7-PART SERIES · FREE COMPANION

Lead enricher

A serverless enricher that catches every raw inbound lead the moment it arrives, cleans the messy fields, fills in the company from the email domain, checks it against your CRM so the same person doesn't land twice, and quietly drops the obvious junk — all before anything is written. It does not score leads and it does not route them; it cleans and completes them, then hands a tidy record to the CRM. Designed on AWS for about $2.50/month at typical small-business volume. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Lead enricher

PART 1 OF 7

JUNE 23, 2026 PART 1 OF 7 · LEAD ENRICHER SERIES ~9 MIN READ

A lead enricher on AWS for a few dollars a month

A small business collects more leads than anyone has time to tidy. The form submission with the name in all caps and a phone number missing its country code. The webhook lead from a free email address that turns out to be a competitor poking around. The same person who fills in the contact form twice in a week and lands in the CRM as two half-complete contacts. The lead with a company name but no idea what the company actually does. Cleaning each one by hand is dull, and it's the first thing that slips when the week gets busy — so the CRM slowly fills with duplicates, junk, and half-records. This post walks through the design of a small enricher that catches every raw lead, cleans it, completes it from public data, checks it against what's already in the CRM, and drops the obvious junk — before anything is written.

KEY TAKEAWAYS

- Three lanes bring leads in: a form/webhook lane, an emailed-enquiry lane, and a manual upload — all landing as raw, messy records.
- Every lead runs the same five-stage pipeline: normalise, enrich, dedupe, junk-filter, then push to the CRM.
- It cleans and completes leads — it does *not* score them and it does *not* route them. That's a different job, deliberately left out.
- Clean, enriched records go to the CRM; obvious junk is logged and dropped before anything is written.
- Designed on AWS for about $2.50/month at typical small-business volume. It never overwrites a field a human already entered.

The whole system on one page

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

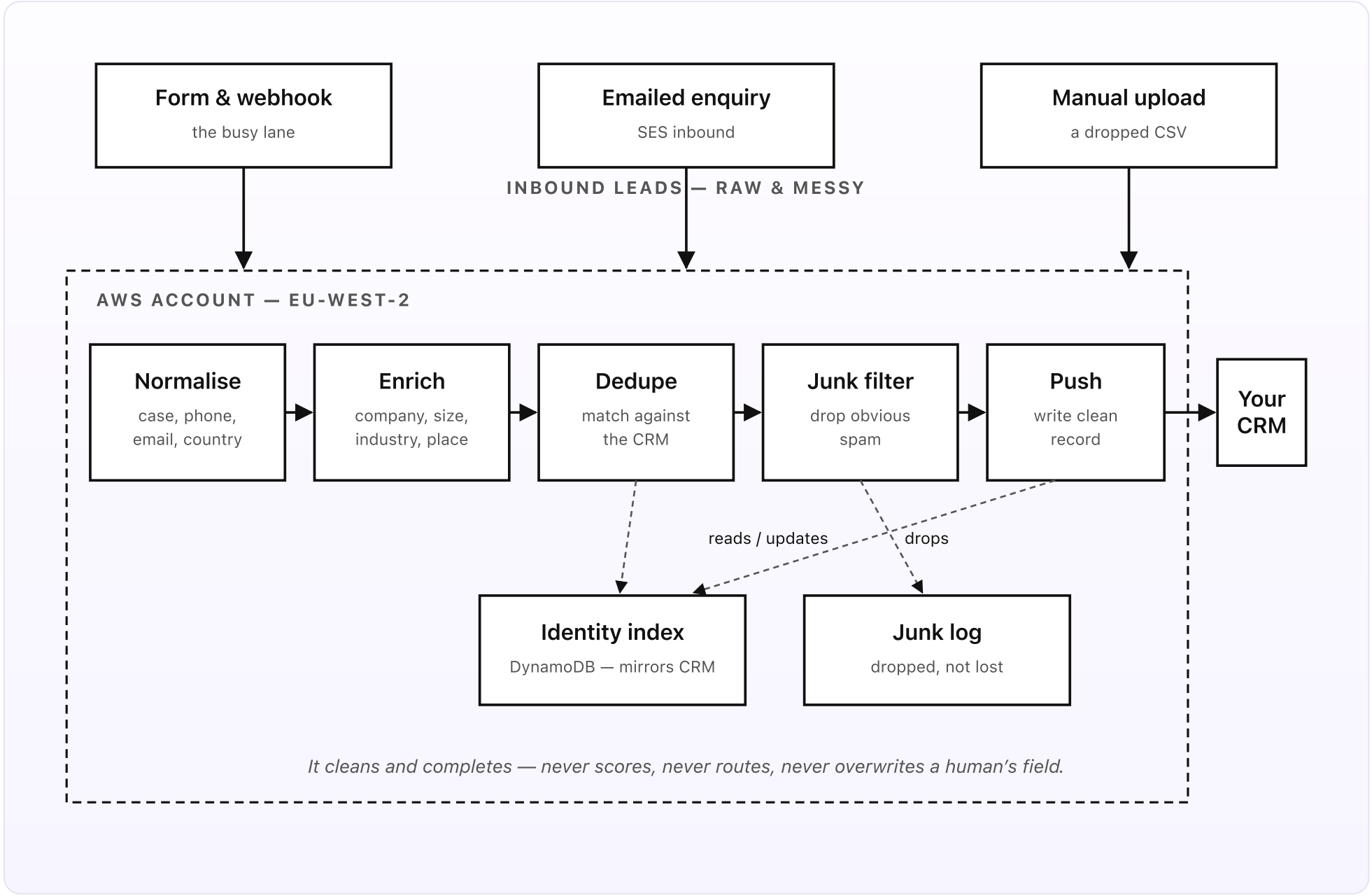

Form & webhook
the busy lane
Emailed enquiry
SES inbound
Manual upload
a dropped CSV
INBOUND LEADS — RAW & MESSY
AWS ACCOUNT — EU-WEST-2
Normalise
case, phone, email, country
Enrich
company, size, industry, place
Dedupe
match against the CRM
Junk filter
drop obvious spam
Push
write clean record
Your CRM
reads / updates
drops
Identity index
DynamoDB — mirrors CRM
Junk log
dropped, not lost
It cleans and completes — never scores, never routes, never overwrites a human's field.

Fig 1. Three lanes in, one pipeline, the CRM on the far side. Every lead is normalised, enriched, deduped against the CRM, and junk-filtered before a clean record is written. Junk is logged and dropped; nothing reaches the CRM unchecked.

What you set up once

- **The lead lanes.** Most leads arrive through the *form/webhook lane*: your website form, an ad platform, or a partner posts a JSON payload, and a small intake function drops the raw body straight into S3 and onto a queue. A second *email lane* catches enquiries sent to an address like `hello@your-company.com` — SES writes the raw message to S3 and the same pipeline picks it up. A third *manual lane* lets someone drop a CSV of leads (from an event, say) into a folder. Whatever the source, the lead enters as a raw, messy record — nothing is assumed clean.
- **The enrichment source.** One public firmographic data provider, reached over its API, with the key held in Secrets Manager. This is what turns an email domain into a company name, a rough headcount, an industry, and a location. It's also the one line on the bill that grows with volume, so the design caches aggressively and never calls it for a lead that's already been flagged as junk.
- **The identity index.** A small DynamoDB table that mirrors the people already in your CRM on a few normalised keys — lowercased email, the phone number in international format, and a name-plus-company fingerprint. It's seeded once from a CRM export and kept current as new records are written. The dedupe stage reads it; the push stage updates it. The CRM stays the source of truth; this is just a fast, cheap mirror to check against.

What runs on every lead

- **Normalise.** The first stage turns the raw fields into canonical ones: the name to sensible casing, the email to a single lowercase form, the phone to E.164, and the country resolved from whatever clues are present. It's mostly plain Python, with one small Bedrock Haiku 4.5 call reserved for the genuinely fuzzy parts — splitting an awkward full name, or inferring a country when only a dialling code is present. Part 2 walks through it.
- **Enrich.** The clean email domain becomes the key for a firmographic lookup: company name, size band, industry, and headquarters location, attached to the record with a note of where each field came from. Cached results mean the same domain is never looked up twice in a hurry. Part 3 covers it.
- **Dedupe.** The enriched lead is checked against the identity index — first for an exact match on a normalised key, then a fuzzy pass for near-matches, with Bedrock judging only the genuinely ambiguous cases. A confident match links the lead to the existing person; a confident non-match is written as new. Part 4 explains the matching.
- **Junk filter.** Before anything is written, the lead faces a layered junk check: cheap deterministic rules first (disposable domains, gibberish, role addresses), a bounded model judgement only for the borderline ones. Obvious junk is logged and dropped. Part 5 is about deciding what isn't worth keeping.
- **Push.** A clean, enriched, deduped, non-junk lead is written to the CRM — created as a new contact, or attached to the existing one if dedupe found a match. It never overwrites a field a human already entered; it only fills genuine gaps. The identity index is updated so the next duplicate is caught.

In plain words

A lead arrives from your website form: name “jane DOE”, email “Jane.Doe@acme-widgets.co.uk ” (capitalised, trailing space), phone “07700 900123”, company blank, message “interested in a quote.” The enricher normalises it to “Jane Doe”, `jane.doe@acme-widgets.co.uk`, and `+447700900123`, country United Kingdom. It takes the domain `acme-widgets.co.uk`, looks it up, and fills in: Acme Widgets Ltd, 50–200 staff, industrial supplies, Birmingham. It checks the identity index, finds that Jane already enquired three weeks ago, and links this enquiry to her existing record rather than creating a second “Jane Doe.” It runs the junk checks — real domain, real name, plausible message — and writes the completed record. A rep opens the CRM to a single, full contact with a company profile attached, not a bare name and a guess.

The cost of running this is about $2.50 a month at small-business volume. The cost of *not* running it is a CRM that fills with duplicates nobody trusts, junk that wastes a rep’s morning, and half-records that make every report a little bit wrong.

DESIGN RULES THAT SHAPED EVERY DECISION

- Clean and complete, full stop. It does not score leads and it does not route them — those are someone else's job, deliberately kept out.
- Deterministic by default. The model is called only for fuzzy judgement, and its output is always bounded by plain rules afterwards.
- Never overwrite a human's field. The enricher fills genuine gaps and links duplicates; it never replaces a value somebody already entered.
- Junk is dropped, never silently lost. Every dropped lead is written to a log with the reason, so you can audit and reverse a bad call.
- Check before you spend. The cheap junk rules run before the paid enrichment lookup, so junk never costs you an API call.
- One region, no always-on compute. Everything is event-driven and serverless, so it costs almost nothing when it's quiet.

Why this shape

Most small teams handle raw leads in one of three ways: they let the form write straight into the CRM and tidy up later (which never happens), they paste leads in by hand (slow, and inconsistent between people), or they buy a heavy all-in-one platform that scores and routes and emails and does ten other things they don't need. The first fills the CRM with junk and duplicates. The second doesn't scale past a few leads a day. The third is expensive and couples cleaning — a job with a clear right answer — to scoring and routing, jobs that depend on your sales process and change all the time.

The setup above does exactly one thing well: it makes sure that whatever lands in your CRM is clean, complete, deduped, and real. It deliberately stops there. Scoring a lead and routing it to the right rep is a separate concern with its own logic; bolting it on here would mean redeploying the cleaner every time the sales playbook changes. By keeping the enricher to cleaning and completing, the rules that matter — what counts as junk, which fields to fill — stay stable, and the expensive, changeable parts live somewhere else.

The next four posts walk through each stage in turn: how a raw lead gets normalised, how it gets enriched from public data, how a duplicate gets caught against the CRM, and how junk gets filtered out. One diagram per post. A cost breakdown and a full engineering reference at the end.

PART 2 OF 7

JUNE 23, 2026 PART 2 OF 7 · LEAD ENRICHER SERIES ~8 MIN READ

How a raw lead gets normalized

A raw lead arrives looking like a person typed it in a hurry, because one did. The name is “jane DOE”, the phone is “(0) 7700 900123”, the email has a trailing space and a capital letter, and the country is blank. None of that is wrong, exactly — it’s just not in a shape anything downstream can match or trust. This post is about the first stage: turning the messy, partial fields of a raw lead into clean, canonical ones, deterministically where it can and with a small model call only where fuzzy judgement actually helps.

KEY TAKEAWAYS

- Normalising turns the messy, partial fields of a raw lead into canonical ones: clean name, one email form, an E.164 phone, a resolved country.
- Most of it is plain deterministic Python — case-fixing, trimming, lower-casing, phone parsing — with predictable, testable results.
- One small Bedrock Haiku 4.5 call handles only the fuzzy bits: splitting an awkward full name, or inferring a country from thin clues.
- The original raw payload is kept untouched in S3; normalisation writes a new clean copy, so nothing is ever destroyed in place.
- If a field can't be normalised confidently, it's left blank and flagged — the enricher never guesses a value it can't support.

Why normalise first

Everything downstream depends on clean fields. The enrichment lookup needs a real email domain, not `Jane.Doe@Acme-Widgets.CO.UK ` with a stray capital and a trailing space. The dedupe stage matches on normalised keys — if one copy of Jane has `+447700900123` and another has `07700 900123`, they won't line up and she'll land twice. The junk filter needs a parsed email to check the domain against a blocklist. So normalisation runs first, and its single job is to take whatever shape the lead arrived in and produce canonical fields the rest of the pipeline can rely on.

The work splits cleanly into two kinds. Most of it has a single right answer and no judgement involved — trim the whitespace, lowercase the email, parse the phone. That's plain Python, fully deterministic, and easy to test. A smaller part is genuinely fuzzy — is "Jan Van Der Berg" a first name of "Jan" and a surname of "Van Der Berg", or something else? — and that's where one small model call earns its place. The design keeps the two strictly separate, and never lets the model touch a field the rules can already settle.

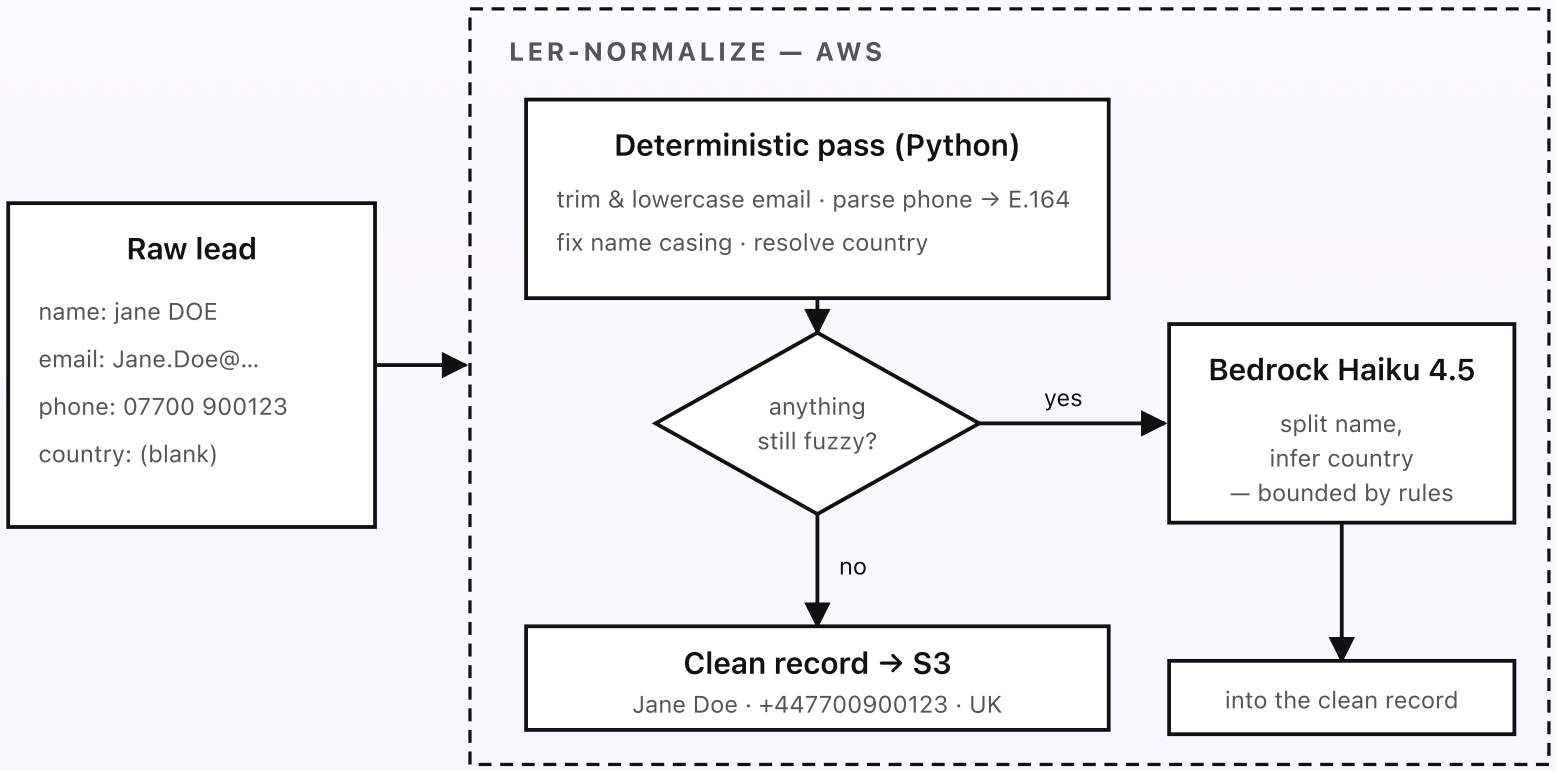
The deterministic pass

The first pass is all rules, applied field by field:

- **Email.** Trim, lowercase, strip a display name if the value arrived as `"Jane Doe" <jane@acme.co.uk>`, and validate the basic shape. The domain is pulled out and kept separately — it's the key the enrichment stage will use. A plus-tag like `jane+newsletter@acme.co.uk` is preserved in the stored email but the bare form is also recorded for matching.
- **Phone.** Parsed against the resolved country into E.164 (`+447700900123`), the single canonical form the dedupe index keys on. A number that can't be parsed to a valid form is left blank and flagged, rather than stored half-formatted — a half-parsed number is worse than none, because it matches nothing and looks deceptively complete.
- **Name.** Whitespace collapsed, obvious all-caps or all-lowercase fixed to title case with a sensible particle list (so "McDonald" and "van der Berg" survive), and surrounding junk like trailing commas removed.
- **Country.** Resolved from explicit fields first (a country dropdown, a postal code pattern), then from the phone's dialling code, then from the email's country-

code top-level domain as a weak last hint. Each source carries a confidence, and the strongest wins.

By the end of this pass, most leads are fully normalised with no model call at all. The clean fields are written as a new record in S3 alongside — never overwriting — the raw original.



A field that can't be normalised confidently is left blank and flagged — never guessed.
The raw payload stays untouched in S3; normalisation only ever writes a new copy.

Fig 2. Normalisation is a deterministic pass first, with one small Bedrock call reserved for the genuinely fuzzy fields. The raw payload is never modified; a clean copy is written alongside it.

Where the model earns its place

Some fields resist rules. A single “full name” box that contains “Dr. Maria Garcia-Lopez” needs to become a title, a first name, and a compound surname — and the split isn’t something a regex gets right across cultures. A lead with no country field, a generic `.com` email, and a phone number written as “+1 ext. 204” needs a judgement call to land on the United States. These are the cases the deterministic pass flags as ambiguous and hands to one Bedrock Haiku 4.5 call.

The call is deliberately narrow. It’s given only the specific ambiguous field and asked for a structured answer — first name, last name, and country code — not invited to rewrite the whole record. Crucially, its output is then bounded by the same rules: a country it returns must be a real ISO country, a name split must reconstruct to the original characters, and anything outside those bounds is rejected and the field left blank and flagged. The model proposes; the rules dispose. That keeps a confident-sounding but wrong guess from quietly becoming a stored fact.

Because the model only fires when the deterministic pass couldn’t settle a field, most leads never reach it — which is exactly why the Bedrock line on the cost breakdown stays small. A clean lead from a UK form with separate name fields and a valid phone is fully normalised by Python alone.

What normalisation deliberately doesn’t do

It doesn’t enrich — it won’t go and find the company behind the domain, that’s the next stage. It doesn’t dedupe — it doesn’t care yet whether this Jane already exists. And it doesn’t judge whether the lead is junk; a gibberish name is

normalised just like a real one and left for the junk filter to catch with full context. Each stage does one job, so each one is simple to reason about and to test. Normalisation's contract is narrow and total: take messy fields in, hand canonical fields out, and never invent a value you can't defend.

WHY THIS SHAPE

- Deterministic first, model last. Rules settle the fields that have a right answer; the model only sees what's genuinely ambiguous.
- The model is fenced in. Its output must pass the same rules as everything else — a real country, a reconstructable name — or it's rejected.
- Blank beats wrong. A field that can't be normalised confidently is left empty and flagged, not filled with a plausible guess.
- The raw lead is immutable. Normalisation writes a new clean copy to S3; the original payload is always there to re-run or audit against.
- Canonical forms are chosen for matching. E.164 phones and lowercased emails exist so the dedupe stage downstream actually lines duplicates up.

PART 3 OF 7

JUNE 23, 2026 PART 3 OF 7 · LEAD ENRICHER SERIES ~7 MIN READ

How a lead gets enriched

A clean lead is still a thin one. You have a name, a phone, an email, and a country — but not what the person actually does, how big their company is, or what industry they're in. A lot of that is hiding in plain sight in the email domain. This post is about the enrichment stage: taking the company from the domain, looking it up against public firmographic data, and attaching size, industry, and location to the record — carefully, cached, and always with a note of where each field came from so nothing is silently invented.

KEY TAKEAWAYS

- Enrichment turns a clean email domain into a company profile: name, size band, industry, and location, attached to the lead.
- The work is a single firmographic API lookup keyed on the domain, with the API key held in Secrets Manager and never in code.
- Results are cached per domain in DynamoDB, so the same company is never paid for twice in a short window — the cache is what keeps the bill low.
- Free and personal email domains (gmail, outlook) are recognised and skipped, not looked up — there's no company behind them.
- Every enriched field records its source and the lookup date; a field the provider can't supply is left blank, never invented.

From a domain to a company

After normalisation you have a clean lead but a thin one: a real name, a formatted phone, a validated email, a country. What you don't have is any sense of *who* this is in business terms — what company they're from, how big it is, what it does. For a B2B enquiry, that context is most of what makes a lead useful, and a surprising amount of it is recoverable from one field you already cleaned: the email domain.

The enrich stage takes the domain — `acme-widgets.co.uk` — and looks it up against a public firmographic data provider. The provider returns a structured company record: the legal name, a headcount band, an industry classification, and

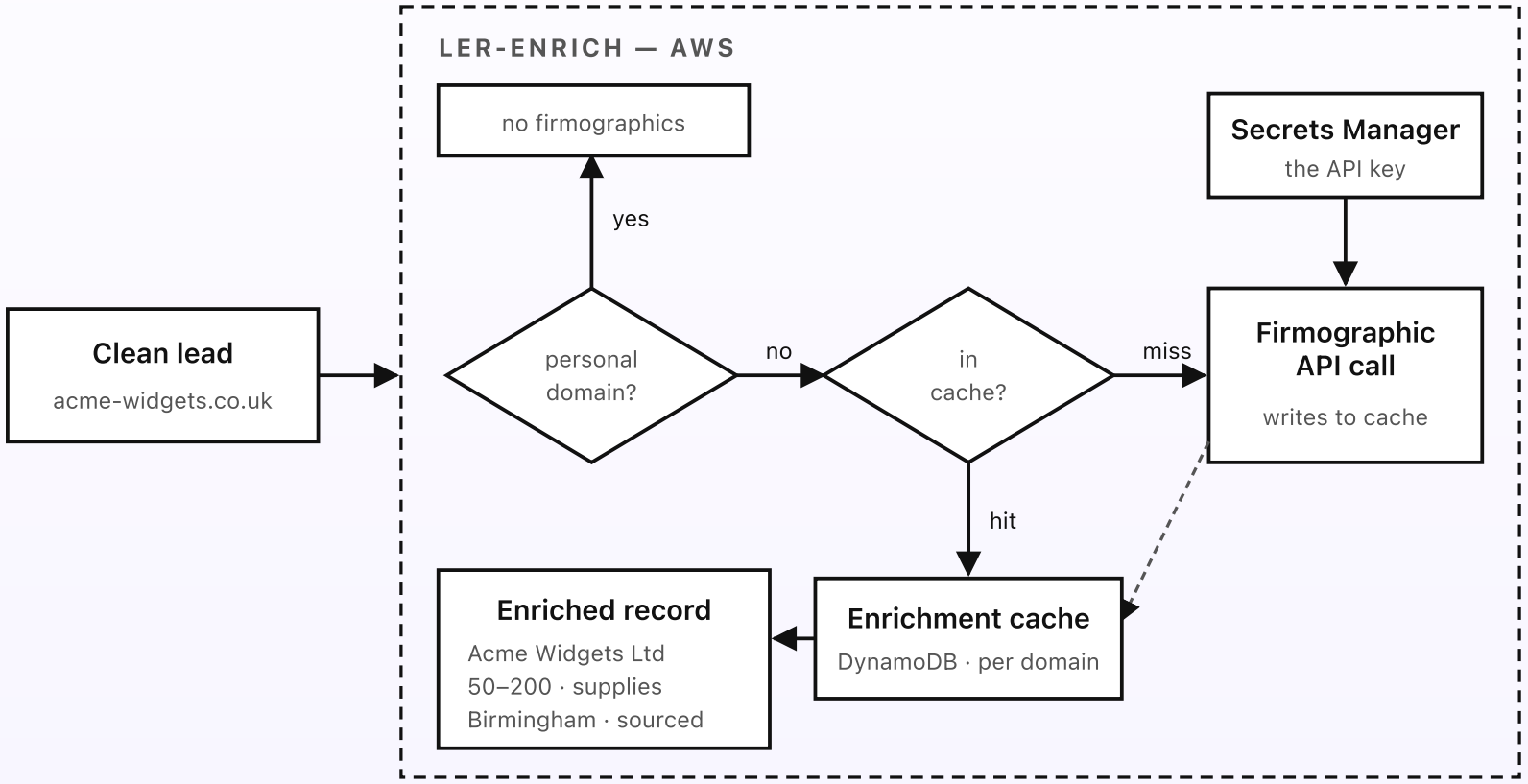
a headquarters location. The enricher attaches those to the lead, each one tagged with where it came from and when it was fetched. That's the whole job: complete the record from public data, carefully and traceably.

The lookup, and the key behind it

The lookup itself is a single authenticated API call. The provider's key never appears in the function code or the environment — it lives in Secrets Manager, and the enrich function fetches it at runtime with a tightly scoped IAM permission to read that one secret. Rotating the key, or swapping providers entirely, is a change in one place with no redeploy of the pipeline.

Two things happen before any call is made, and both exist to save money:

- **Personal domains are skipped.** If the domain is a known free or personal-email host — gmail.com, outlook.com, yahoo, icloud, and a maintained list of others — there is no company to look up, so the enricher doesn't try. The lead keeps its clean fields and is simply marked "personal email, no firmographics," and the API call is never spent.
- **The cache is checked.** Companies don't change overnight, and the same domains recur constantly — several people from the same firm enquire over a month. So every successful lookup is cached in DynamoDB, keyed by domain, with a time-to-live. A second lead from `acme-widgets.co.uk` within the cache window reuses the stored profile and costs nothing. At small-business volume this cache absorbs a large share of the traffic, which is exactly why the enrichment line on the bill is dollars and not tens of dollars.



A field the provider can't supply is left blank, never invented. The cache is what keeps the bill low.

Fig 3. Enrichment skips personal domains, checks the per-domain cache, and only pays for a fresh lookup on a cache miss. The key lives in Secrets Manager; every field carries its source.

Attaching the profile, honestly

When a lookup returns, the enricher doesn't just dump the provider's fields onto the lead. It maps them to a small, fixed schema — company name, size band, industry, location — and tags each value with two things: the source ("firmographic provider") and the date it was fetched. That provenance matters later. When the dedupe stage compares this lead's company against an existing record, it can tell a human-entered company name from an auto-enriched one. And when the push stage decides whether to fill a CRM field, it knows it's filling a gap with sourced data, not overwriting something a rep typed.

A field the provider can't supply is left blank. If there's no headcount for a tiny one-person consultancy, the size band stays empty rather than being filled with a default or a guess. The same restraint runs through the whole system: an empty field is honest, a fabricated one is a landmine that surfaces months later as a wrong number in a report.

When the lookup fails

External APIs are not always there. The provider might rate-limit, time out, or be down entirely. The enricher treats an enrichment failure as a soft failure, never a hard one: the lead keeps its clean normalised fields, is marked "enrichment pending," and continues through dedupe and the junk filter so a real enquiry isn't held hostage to a third party's uptime. The pending lookup is queued for a retry, and an EventBridge-scheduled backfill (covered in the engineering reference) sweeps up anything still un-enriched once the provider recovers. A lead never gets stuck, and it never gets dropped just because the enrichment step couldn't complete.

WHY THIS SHAPE

- One key, one place. The provider key lives in Secrets Manager with a scoped read permission; rotating or swapping it touches nothing else.
- Cache before you call. A per-domain DynamoDB cache absorbs repeat companies, turning a per-lead cost into a per-new-company one.
- Don’t pay for nothing. Personal and free domains are skipped outright — there’s no company there to find.
- Provenance on every field. Each enriched value carries its source and date, so downstream stages can tell sourced data from human input.
- Enrichment failures are soft. A provider outage marks the lead pending and lets it continue; a scheduled backfill finishes the job later.

PART 4 OF 7

JUNE 23, 2026 PART 4 OF 7 · LEAD ENRICHER SERIES ~7 MIN READ

How a duplicate gets caught

The same person reaches you more than once. They fill in the form on Monday, reply to an ad on Thursday, and get imported from a list at the weekend — three enquiries, one human. If each one lands in the CRM as a fresh contact, your sales pipeline quietly doubles and nobody trusts the numbers. This post is about the dedupe stage: how the enricher checks a clean, enriched lead against the people already in your CRM, catches the duplicate even when the details don’t line up exactly, and links rather than blindly merges.

KEY TAKEAWAYS

- A DynamoDB identity index mirrors the people already in your CRM on normalised keys: lowercased email, E.164 phone, and a name-plus-company fingerprint.
- An exact-key check catches the obvious repeats instantly and for almost nothing — most duplicates are caught here.
- A fuzzy pass compares normalised name, company, and domain for near-matches that don't share an exact key.
- Only genuinely ambiguous near-matches go to Bedrock Haiku 4.5 for a same-person / different-person judgement.
- A confident match links the lead to the existing record; a confident non-match is written as new. It links, it never blindly merges.

One person, many enquiries

The same human reaches you through more than one door. Jane fills in the website form on Monday, replies to an email campaign on Thursday, and turns up in a list import at the weekend. That's three enquiries and one person. If each one lands in the CRM as a fresh contact, the pipeline count doubles, the same person gets three follow-up emails, and every report built on contact counts is quietly wrong. The dedupe stage exists to make sure a lead that's really an existing person gets attached to that person, not added as a stranger.

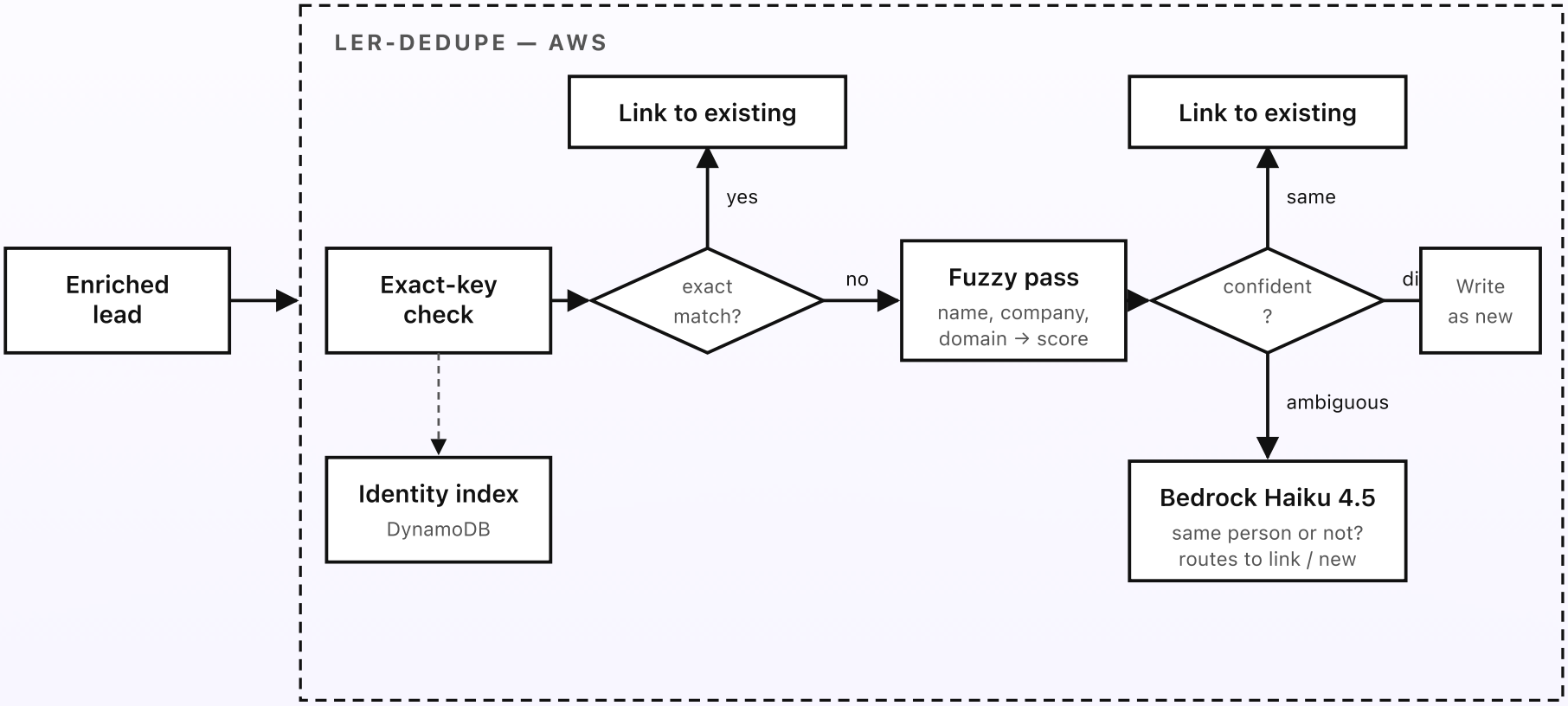
The hard part is that the three enquiries rarely look identical. The form gave a work email; the campaign reply came from a personal one. One has a phone, one doesn't. The name is "Jane Doe" once and "Jane M. Doe" another time. Exact string matching catches the easy cases and misses the ones that matter most. So dedupe runs in layers: cheap and certain first, fuzzy where it has to be, and a model only for the genuinely undecidable.

The identity index

Dedupe doesn't query the CRM on every lead — that would be slow, rate-limited, and expensive. Instead it reads a small DynamoDB *identity index* that mirrors the CRM. Each person in the CRM has entries under a few normalised keys:

- **Email key** — the lowercased, plus-stripped email address.
- **Phone key** — the E.164 phone number.
- **Name-and-company fingerprint** — a normalised, lower-cased combination of the person's name and their company, used when neither email nor phone lines up.

The index is seeded once from a CRM export and kept current: every time the push stage writes or links a record, it updates the index too. Because the keys are exactly the canonical forms the normalise stage produces, a new lead's keys can be looked up directly — no re-cleaning, no guessing.



It links to a record — it never blindly merges, and never overwrites a field a human already entered.

Fig 4. Dedupe runs cheap and certain first: an exact-key check against the identity index, then a fuzzy pass, with Bedrock judging only the genuinely ambiguous near-matches.

Exact first, fuzzy second, model last

The first check is exact and instant. The lead's email key, phone key, and name-and-company fingerprint are looked up directly in the identity index. A hit on any of them is a confident duplicate — the same email is the same person — and the lead is linked to the existing record with no further work. This catches the large majority of real duplicates, costs a couple of DynamoDB reads, and involves no model at all.

When no exact key matches, the fuzzy pass runs. It pulls the small set of candidate records that share *something* — the same company, a similar surname, the same email domain — and scores each against the lead on normalised name similarity, company match, and domain. A high score with corroborating signals (same company, same domain, near-identical name) is treated as the same person; a low score is a clean stranger. Both of those are decided by the deterministic score against tuned thresholds — still no model.

Only the middle band — a near-match the score can't confidently call — is handed to Bedrock Haiku 4.5. "J. Doe at Acme" with a personal email versus "Jane Doe at Acme Widgets" with a work email: plausibly the same person, plausibly two. The model is given just the two candidate records and asked for a same-person / different-person verdict with a brief reason. Its answer routes the lead to link or write-as-new, and that judgement is logged. Because the model only sees the ambiguous slice, it's a small fraction of leads — which keeps both the cost and the risk down.

Link, don't merge

A confident match doesn't mean overwriting the existing record with the new one. The enricher *links*: it attaches the new enquiry to the existing person and fills only genuinely empty fields with the new lead's sourced data. If the CRM record already has a phone number and the new lead has a different one, the enricher does not replace it — it records the new number as additional information for a human to reconcile, and leaves the existing value alone. This is the same restraint that runs through the whole system: the enricher completes and connects, it never destroys what a person already entered.

The choice to mirror the CRM in a cheap index, rather than hammer the CRM's own API on every lead, is what makes dedupe fast and affordable. The index can be wrong at the edges — a record added directly in the CRM a minute ago might not be mirrored yet — so the push stage does one final exact check against the CRM itself before creating a contact, as a backstop. Two layers, cheap one first, authoritative one last.

WHY THIS SHAPE

- Cheap and certain first. Exact-key lookups catch most duplicates for two DynamoDB reads and no model call.
- The model only sees the hard slice. Confident same and confident different are decided deterministically; only the ambiguous band reaches Bedrock.
- An index, not the live CRM. Dedupe reads a cheap DynamoDB mirror; the push stage does one authoritative CRM check as a backstop.
- Link, never blindly merge. A match attaches the enquiry to the existing person and fills only empty fields — it never overwrites a human's value.
- Every verdict is logged. Whether matched by key, by score, or by model, the decision and its reason are recorded for audit.

PART 5 OF 7

JUNE 23, 2026 PART 5 OF 7 · LEAD ENRICHER SERIES ~7 MIN READ

How junk gets filtered out

Not every lead is a lead. Some are bots filling in the form with gibberish. Some are disposable email addresses that exist for ten minutes. Some are sales pitches dressed up as enquiries, and some are just test submissions someone left behind. If those reach the CRM they cost you enrichment lookups, clutter the pipeline, and waste a rep's morning. This post is about the junk filter: how the enricher decides, cheaply and defensibly, that a lead isn't worth writing — and how it drops it without ever silently losing something real.

KEY TAKEAWAYS

- The junk filter drops obvious spam and bot submissions before anything is written to the CRM — the last gate before push.
- It runs in layers: cheap deterministic checks first (disposable domains, gibberish, role addresses, honeypots), then a bounded model judgement only for the borderline.
- A cheap spam pre-screen also runs at intake, so the most obvious junk never even reaches the paid enrichment lookup.
- Every dropped lead is written to a junk log with the reason — dropped, never silently lost, and always reversible.
- The bar is deliberately conservative: when a lead is genuinely borderline, it's kept and flagged, not dropped. A missed real lead costs more than a kept junk one.

Not every lead is a lead

An open web form is an open invitation, and not just to customers. Bots fill it with gibberish to test it. Spammers use it to deliver sales pitches dressed up as enquiries. People submit disposable email addresses that exist for ten minutes. Someone on your own team leaves a test submission behind. If any of those reach the CRM, they cost you an enrichment lookup, clutter the pipeline, and waste a rep's time chasing a ghost. The junk filter's job is to decide, cheaply and defensibly, that a lead isn't worth writing — and to do it without ever quietly losing something real.

The filter is the last gate before the push stage. By the time a lead reaches it, the lead has been normalised, enriched, and deduped, so the filter has full context: a parsed email and domain, a company profile (or a note that there isn't one), and the message text. It uses all of that, in layers, cheapest and most certain first.

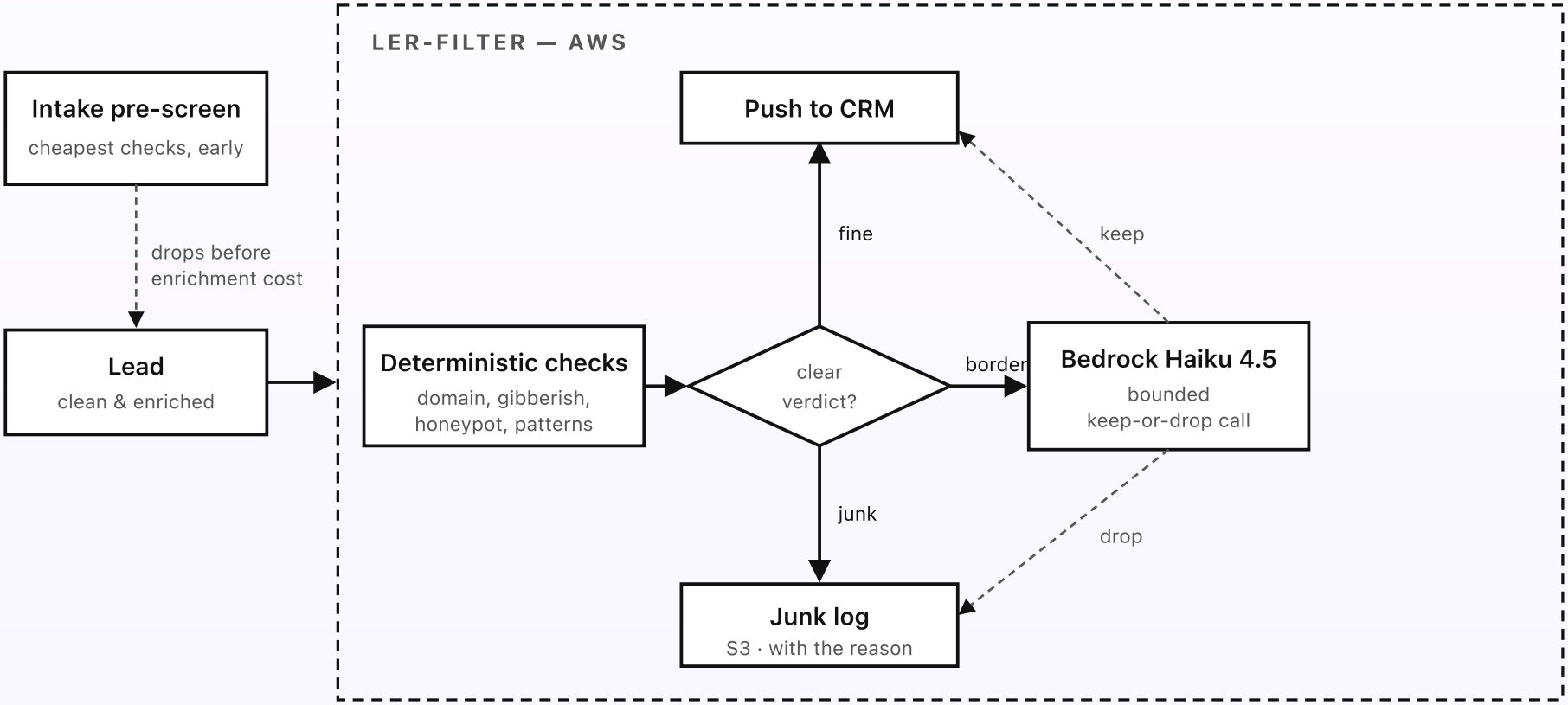
The cheap, deterministic layer

Most junk is obvious, and obvious junk should be caught by a rule, not a model. The deterministic layer runs a set of plain checks:

- **Disposable and throwaway domains.** The email domain is checked against a maintained list of disposable-mail providers. A ten-minute mailbox is almost never a real enquiry.
- **Gibberish.** A name like "asdfgh qwerty" or a message that's a wall of random characters fails simple plausibility checks — vowel ratios, repeated characters, dictionary hits.
- **Honeypot and timing signals.** If the form carried a hidden honeypot field that a human never sees but a bot fills in, or the submission arrived implausibly fast, that's a strong bot tell, passed through from intake.
- **Role and abuse addresses.** Submissions from `postmaster@`, `abuse@`, or obvious test addresses are flagged — not always junk, but rarely a buying customer.
- **Known spam patterns.** Messages matching well-worn spam templates (link-stuffed SEO pitches, cryptocurrency come-ons) are caught by pattern.

Each check produces a signal, not an instant verdict. A single weak signal doesn't drop a lead; the signals are weighed together. A lead that trips several — disposable domain *and* gibberish name *and* a honeypot hit — is dropped with high

confidence and no model call. A lead that trips none sails through. Only the leads in between go to the next layer.



Conservative bar: a borderline lead is kept and flagged, not dropped. Every drop is logged and reversible.

Fig 5. The junk filter weighs cheap deterministic signals first and sends only borderline leads to a bounded model judgement. A cheap pre-screen at intake drops the worst junk before it costs an enrichment lookup.

The bounded model layer

Some junk is subtle. A sales pitch can be written in fluent, plausible English with a real company domain — it passes every deterministic check and is still not a customer. A real enquiry can look thin and odd — a one-word message from a personal email — and still be genuine. These borderline cases are where a model adds real value, so the borderline slice goes to one Bedrock Haiku 4.5 call.

The call is given the message text and the surrounding context — the enriched company, the domain, the deterministic signals — and asked a single bounded question: is this a genuine business enquiry, or unsolicited / spam / test? It returns a verdict and a short reason. The verdict is advisory, not absolute: it routes the lead to push or to the junk log, but it can't do anything else, and its reason is logged alongside the lead so a person can review the call later. The model judges; it never reaches past this one decision.

Dropped, never lost — and a conservative bar

Nothing is ever silently discarded. Every dropped lead is written to a junk log in S3 with the full record and the exact reason it was dropped — "disposable domain + honeypot hit," or "model: unsolicited SEO pitch." That log is the safety net: if the filter ever drops something it shouldn't, you can find it, see why, and replay it through the push stage. A daily digest email (via SES) summarises what was dropped and why, so a person actually looks at the boundary occasionally instead of trusting it blindly.

And the bar is deliberately tilted. The cost of dropping a real lead — a customer you never call back — is far higher than the cost of letting a junk one through, which just gets cleaned up later. So when a lead is genuinely on the line, the filter

keeps it and flags it for a human glance rather than dropping it. The filter is aggressive about *obvious* junk and cautious about everything else. That asymmetry is the whole design: catch the clear waste, never gamble with a real enquiry.

WHY THIS SHAPE

- Layers, cheapest first. Plain rules catch obvious junk; the model only ever sees the genuinely borderline slice.
- Pre-screen at intake. The cheapest spam checks run before enrichment, so the worst junk never costs a paid lookup.
- The model is fenced to one question. It returns keep-or-drop with a reason and can do nothing else; the reason is always logged.
- Dropped, never lost. Every drop goes to a junk log with its reason, and a daily digest keeps a human watching the boundary.
- A conservative bar. Borderline leads are kept and flagged, not dropped — a missed real lead costs far more than a kept junk one.

PART 6 OF 7

JUNE 23, 2026 PART 6 OF 7 · LEAD ENRICHER SERIES ~5 MIN READ

What the lead enricher costs

The whole point of designing this on serverless AWS is that it costs almost nothing when it's quiet and scales its cost with the work, not with the clock. There's no server sitting idle, no gateway charged by the hour. This post adds it up properly: every service, a real monthly figure at a stated volume, and an honest note on which line moves when the volume does. The short version is about $2.50 a month at small-business volume — and the enrichment API is the line that matters.

KEY TAKEAWAYS

- At about 1,000 leads a month, the whole system runs for roughly $2.50 — and most of that is two lines.
- The firmographic enrichment API is the main variable cost; the per-domain cache and skipping personal domains are what keep it near a dollar.
- Bedrock Haiku 4.5 is the second variable line, and it's small because the model fires only on the fuzzy slice of each stage.
- The fixed AWS pieces — Lambda, DynamoDB, S3, SES, SQS — come to pennies; Secrets Manager is a flat 40 cents for the one stored key.
- At 10× the volume the bill lands near $21, almost entirely enrichment and Bedrock. The fixed costs barely move.

What it costs at small-business volume

The reason to build this on serverless AWS is that the cost follows the work. There's no instance billed by the hour while the form sits quiet overnight, no gateway with a monthly minimum. Every piece is event-driven: it runs when a lead arrives and costs nothing the rest of the time. So the bill is dominated not by infrastructure but by the two things that genuinely cost money per lead — the enrichment API call and the occasional model call.

Here's the breakdown at a stated volume of **about 1,000 leads a month** in one region (eu-west-2). The figures are deliberately rounded and a little conservative.

Service	What it does here	~Monthly
Firmographic API	Company lookups on a cache miss — the main variable	$1.00
Bedrock (Haiku 4.5)	Fuzzy normalisation, ambiguous dedupe, borderline junk calls	$0.88
Secrets Manager	One stored key (the enrichment API key)	$0.40
CloudWatch Logs	Logs from every Lambda, 7-day retention	$0.10
DynamoDB (on-demand)	Identity index + enrichment cache reads and writes	$0.05
S3	Raw payloads, clean records, the junk log	$0.05
SES	Daily junk digest + ops alerts	$0.02
Lambda (arm64)	Five pipeline stages + schedules — within the free tier	$0.00
SQS + DLQ	Queues between stages — within the free tier	$0.00
EventBridge Scheduler	Retry + overnight backfill triggers	$0.00
AWS Budgets	Cost alarm (first two budgets free)	$0.00
Total		**~$2.50**

Fig 6. Monthly cost at about 1,000 leads a month in eu-west-2. Two lines — the enrichment API and Bedrock — are most of the bill; everything else is pennies or free-tier.

The line that matters: enrichment

The firmographic lookup is the only line that scales straight with new traffic, so the design works hard to keep it small. Two decisions from Part 3 do most of that work. First, **personal and free email domains are skipped entirely** — a gmail or outlook lead is never looked up, because there's no company behind it, and for many small businesses that's a meaningful slice of inbound. Second, **every successful lookup is cached per domain**, so the repeated companies — several enquiries from the same firm over a month — are paid for once, not each time. At 1,000 leads, after skips and cache hits, you might pay for a few hundred genuine lookups, which at typical per-lookup pricing lands around a dollar. The exact figure depends on your provider; if yours is dearer, this is the line to watch, and the cache is the lever.

Bedrock is the second variable line, and it stays small for the same structural reason the whole system stays cheap: the model only fires on the fuzzy slice. Most leads are normalised by plain Python with no model call, deduped by an exact key with no model call, and passed by the junk filter on deterministic signals alone. Only the awkward names, the ambiguous near-matches, and the borderline junk reach Haiku 4.5 — a fraction of leads, each a small, cheap call. That's why the model line is cents-to-a-dollar, not the headline cost.

At ten times the volume

Push to **about 10,000 leads a month** and the bill lands near **$21** — and the shape is the point. The enrichment API and Bedrock scale roughly with traffic, so those two lines do most of the growing (toward ~$9 and ~$9 respectively, before any

volume discount your provider might offer). DynamoDB, S3, SES, and CloudWatch tick up modestly into the low dollars combined. Lambda and SQS may finally edge past the free tier into pennies. And Secrets Manager stays at exactly 40 cents, because it's one key whether you process ten leads or ten thousand.

That's the serverless dividend in one sentence: the fixed pieces barely move, and the bill you grow into is almost entirely the two services that genuinely do per-lead work. There's no step-change where you suddenly need a bigger instance, and an **AWS Budgets** alarm sits on top of the whole thing so that if a runaway loop or a provider pricing change ever pushes the bill past a threshold you set, you hear about it the same day rather than at the end of the month.

WHY THIS SHAPE

- Cost follows work. Event-driven and serverless throughout, so an idle form costs nothing.
- Two lines, watched closely. The enrichment API and Bedrock are the only things that scale with traffic — everything else is rounding.
- The cache is the cost lever. Per-domain caching and skipping personal domains turn a per-lead enrichment cost into a per-new-company one.
- The model is rationed. Bedrock fires only on the fuzzy slice, which is exactly why it's a small line and not the headline.
- A Budgets alarm guards the edges. If a loop or a price change spikes the bill, you find out the same day.

PART 7 OF 7

JUNE 23, 2026 PART 7 OF 7 · LEAD ENRICHER SERIES ~9 MIN READ

Engineering reference: the lead enricher architecture

This is the enricher with the friendly labels stripped off. Same system, drawn for someone who has to build or review it: the Lambdas and what triggers each one, the EventBridge schedules, the DynamoDB identity index and its exact key schema, the S3 buckets, the SES setup, the IAM scopes that keep each function to its own job, the Bedrock model id, and the single region it all runs in. If the earlier posts are the “what” and the “why”, this one is the “how, exactly.”

KEY TAKEAWAYS

- Seven Lambdas (Python 3.14, arm64): one per lead lane and one per pipeline stage, chained over the EventBridge default bus.
- Two DynamoDB tables on-demand: `ler-identity` (the dedupe index) and `ler-enrich-cache` (per-domain firmographics with a TTL).
- Three S3 buckets: raw payloads (versioned), clean enriched records, and the junk log. SES handles the inbound email lane and the outbound digest.
- EventBridge Scheduler runs three jobs: a retry sweep, an overnight backfill + identity re-sync, and the daily junk digest.
- One Bedrock model (`global.anthropic.claude-haiku-4-5-20251001-v1:0`), one region (eu-west-2), least-privilege IAM per function, no API Gateway and no always-on compute.

The whole architecture, drawn for engineers

Same system as the earlier posts, with the friendly labels off. Three ingress lanes, a five-stage event-driven chain, two outputs, and three scheduled jobs — all in one region.

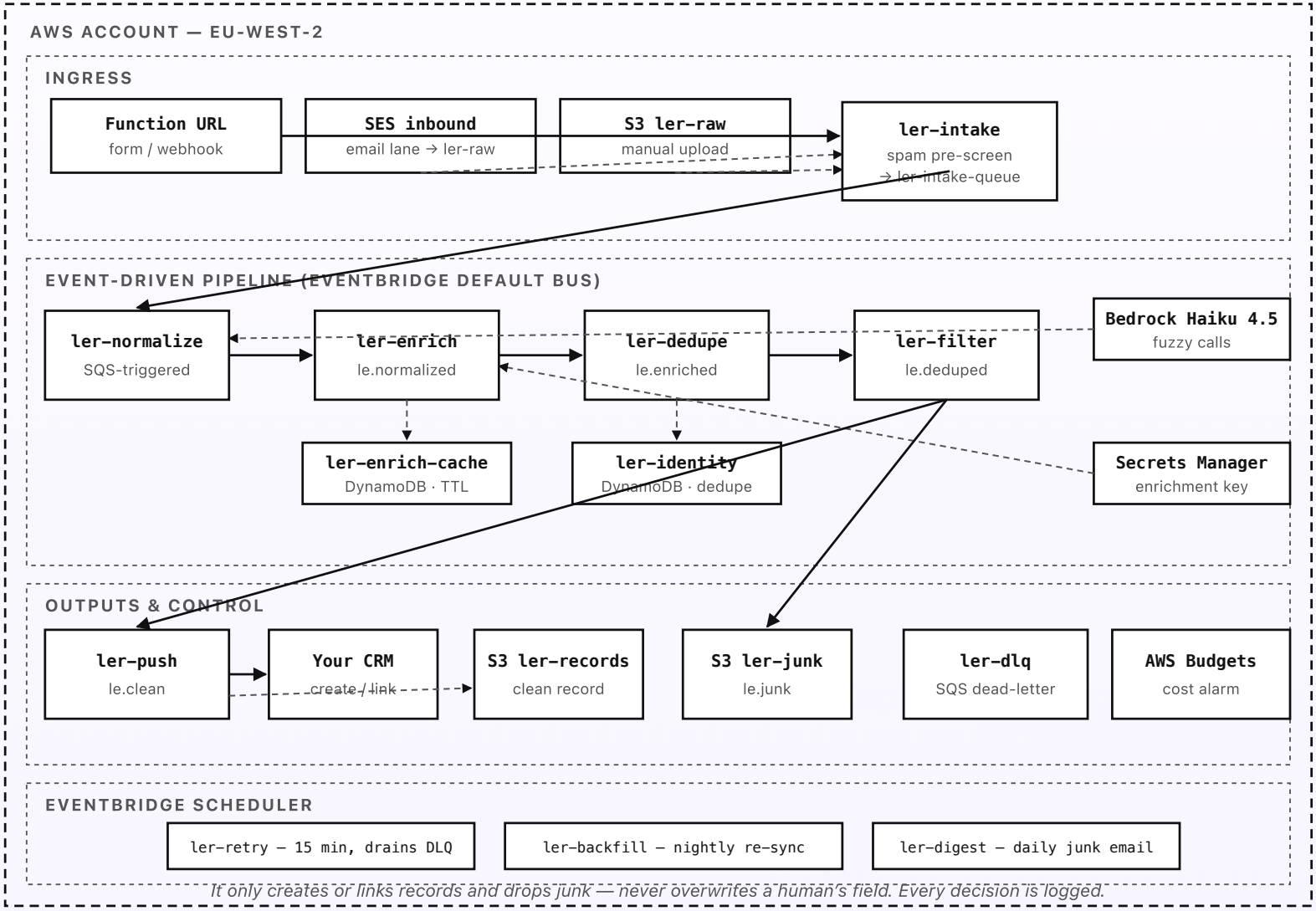

AWS ACCOUNT — EU-WEST-2
INGRESS
Function URL
form / webhook
SES inbound
email lane → ler-raw
S3 ler-raw
manual upload
ler-intake
spam pre-screen
→ ler-intake-queue
EVENT-DRIVEN PIPELINE (EVENTBRIDGE DEFAULT BUS)
ler-normalize
SQS-triggered
ler-enrich
le.normalized
ler-dedupe
le.enriched
ler-filter
le.deduped
Bedrock Haiku 4.5
fuzzy calls
ler-enrich-cache
DynamoDB · TTL
ler-identity
DynamoDB · dedupe
Secrets Manager
enrichment key
OUTPUTS & CONTROL
ler-push
le.clean
Your CRM
create / link
S3 ler-records
clean record
S3 ler-junk
le.junk
ler-dlq
SQS dead-letter
AWS Budgets
cost alarm
EVENTBRIDGE SCHEDULER
ler-retry — 15 min, drains DLQ
ler-backfill — nightly re-sync
ler-digest — daily junk email
It only creates or links records and drops junk — never overwrites a human's field. Every decision is logged.

Fig 7. The full topology: three ingress lanes, a five-stage chain over the EventBridge default bus, two outputs, a dead-letter queue, and three scheduled jobs — all in eu-west-2.

Lambdas

All Python 3.14 on arm64, 256–512 MB, with a dead-letter queue (`ler-dlq`) on every asynchronous path.

- **`ler-intake`** — the front door. Triggered three ways: a **Lambda Function URL** (IAM-auth or a shared secret) for the form/webhook lane, an **S3 PUT** on `ler-raw` for the manual-upload and SES-inbound lanes. Writes the raw payload to `ler-raw` (versioned), runs the cheap spam pre-screen (honeypot, disposable domain, timing), drops obvious bot junk straight to `ler-junk`, and enqueues survivors to the SQS queue `ler-intake-queue`. Memory: 256 MB.
- **`ler-normalize`** — consumes `ler-intake-queue` (batch size 1). Deterministic field cleanup (email, phone → E.164, name casing, country); calls Bedrock Haiku 4.5 only for flagged-ambiguous fields, bounding the result against ISO/reconstruction rules. Writes the clean record and emits `le.normalized` to the EventBridge default bus. Memory: 512 MB.
- **`ler-enrich`** — EventBridge rule on `le.normalized`. Skips personal/free domains; reads/writes `ler-enrich-cache`; on a miss, fetches the firmographic key from Secrets Manager and calls the provider, writing the result back to the cache with a TTL. Tags each field with source + date. Emits `le.enriched` (or marks `enrichment_pending` on provider failure and still emits). Memory: 256 MB. Timeout: 30 s.

- **`ler-dedupe`** — EventBridge rule on `le.enriched`. Exact-key lookup against `ler-identity`; deterministic fuzzy scoring for near-matches; Bedrock Haiku 4.5 only for the ambiguous band. Emits `le.deduped` carrying the verdict (`new` or `link:<crm_id>`). Memory: 256 MB.
- **`ler-filter`** — EventBridge rule on `le.deduped`. Weighs deterministic junk signals; Bedrock Haiku 4.5 only for borderline cases. Emits `le.clean` or `le.junk`. Memory: 256 MB.
- **`ler-push`** — EventBridge rule on `le.clean`. Does one final authoritative duplicate check against the live CRM (backstop to the index), then creates or links the contact via the CRM API (key in Secrets Manager), filling only empty fields — never overwriting. Writes the clean record to `ler-records` and updates `ler-identity`. Memory: 256 MB. Timeout: 30 s.
- **`ler-junk-writer`** — EventBridge rule on `le.junk`. Appends the dropped record and its reason to `ler-junk`. Memory: 128 MB.

EventBridge Scheduler

- **`ler-retry`** — every 15 minutes. Re-drives messages from `ler-dlq` and retries any lead marked `enrichment_pending` from a transient provider failure.
- **`ler-backfill`** — nightly (03:00). Re-enriches still-pending leads once the provider has recovered, and re-syncs `ler-identity` from a fresh CRM export so the index doesn't drift from the source of truth.
- **`ler-digest`** — daily (08:00). Summarises the previous day's `ler-junk` drops and emails the digest via SES so a human watches the boundary.

DynamoDB

- `ler-identity` (on-demand) — the dedupe/identity index. *Partition key:* `id_key` (string) — a typed, normalised key such as `EMAIL#jane.doe@acme-widgets.co.uk`, `PHONE#+447700900123`, or `FP#janedoe|acmewidgets`. *Attributes:* `crm_id`, `person_name`, `company`, `source`, `updated_at`. A global secondary index on `crm_id` lets the backfill rewrite every key for a person in one pass. Multiple `id_key` rows can point at the same `crm_id` — that's how one person is reachable by email, phone, or fingerprint.
- `ler-enrich-cache` (on-demand) — per-domain firmographic cache. *Partition key:* `domain` (string). *Attributes:* `profile` (JSON), `fetched_at`, `ttl` (epoch seconds; DynamoDB TTL enabled, ~90 days). A miss or an expired item triggers a fresh provider call.

S3 & SES

- **S3** `ler-raw` — raw inbound payloads (form JSON, SES MIME, uploaded CSVs). Versioning on; the immutable original behind every lead.
- **S3** `ler-records` — the clean, enriched, deduped record written for each lead that reached the CRM.
- **S3** `ler-junk` — the junk log: every dropped lead with its reason. The replay source if the filter ever drops something real.
- **SES inbound** — rule set for `hello@your-company.com`, action S3 PUT to `ler-raw`, which triggers `ler-intake`.
- **SES outbound** — the daily junk digest and ad-hoc ops alerts (provider down, DLQ backing up).

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0`, invoked via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Three callsites: `ler-normalize` (fuzzy field resolution), `ler-dedupe` (ambiguous same-person judgement), and `ler-filter` (borderline keep-or-drop). Each call is narrow, structured, and bounded by deterministic rules afterwards.
- **Heavier model.** `anthropic.claude-sonnet-4-6-...` is configured but off by default — available behind a per-callsite flag if a particular dedupe or junk decision ever needs more judgement than Haiku gives. The match and filter logic never *depends* on a model; the model only ever breaks a tie.

IAM (least privilege per function)

- **`ler-intake` role:** `s3:PutObject` / `GetObject` on `ler-raw` and `ler-junk`; `sqs:SendMessage` on `ler-intake-queue`. *No* Bedrock, *no* DynamoDB.
- **`ler-normalize` role:** `sqs:ReceiveMessage` / `DeleteMessage` on `ler-intake-queue`; `s3:GetObject` / `PutObject` on `ler-raw` / `ler-records`; `bedrock:InvokeModel` on the Haiku ARN; `events:PutEvents`.
- **`ler-enrich` role:** `dynamodb:GetItem` / `PutItem` on `ler-enrich-cache`; `secretsmanager:GetSecretValue` on the enrichment-key secret only; `events:PutEvents`. *No* write access to `ler-identity`.
- **`ler-dedupe` role:** `dynamodb:Query` / `GetItem` on `ler-identity` (read); `bedrock:InvokeModel` on the Haiku ARN; `events:PutEvents`. *No* CRM credentials.

- **`ler-filter` role:** `bedrock:InvokeModel` on the Haiku ARN; `events:PutEvents`. *No* data-store writes.
- **`ler-push` role:** `secretsmanager:GetSecretValue` on the CRM-key secret; `dynamodb:PutItem` / `UpdateItem` on `ler-identity`; `s3:PutObject` on `ler-records`. The only role that can write the identity index or talk to the CRM.

Region & operations

- **Region.** Everything in `eu-west-2` (London). The Bedrock *Global* cross-Region inference profile routes the model call for capacity, but no data store, queue, or bucket leaves the region.
- **CloudWatch Logs.** Every Lambda logs at 7-day retention — enough to debug a bad day, short enough to stay free-tier-adjacent.
- **DLQ.** `ler-dlq` (SQS) catches any event that fails twice; `ler-retry` re-drives it and alerts via SES if it backs up.
- **AWS Budgets.** A monthly cost alarm (first two budgets free) on the account, posting to SNS, so a runaway loop or a provider price change is caught the same day.
- **No API Gateway, no NAT Gateway, no always-on compute.** Ingress is a Lambda Function URL plus SES and S3 events; everything else is event- or schedule-driven.