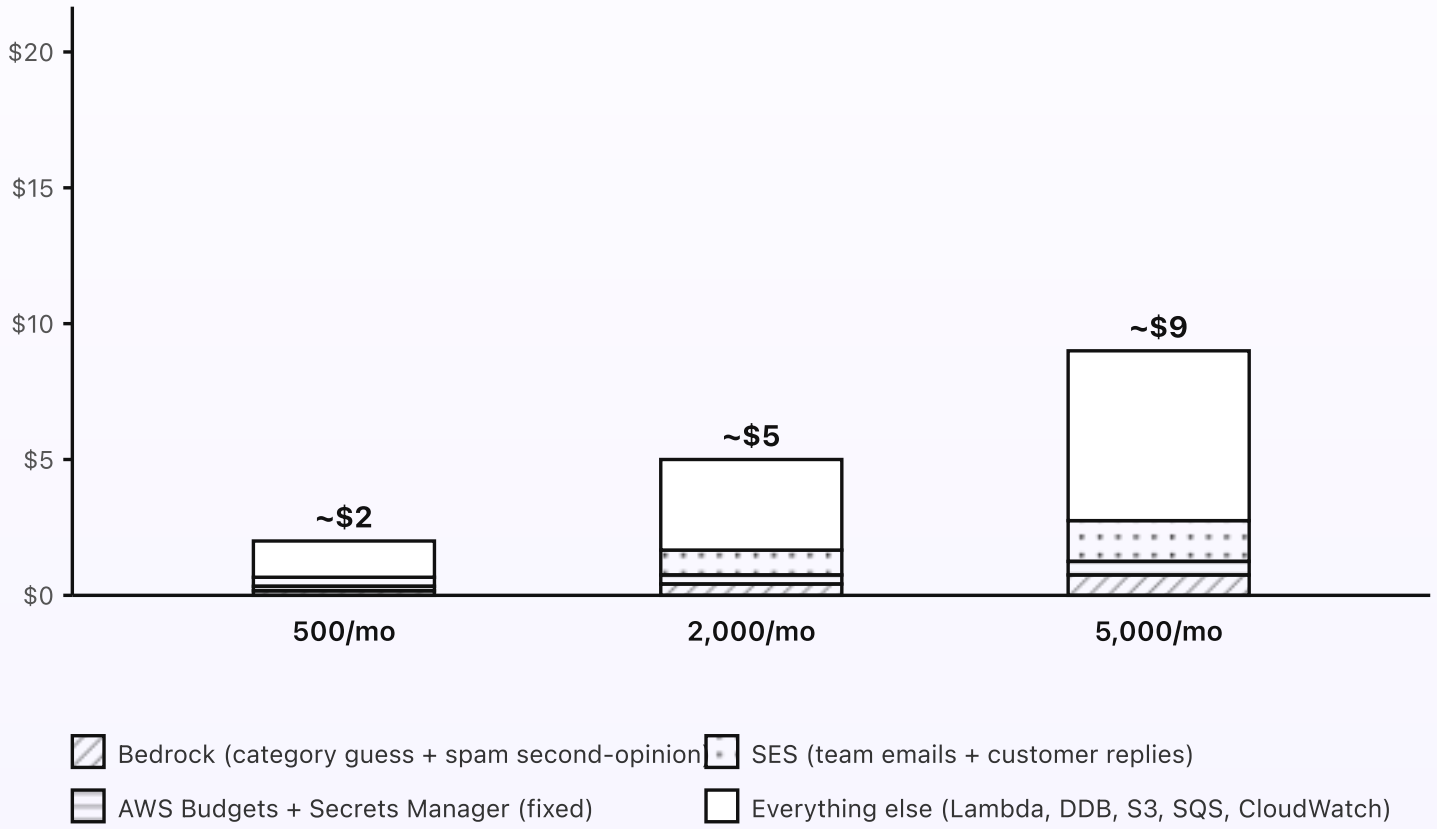
What the form intake router costs

The router is one of the cheapest systems in this whole series. Each submission saves a small file, writes a couple of DynamoDB rows, drops a few jobs on a queue, and sends a couple of emails. The hot path calls no model. Bedrock fires only on the few submissions that need a category guess or a borderline-spam second-opinion. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2/month at typical SMB volume (around 500 submissions a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The hot path costs pennies — no model calls on a normal submission.
- Bedrock fires only on category guesses and borderline-spam second-opinions — a minority of submissions.
- At 2,000 submissions the bill is around $5. At 5,000 it's around $9.

Cost at three volumes



The hot path is the dominant cost — and even that is a fraction of a cent per submission.

Fig 6. Monthly cost at three submission volumes. Bedrock stays a small sliver because it only fires on category guesses and borderline-spam checks. The dominant cost is the everything-else bucket: the hot path running on every submission.

Where the dollars actually go

Lambda runtime (the bulk). Every submission runs the intake door, the checker, and a handful of small delivery workers. Each is a few hundred milliseconds of arm64 Lambda time. At 500 submissions a month that's pennies; at 5,000 it's still well under a couple of dollars. Add the `sheet-sync` Lambda mirroring the routing sheet every fifteen minutes — a tiny fixed amount — and the Lambda total lands under a few dollars at all three volumes.

SES. The most visible variable cost, because every submission sends two emails: the team notification and the customer reply. SES outbound is $0.10 per thousand sent, so 500 submissions (about 1,000 emails) is around a dime a month; 5,000 submissions is around a dollar. Inbound for the email-fallback lane is the same $0.10 per thousand received and negligible at this scale.

DynamoDB on-demand. Four small tables: `submissions`, `deliveries`, `audit`, and a tiny `rate-limit` table. A few reads and writes per submission. Pennies a month at any of these volumes.

SQS. A few delivery jobs per submission plus the occasional retry. The first million requests a month are free; an SMB never leaves the free tier. Effectively zero.

S3 + storage. The raw payload per submission plus the mirrored routing sheet. A few hundred KB to a few MB total at SMB volume. Effectively free.

Bedrock (only when something fires it). The hot path uses no Bedrock. The category guess fires only on generic contact-form submissions whose category isn't fixed by the form; the spam second-opinion fires only on borderline submissions. Each is a Haiku 4.5 call with a few hundred input tokens and a tiny

output, a fraction of a cent. Even if a third of your submissions trigger one, it's a small sliver of the bill at 5,000 a month.

What doesn't cost money

- **API Gateway.** Replaced by a Lambda Function URL for the intake door.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Nothing runs between submissions.
- **A managed form service.** No per-submission SaaS fee, and your leads stay in your own AWS account.
- **Models on the hot path.** Required fields, spam rules, and routing are plain Python. Bedrock fires only on the two exceptions.

How the cost scales

Lambda runtime, DynamoDB, and SES all grow roughly linearly with submission count, because every submission runs the hot path and sends its emails. Bedrock grows with the *minority* of submissions that need a guess or a second-opinion, so it stays a small fraction even as volume climbs. The bill at 10,000 submissions a month is around $16; at 25,000 it's around $35. Past those volumes you'd look at batching the audit writes and trimming the emails (a daily digest instead of one-per-lead for low-priority forms), but those are tuning steps, not redesigns.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The router's normal-volume bill stays comfortably under that ceiling.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, DynamoDB schemas, the SQS and DLQ config, and the Function URL setup.

PART 7 OF 7

JUNE 9, 2026 PART 7 OF 7 · FORM INTAKE ROUTER SERIES ~8 MIN READ

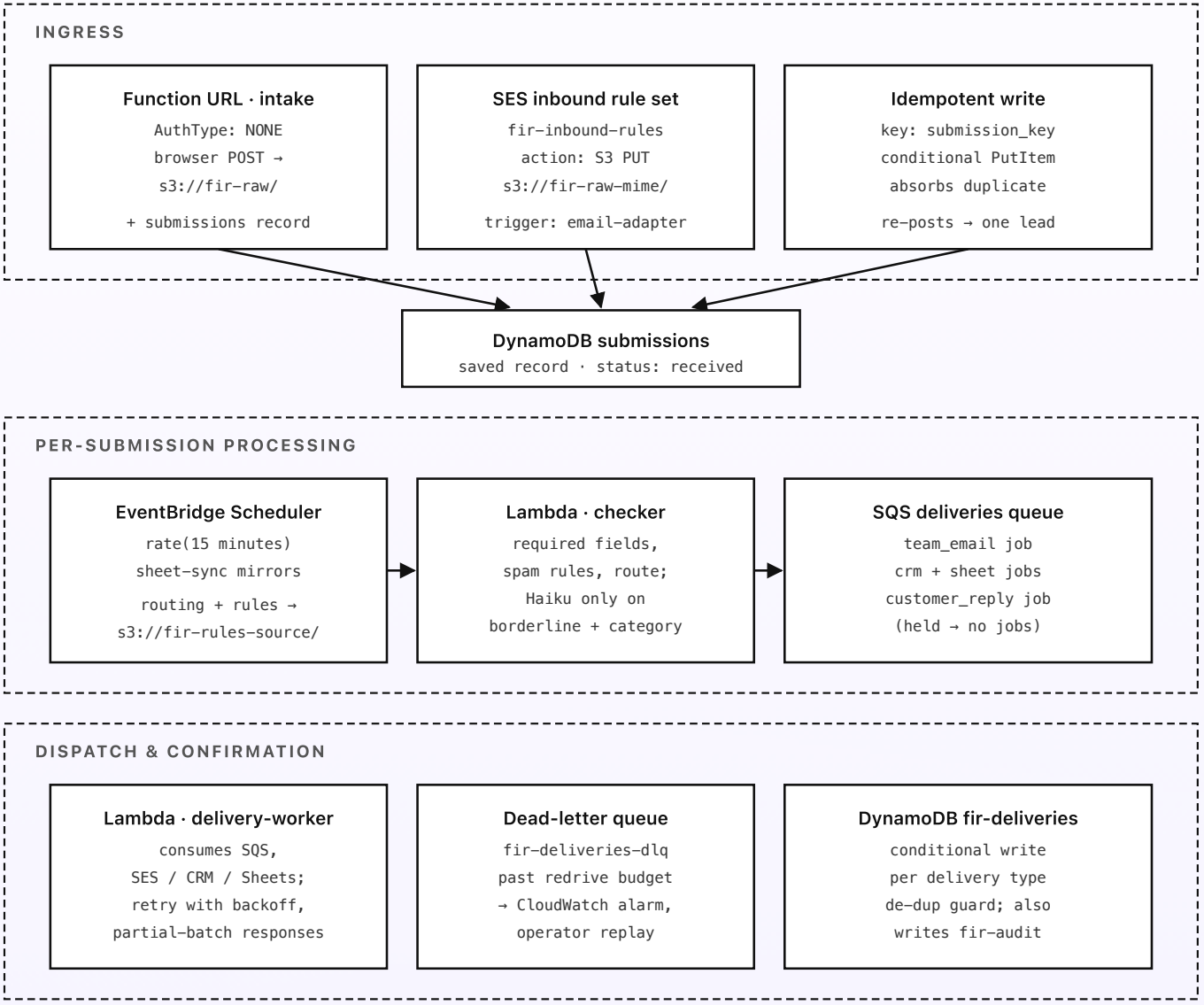
Engineering reference: the form intake router architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the Function URL setup, the SQS and dead-letter queue config, the DynamoDB schemas, and the idempotency design. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). Lambda Function URLs, SQS, SES, Bedrock cross-Region inference, and DynamoDB 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 downstream tool being briefly unreachable, which the queue already handles, 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



Every submission is saved and acknowledged first — and every delivery is logged to fir-audit.

Fig 7. AWS topology, in three bands: ingress (three capture lanes into the saved record), per-submission processing (the checker emitting fan-out jobs), dispatch and confirmation (delivery workers with retries, a dead-letter queue, and an idempotent log). Every step is queue- or event-driven; the only synchronous hop is intake-door → acknowledgment.

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-door`** — Lambda Function URL, `AuthType: NONE`, CORS locked to your site origins. On each POST it parses the body, writes the raw payload to `s3://fir-raw/<submission_key>`, performs a conditional `PutItem` on the `submissions` table keyed by `submission_key` (so a duplicate re-post is a no-op that returns the same 200), and returns the acknowledgment. It then invokes `checker` asynchronously (`InvocationType: Event`) so the customer round-trip never waits on validation. Memory: 256 MB. Timeout: 10 s.
- **`email-adapter`** — S3 PUT trigger on `s3://fir-raw-mime/`. Parses the inbound MIME from the email-fallback lane, maps the email's fields to the canonical submission shape with a `submission_key` derived from the message id, writes to `s3://fir-raw/` and the `submissions` table, and invokes `checker`. Falls back to a permissive field parser for plain-text bodies. Memory: 256 MB. Timeout: 30 s.

- **`checker`** — invoked by `intake-door` / `email-adapter`. Reads the saved submission and `s3://fir-rules-source/rules.json`. Validates required fields and formats; runs the deterministic spam stack (honeypot, time-on-page, per-address rate limit via the `fir-rate-limit` table, banned patterns). Calls Bedrock Haiku 4.5 only on a borderline spam score or an unknown category. Resolves the routing rule from `s3://fir-rules-source/routing.json`. On `routed-ready`, emits one SQS message per delivery to the `fir-deliveries` queue; on `held`, updates status and stops. Memory: 512 MB. Timeout: 30 s.
- **`delivery-worker`** — SQS event source on `fir-deliveries` with `BatchSize: 1` and `ReportBatchItemFailures` enabled (partial-batch responses). Per job, switches on `delivery_type`: `team_email` / `customer_reply` via SES `SendEmail`; `crm` via the CRM API; `sheet` via the Sheets API. Before sending it checks `fir-deliveries` for an existing row keyed by `(submission_key, delivery_type)`; if present it acks the message without re-sending. On success it writes that row with a conditional put and an `fir-audit` entry. On failure it raises, letting SQS redrive per the queue's `maxReceiveCount`. Memory: 256 MB. Timeout: 30 s.
- **`sheet-sync`** — EventBridge Scheduler target, every 15 minutes. Uses the Google Sheets API (service-account credentials in Secrets Manager under `fir/google/sa`) to export the routing sheet and the rules tab as JSON and write to `s3://fir-rules-source/` only if changed since the last sync. Memory: 256 MB. Timeout: 30 s.
- **`dlq-alarm`** — SQS event source on `fir-deliveries-dlq`. On any message, marks the affected submission's delivery as `dead-lettered` in `fir-deliveries`, writes `fir-audit`, and publishes to the `fir-cost-alarm` SNS topic's sibling `fir-ops-alarm` for the on-call admin. Does *not* delete the DLQ

message; an operator replays after the downstream tool recovers. Memory: 256 MB.

- `replay` — manual / admin Function URL (IAM-auth). Redrives messages from `fir-deliveries-dlq` back to `fir-deliveries` in batches, or re-drives a specific submission from its saved record. Used after an outage is resolved. Memory: 256 MB. Timeout: 60 s.
- `held-review` — Function URL behind the internal held-queue UI. Lists `held` submissions with their `held_reason`; on release, re-runs `checker` with the spam stack bypassed for that one submission so a false-positive lead routes normally. Memory: 256 MB.

Storage and queues

- **DynamoDB** · `submissions` — one row per submission. PK `submission_key`; attributes: `form_id`, `category`, `fields`, `status` (received/checking/held/routed-ready), `held_reason`, `received_at`. On-demand. TTL on a copy of the raw fields at 90 days (the S3 payload is the long-term store).
- **DynamoDB** · `fir-deliveries` — one row per completed or dead-lettered delivery. PK `(submission_key, delivery_type)`; attributes: `result` (sent/dead-lettered/duplicate-skipped), `target`, `sent_at`, `attempts`. On-demand. The conditional write on this table is the de-dup guard.
- **DynamoDB** · `fir-audit` — one row per action of any kind. PK `(submission_key, ts)`; attributes: `action`, `delivery_type`, `result`, `actor`. On-demand. No TTL — long-term audit trail.

- **DynamoDB** · `fir-rate-limit` — sliding-window counters for the spam rate check. PK `source_ip`; attribute `count` with a short TTL so windows self-expire. On-demand.
- **SQS** · `fir-deliveries` — standard queue, visibility timeout 6× the worker timeout, `maxReceiveCount: 6` in the redrive policy to `fir-deliveries-dlq`. Backoff is the natural product of visibility-timeout redelivery.
- **SQS** · `fir-deliveries-dlq` — dead-letter queue, 14-day retention, triggers `dlq-alarm`.
- **S3** · `fir-raw` — canonical raw payload per submission. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3** · `fir-raw-mime` — raw inbound MIME from the email-fallback lane. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3** · `fir-rules-source` — mirrored routing table and rules as JSON. Versioning enabled so a bad sheet edit rolls back in one click.

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, both in `checker`: the borderline-spam second-opinion and the category guess for generic contact forms. Heavier reasoning (Sonnet 4.6) is not used — neither call justifies it.
- **Embeddings.** Not used. Routing is a deterministic table lookup keyed by `form_id` and category; spam is rule-driven. No Knowledge Base, no S3 Vectors.

- **Quotas.** Default account quotas are more than enough at SMB volume. The hot path doesn't call Bedrock; only the minority of submissions that are borderline or uncategorized do.

Function URL and ingress

- The `intake-door` Function URL is `AuthType: NONE` (public, by design — it's a form endpoint) with CORS `AllowOrigins` pinned to your site domains and `AllowMethods: POST`. The function rejects bodies over a small size cap and requires the `form_id` and `submission_key` fields.
- Abuse control is layered, not at the edge: the spam stack's rate limit (the `fir-rate-limit` table) plus the honeypot and time-on-page checks. A reserved-concurrency cap on `intake-door` bounds a flood's blast radius.
- For the email-fallback lane, set the MX record on a dedicated subdomain (e.g. `forms.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`; the SES inbound rule set `fir-inbound-rules` does spam-scan → S3 PUT to `s3://fir-raw-mime/<message-id>` → stop.

IAM (least privilege per Lambda)

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

- **intake-door role:** `s3:PutObject` on `fir-raw`; `dynamodb:PutItem` (conditional) on `submissions`; `lambda:InvokeFunction` on `checker`. *No* `bedrock:*`, no SES, no SQS.

- **checker role:** `s3:GetObject` on `fir-raw` and `fir-rules-source`; `dynamodb:GetItem` + `UpdateItem` on `submissions` and `fir-rate-limit`; `sqs:SendMessage` on `fir-deliveries`; `bedrock:InvokeModel` on the Haiku ARN.
- **delivery-worker role:** `sqs:ReceiveMessage` + `DeleteMessage` on `fir-deliveries`; `ses:SendEmail` from the verified sender identity; `dynamodb:PutItem` (conditional) on `fir-deliveries` and `PutItem` on `fir-audit`; `secretsmanager:GetSecretValue` on the CRM and Sheets secrets; outbound network access to the CRM API host and `sheets.googleapis.com`.
- **dlq-alarm / replay roles:** `sqs:*` scoped to `fir-deliveries-dlq` and (replay only) `sqs:SendMessage` on `fir-deliveries`; `sns:Publish` on `fir-ops-alarm`; `dynamodb:UpdateItem` on `fir-deliveries` and `fir-audit`.
- **sheet-sync role:** `secretsmanager:GetSecretValue` on `fir/google/sa`; `s3:PutObject` on `fir-rules-source`; outbound network to `sheets.googleapis.com`.

Idempotency and exactly-once-effect

The system is "at-least-once" end to end and leans on two conditional writes to get exactly-once *effect*. At ingress, the `submissions` `PutItem` is conditional on `attribute_not_exists(submission_key)` — a duplicate re-post finds the row present, skips creation, and returns the original acknowledgment. At egress, the `fir-deliveries` `PutItem` is conditional on `attribute_not_exists((submission_key, delivery_type))` — a redelivered SQS message whose work already completed is detected and acked without a second send. Both keys are derived deterministically (the browser-generated

`submission_key`, and the fixed `delivery_type` enum), so retries always collide with their own prior success rather than creating a new effect.

SES outbound

- Verify a sender identity at `forms@your-company.com` with DKIM and SPF on the parent domain; request production access out of sandbox.
- Two template families in the rules doc: the team-notification template (full submission, link to the held-review UI) and the per-form customer auto-reply template. Both rendered by `delivery-worker`; no model in the rendering path.
- Set the SES configuration set to publish bounce and complaint events to an SNS topic so a customer-reply address that hard-bounces is flagged rather than silently retried.

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:** `fir-deliveries-dlq` depth > 0 (a delivery type is failing); `intake-door` 5xx rate > 1% in 5 min (the door is the one piece that must always answer); checker error rate > 1% in 24h.
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `fir-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for the Sheets API live in Secrets Manager under `fir/google/sa`. CRM API tokens live under `fir/crm/token`. The SES sender identity lives in IAM and the verified-domain config. The allowed CORS origins, the spam thresholds, the rate-limit window, the category list, and the admin fallback address all live in Parameter Store under `/fir/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 `fir-raw` and `fir-rules-source` so a bad payload or sheet edit can be rolled back, and keep the SQS redrive policy and DLQ in the same stack as the worker so `maxReceiveCount` and the queue are versioned together. Total deployable surface: around eight Lambdas, four DynamoDB tables, two SQS queues (main + DLQ), three S3 buckets, one EventBridge Scheduler rule, 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.