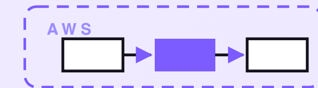
7-PART SERIES · FREE COMPANION



Translation relay

A serverless relay that lets a small team serve customers in any language. It detects the language of each incoming message, translates it to your team's language, and translates the staff reply back — keeping the conversation flowing both ways. It keeps names, numbers, and prices exact, and flags anything it's unsure about for a human. A staff member always writes the actual reply. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

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

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/translation-relay

CONTENTS

Translation relay

01 A translation relay on AWS for a few dollars a month

02 How a message gets its language detected

03 How a message gets translated for staff

04 How a reply gets translated back

05 How names and prices stay exact

06 What the translation relay costs

07 Engineering reference: the translation relay architecture

PART 1 OF 7

MAY 31, 2026 PART 1 OF 7 · TRANSLATION RELAY SERIES ~5 MIN READ

A translation relay on AWS for a few dollars a month

A small business sells to people who don’t all speak the same language. A customer in São Paulo writes in Portuguese. A supplier in Osaka replies in Japanese. A new buyer in Berlin asks a question in German. Your support team speaks English and maybe one other language between them. Hiring a fluent speaker for every market isn’t realistic, and pasting messages into a free translate box is slow, leaks customer data, and quietly mangles the one figure that mattered. This post walks through the design of a small relay that translates each message in, lets a human write the reply, and translates that reply back — while keeping every name and price exact.

KEY TAKEAWAYS

- Messages arrive two ways: by email and through a small web widget on your site.
- Three pieces inside AWS: detect the language, translate in for staff, translate the staff reply back.
- A person always writes the reply. The relay never sends a machine-written message to a customer.
- Names, numbers, and prices are locked before translation and put back after, so they stay exact.
- Designed on AWS for about $3/month at typical small-business volume.

The whole system on one page

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

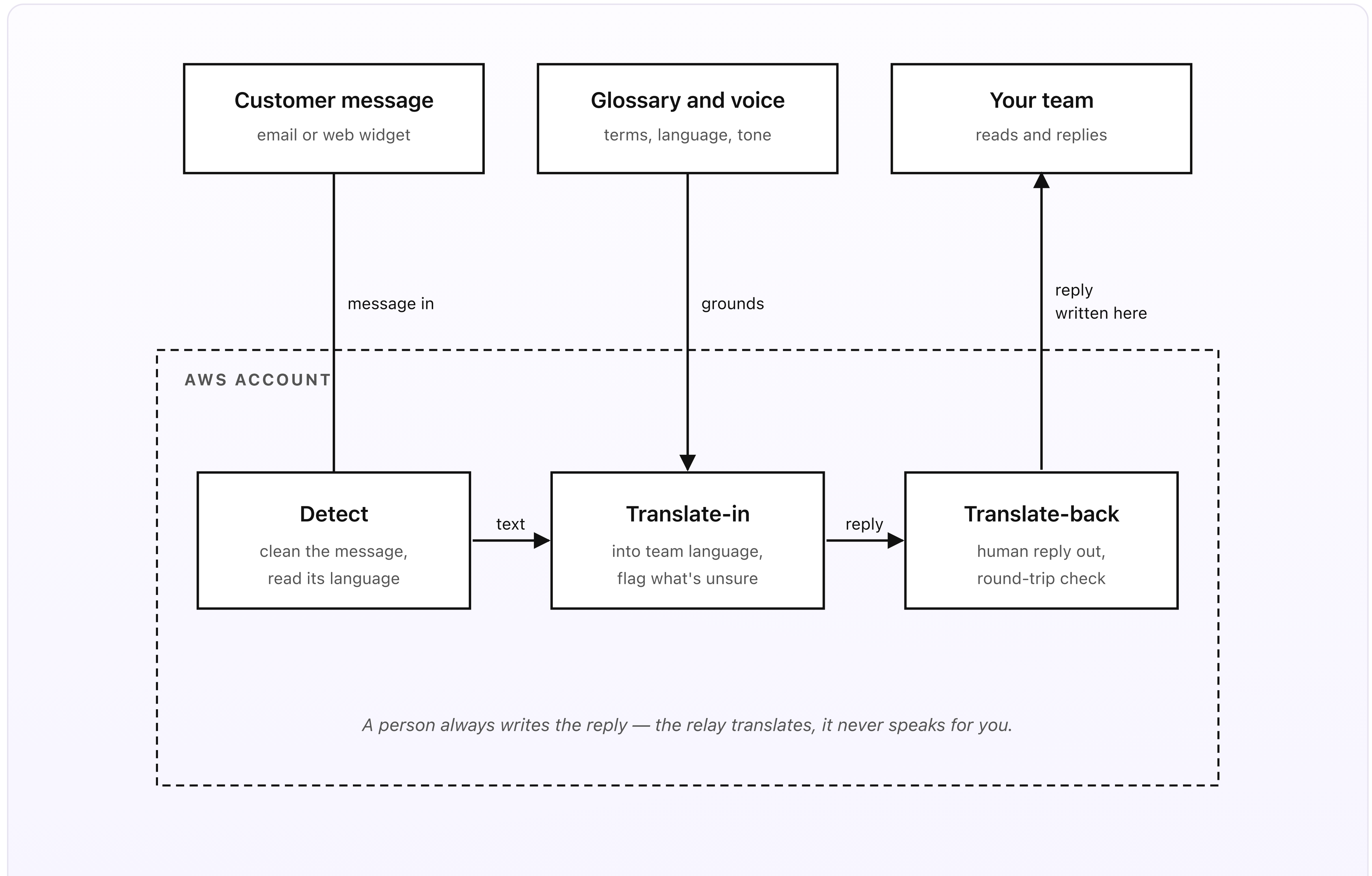

Customer message
email or web widget
Glossary and voice
terms, language, tone
Your team
reads and replies
message in
grounds
reply
written here
AWS ACCOUNT
Detect
clean the message,
read its language
text
Translate-in
into team language,
flag what's unsure
reply
Translate-back
human reply out,
round-trip check
A person always writes the reply — the relay translates, it never speaks for you.

Fig 1. Two inputs outside, three pieces inside AWS. Messages flow in by email or a web widget. Detect reads the language, Translate-in turns it into your team's language, and Translate-back carries the human-written reply out in the customer's language.

What you set up once (the outside)

- **Customer messages.** Two ways in. The first is email — you point your support address (or a copy of it) at the relay, and any message a customer sends lands in the system. The second is a small chat widget you drop on your website with one line of code; a visitor types in their own language and it goes straight to the relay. Either way, the customer writes in whatever language they want and never has to think about translation.
- **A glossary folder.** Two short things in a Drive folder. The *glossary* is a Google Sheet, one row per term that must never be translated or changed: your brand name, your product names, account or order IDs, and any phrase that has to land word-for-word. The *voice* note holds your team's working language (say, English) and a line or two on tone — warm and plain, or formal, or however your business sounds. Editing either one doesn't need a deploy; a small sync job picks up the change within fifteen minutes.
- **Your team.** The people who actually answer customers. They read every message in their own language with the original sitting right next to it, write the reply in their own language, and get a plain check of what the relay will send before they press send. They don't need to speak the customer's language — they just need to do their normal job.

What runs on every message (the inside)

- **Detect.** When a message arrives, the relay first cleans it — strips the email signature, the quoted history, the “sent from my phone” footer — so it’s looking at the actual new text. Then it reads which language that text is in. Most messages are easy. Short ones (“ok thanks”), mixed-language ones, and ones full of product codes are harder, so the detect step has a cheap first pass and a careful fallback. Part 2 covers this in detail.
- **Translate-in.** The relay translates the customer’s message into your team’s working language using Bedrock with Claude Haiku 4.5 — a small, fast, inexpensive model that handles almost everything. Before it translates, it locks every protected term from the glossary and every number and price behind a placeholder, so the model can’t touch them. The staff member sees the original and the translation side by side, with any passage the relay wasn’t sure about highlighted. If a passage is genuinely tricky — slang, sarcasm, a mixed-language sentence — it gets a second pass from the stronger Claude Sonnet 4.6 before a person sees it.
- **Translate-back.** The staff member writes the reply in their own language. The relay translates it back into the customer’s language, again locking the protected terms and figures first. Then it does one more thing: it translates that outgoing message *back* into the staff language and shows it next to what they wrote, so they can confirm the meaning carried before anything goes out. Nothing sends until a person presses the button. Then the reply leaves by email or appears in the web widget, in the customer’s language, looking like it was written by a fluent speaker.

In plain words

A customer in São Paulo emails your support address in Portuguese: "Meu pedido #4471 chegou com a tampa quebrada. Posso trocar?" The relay cleans the message, reads that it's Portuguese, locks the order number `#4471` behind a placeholder, and translates the rest into English for your agent Sam: "My order #4471 arrived with the lid broken. Can I exchange it?" Sam doesn't speak a word of Portuguese. He reads the English, checks the order, and writes back in English: "So sorry about that — yes, we'll send a replacement for order #4471 today, no charge." The relay translates that into Portuguese, keeps `#4471` exactly as written, and shows Sam a plain English read-back of the Portuguese so he can confirm it says what he meant. He presses send. The customer gets a fluent Portuguese reply in the same few minutes any other ticket would take, and never knows a translation happened.

The cost of running this is about $3 a month at SMB volume. The cost of *not* running it is the market you don't serve because nobody on the team speaks the language — or the refund you gave on the wrong order because a free translate box swapped a digit and nobody caught it.

DESIGN RULES THAT SHAPED EVERY DECISION

- A person always writes the reply. The relay translates both ways but never sends a machine-written message to a customer.
- Names, numbers, and prices are locked before translation and restored after. The machine can't change a figure.
- Every translation comes back with a confidence read. Low-confidence passages are shown to a human with the original.
- The cheap model handles almost everything; the stronger model runs only on the genuinely tricky passages.
- The glossary lives in Drive. Adding a protected term or changing tone doesn't need a deploy.
- Every turn keeps both the original and the translation. Audit a conversation later and you can read both sides.

Why this shape

Most small teams handle a foreign-language message in one of three ways: they paste it into a free translate box, they forward it to the one bilingual person on staff and wait, or they quietly ignore it. The translate box is fast but it leaks customer data into a third party, mangles the numbers that matter, and gives you a wooden reply that reads like a robot wrote it. The bilingual colleague is great until they're on holiday, or they leave, or the message is in a language nobody on the team speaks. And ignoring it is a lost customer who tells their friends.

The setup above keeps the human exactly where they add value — understanding the customer's problem and deciding what to do — and puts the machine exactly where it's reliable: turning words from one language into another, under guardrails, with the risky bits flagged. The agent never has to trust a translation blindly, because the original is always right there and the figures are never touched. The customer gets served in their own language, fast, by a team that doesn't speak it.

The next four posts walk through each piece in turn: how a message gets its language detected, how a message gets translated for staff, how a reply gets translated back, and how names and prices stay exact through both directions. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 31, 2026 PART 2 OF 7 · TRANSLATION RELAY SERIES ~4 MIN READ

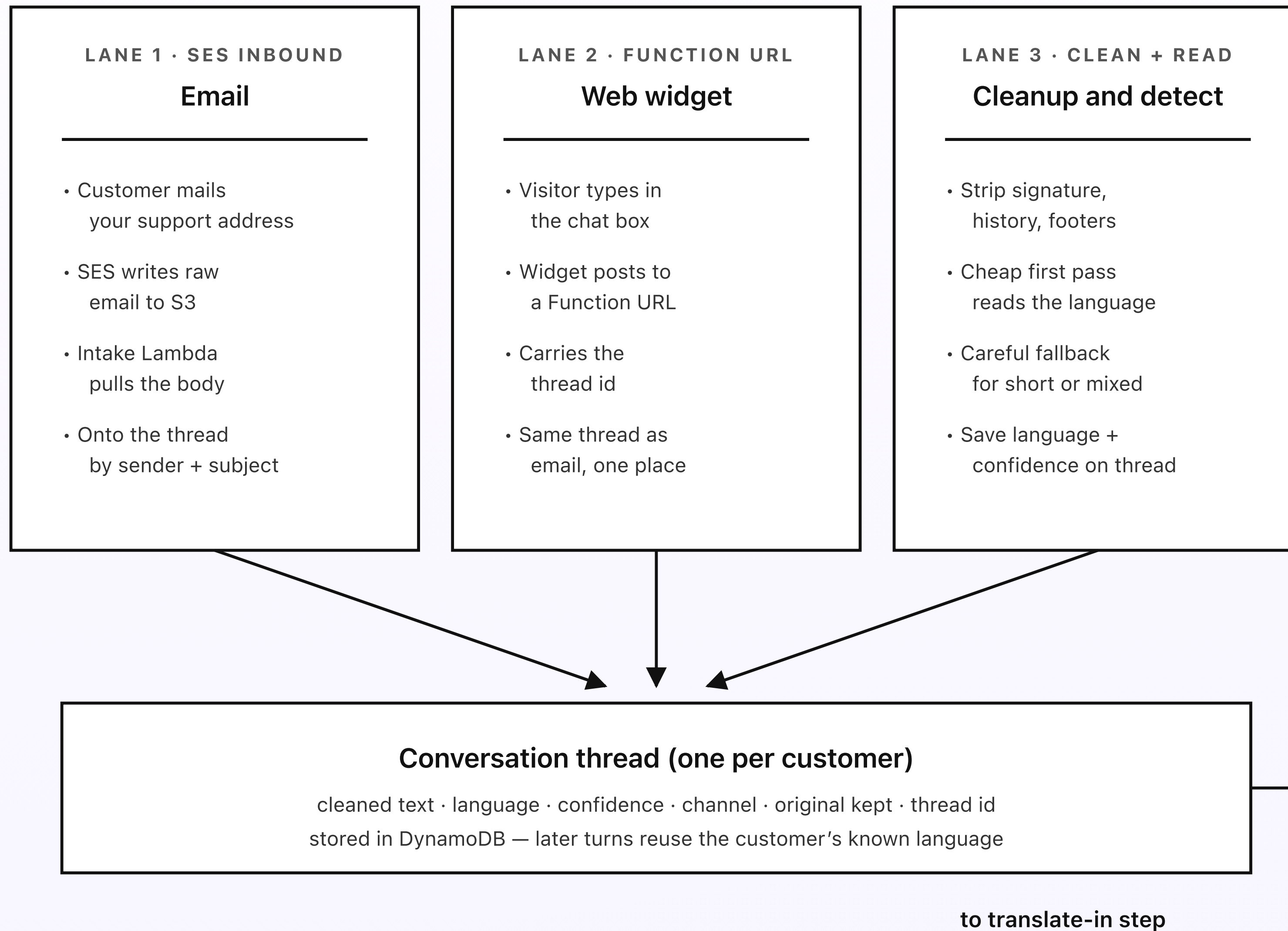
How a message gets its language detected

The relay can’t translate a message until it knows what language the message is in. That sounds trivial — and for a long, clearly-written paragraph it is. But real customer messages are short, full of order numbers, signed with a name in a third language, and sometimes written half in one language and half in another. This post walks through how a message gets in (two ways), how it gets cleaned down to the words that matter, and how the relay reads its language without guessing wrong on the hard ones.

KEY TAKEAWAYS

- Two ways in feed one thread: email through SES inbound and a web widget through a Function URL.
- Every message is cleaned first — signatures, quoted history, and footers are stripped before detection.
- A cheap first pass reads the language; a careful fallback handles short, mixed, or code-heavy notes.
- The detected language and a confidence read are saved on the thread so later turns can reuse them.
- If the relay still isn't sure, the message is marked "language unclear" for a person, never guessed silently.

Two ways in, one cleaned message



The relay never guesses silently — if the language is unclear, the message is flagged for a person.

Fig 2. Two inputs and a cleanup-and-detect step converge on one thread. Email and the web widget both land in the same place; the cleanup step strips noise; a cheap read with a careful fallback figures out the language and saves it on the thread.

Lane 1: email

The most common way in. You point your support address — or a forwarding copy of it — at the relay through Amazon SES. When a customer emails, SES writes the raw message to an S3 bucket, and that write triggers a small intake Lambda. The Lambda walks the email to its plain-text body (falling back to the HTML body, stripped to text, if there's no plain-text part), figures out which conversation it belongs to from the sender and subject, and drops the new message onto that thread. If it's a brand-new sender, it starts a fresh thread.

Email is messy. A reply often carries the entire history of the conversation quoted below it, plus a signature block, plus a mobile footer. Translating all of that would be wasteful and confusing, so the very next step is cleanup — covered in Lane 3.

Lane 2: the web widget

Some customers would rather not email at all. A small chat widget — one line of script on your site — lets a visitor type a question in their own language and get help right there. The widget posts the text to a Lambda Function URL (a plain web address that runs a Lambda, with no API Gateway in front of it), along with a thread id so the back-and-forth stays together. From there it joins the exact same flow as an email: same cleanup, same detection, same thread store. The customer doesn't care which pipe their message took, and neither does the rest of the system.

The widget text is usually cleaner than email — no signature, no quoted history — but it's often shorter, which makes language detection harder, not easier. A two-word message gives the detector very little to go on.

Lane 3: cleanup, then read the language

Whichever way the message arrived, the relay now has to find the words that matter and read their language. Cleanup comes first. A small deterministic step — plain Python, no model — strips the email signature, the quoted reply history (the lines that start with ">" or sit under "On Tuesday, X wrote:"), and footers like "Sent from my iPhone." What's left is the new text the customer actually wrote this time.

Then detection runs in two passes. The first pass is cheap: for a message of reasonable length, a fast script-and-statistics check reads the language confidently and for free. Most messages stop here. The second pass is the careful fallback, and it exists for the hard cases — a three-word reply, a message that mixes two languages, a note that's mostly product codes and prices with only a few real words. For those, the relay asks Bedrock with Claude Haiku 4.5 to read the language, since a model handles short and mixed text far better than a statistics check. Each pass returns both a language and a confidence score, and both go onto the thread.

There's one more rule, and it's the important one: the relay never guesses silently. If even the careful fallback comes back unsure — say, a message that's genuinely half English and half Spanish, or one too short to call — the message is marked "language unclear" and shown to a person, with the original text, before anything gets translated. A wrong language guess at this step poisons every step after it, so it's the one place the system would rather stop and ask.

Why detection is its own step

It would be tempting to fold detection into the translate step — just hand the message to the model and say “translate this to English, whatever it is.” That works most of the time, but it hides the one fact you most want recorded: which language the customer is writing in. The relay needs that fact to translate the reply *back* later, to pick the right tone, and to reuse it on the customer’s next message so a one-word follow-up like “sí” doesn’t get mis-read. Saving the detected language on the thread, with its confidence, makes every later step cheaper and steadier.

Next post: how the cleaned message gets translated into your team’s language, shown side by side with the original, with anything the relay wasn’t sure about highlighted for a human to check.

PART 3 OF 7

MAY 31, 2026 PART 3 OF 7 · TRANSLATION RELAY SERIES ~5 MIN READ

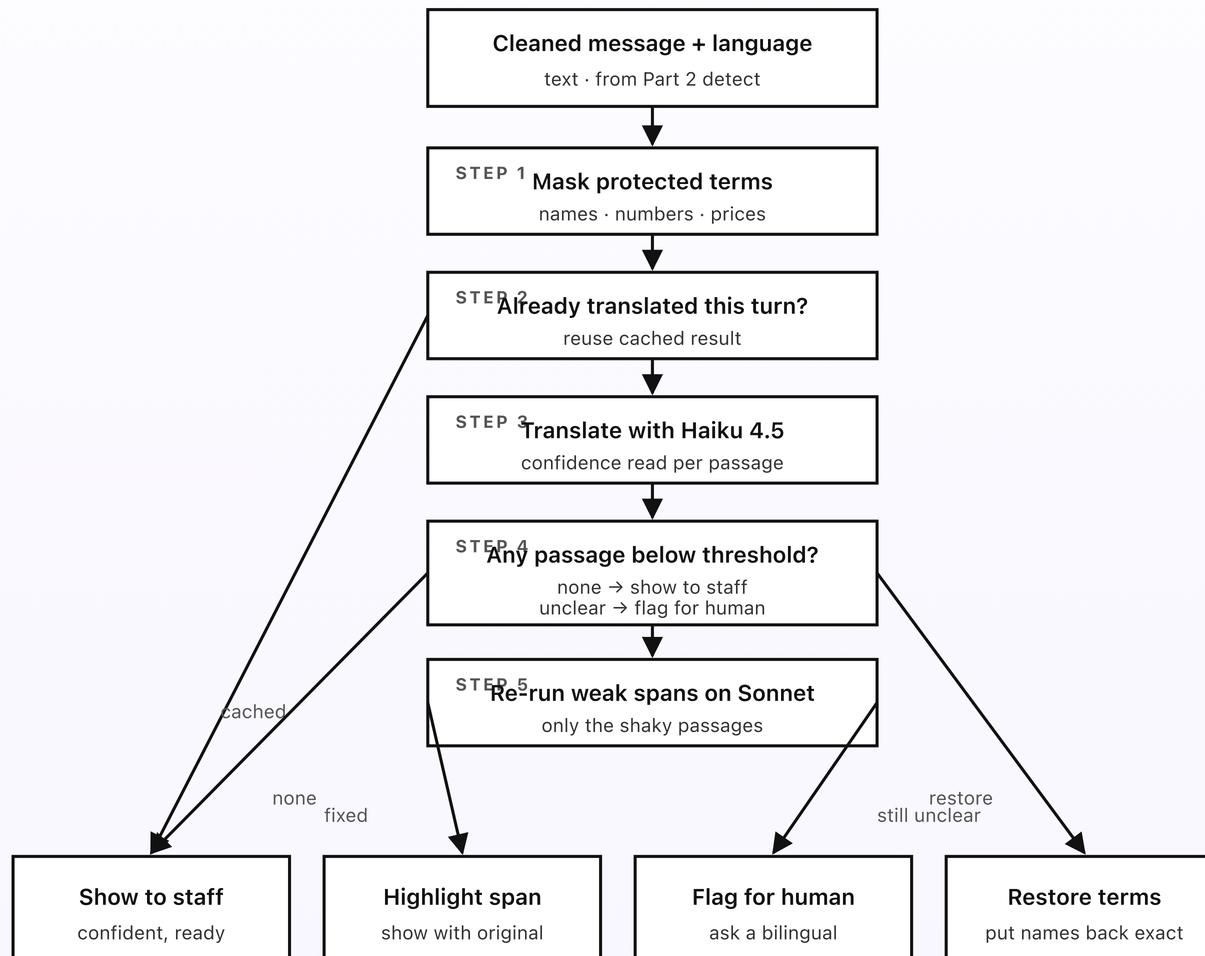
How a message gets translated for staff

The relay knows the language now. Next it turns the customer's message into your team's working language so a staff member who doesn't speak a word of it can read it and act. The goal isn't a perfect literary translation — it's a faithful, plain one, with the bits the machine wasn't sure about marked so a person can judge them. Most messages go through the cheap model and stop. The hard passages get a second, stronger pass before anyone sees them.

KEY TAKEAWAYS

- Protected terms, numbers, and prices are masked before translation, then restored after (covered in Part 5).
- The everyday translation runs on Claude Haiku 4.5 — small, fast, and cheap.
- The model returns a confidence read per passage, not one score for the whole message.
- Low-confidence passages get a second pass from the stronger Claude Sonnet 4.6.
- Staff see the original and the translation side by side, with unsure passages highlighted.

The translate-in flow, per message



The cheap model handles almost everything — the stronger one only touches what needs it.

Fig 3. The translate-in flow, per message. Mask the protected terms, translate with the cheap model, read the confidence per passage, re-run only the shaky passages on the stronger model, then restore the exact names and figures and show staff the result.

Mask first, translate second

Before a single word goes to a model, the relay locks down anything that must not change. Your brand name, your product names, account and order IDs from the glossary, and every number and price in the message are swapped for neutral placeholders — little tokens like `[[TERM_1]]` and `[[NUM_3]]`. The model translates the words around them but literally cannot touch what's inside, because it never sees the real values. After translation, the placeholders are swapped back to the exact originals. This is the single most important guardrail in the whole system, and Part 5 is devoted entirely to it. For now: by the time the message reaches the model, the risky parts are already safe.

The cheap pass does most of the work

The everyday translation runs on Bedrock with Claude Haiku 4.5 — a small, fast model that costs a fraction of a cent per message. The prompt is short and strict: "Translate the message between the placeholders into *{team language}*. Keep the placeholders exactly as they are. Stay faithful and plain; don't add or drop meaning. Return the translation, and for each sentence return a confidence score from 0 to 1."

That per-passage confidence is the part that earns its keep. A whole-message score would tell you the message is “87% fine,” which is useless — the 13% that’s wrong might be the one sentence that decides the refund. Scoring each sentence means the relay can be confident about the easy parts and honest about the one clause it struggled with, and a staff member can see exactly where to look.

The careful pass, only where it’s needed

If every passage clears the confidence threshold, the message is done — the cheap model handled it, no further calls. If one or two passages come back shaky, the relay re-runs *only those passages* on the stronger Claude Sonnet 4.6, with the surrounding sentences as context. This is the heart of keeping the bill small: the expensive model never translates a whole message, only the handful of clauses that genuinely need a better reader. Slang, sarcasm, a half-sentence that switches language mid-way, an idiom that doesn’t map cleanly — these are where Sonnet earns its higher price.

After the second pass, a passage either clears (it’s now confident, and it’s shown normally) or it’s still unclear. A still-unclear passage isn’t buried — it’s highlighted in the staff view with the original text right beside it, so the agent can see “the relay isn’t sure what this clause means.” And if the whole message is genuinely impenetrable — truly mixed languages, or a cultural reference with no equivalent — it’s flagged for a bilingual teammate before a reply is written. The relay would rather admit it’s stuck than hand the agent a confident-looking sentence that’s quietly wrong.

What the staff member actually sees

The agent opens the thread and sees two columns. On the left, the customer’s original message in their own language. On the right, the translation in the team’s language. Anything the relay wasn’t sure about is highlighted, and hovering shows the original snippet for that passage. The protected terms — order numbers, product names — appear identically on both sides, because they were never translated. The agent reads the right column, glances at the highlights, and now understands the customer’s problem as well as if it had come in their own language. They don’t have to trust the machine blindly, because the original is always one glance away.

Next post: the agent writes a reply in their own language, and the relay carries it back out — translating it into the customer’s language, showing a round-trip check, and waiting for a person to press send.

PART 4 OF 7

MAY 31, 2026 PART 4 OF 7 · TRANSLATION RELAY SERIES ~5 MIN READ

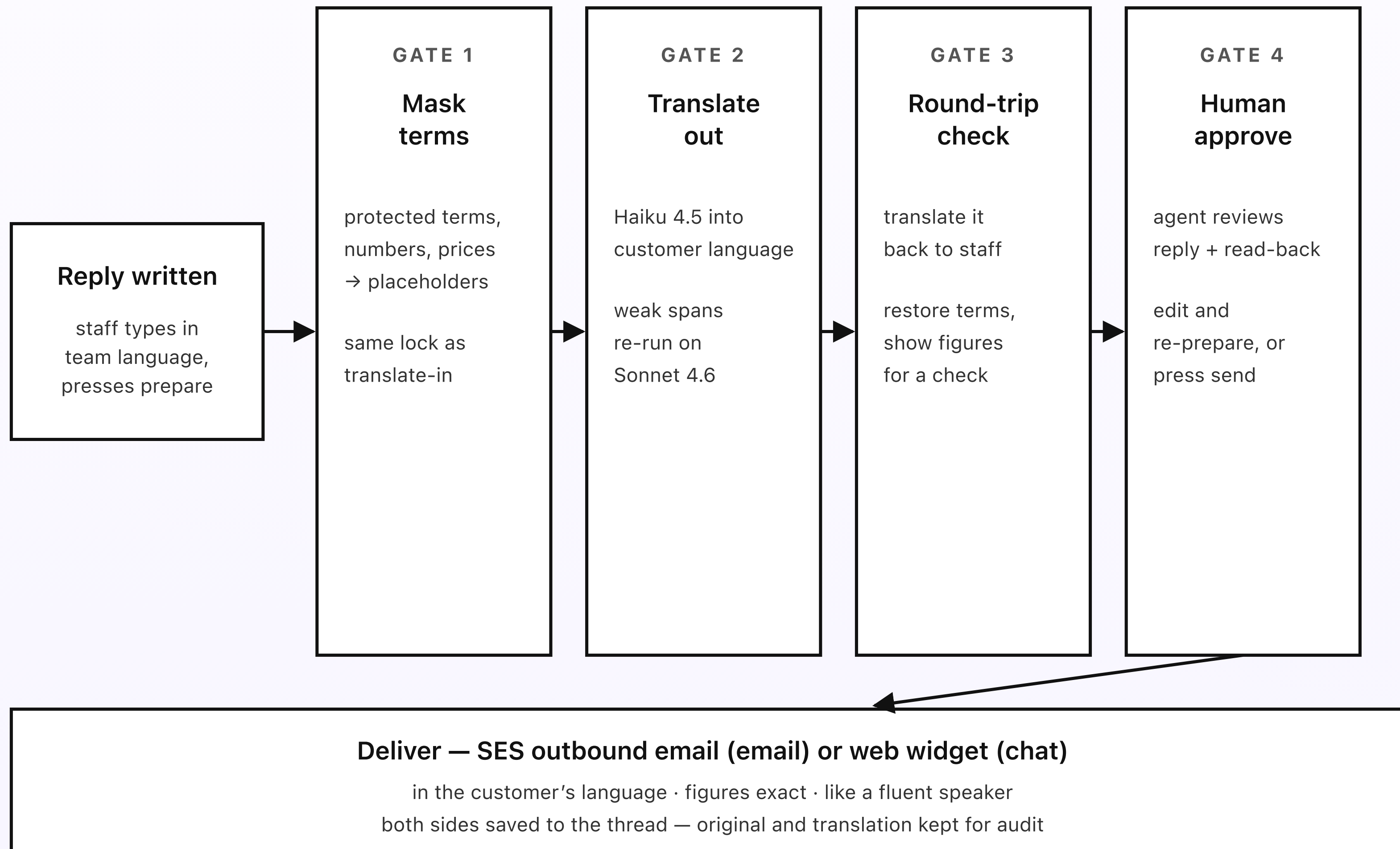
How a reply gets translated back

The agent has read the translated message and written a reply — in their own language, the way they'd write any reply. Now the relay has to carry that reply back out in the customer's language without changing what the agent meant, without touching a figure, and without ever sending on its own. Four small gates sit between the agent pressing "prepare" and the reply actually leaving. The last gate is a human pressing send.

KEY TAKEAWAYS

- The staff member writes the reply in their own language; the relay never writes the reply.
- Protected terms, numbers, and prices are masked before the reply is translated, then restored after.
- The relay shows a round-trip check — the outgoing reply translated back to the staff language.
- Nothing sends until a person presses send. The relay prepares; the human approves.
- The reply leaves by email (SES outbound) or appears in the web widget, in the customer’s language.

Four gates on every outgoing reply



The relay prepares the reply and shows its work — a person always presses send.

Fig 4. Four gates between the written reply and the sent reply. Mask the terms, translate out, show a round-trip check back to the staff language, and wait for a human to approve. Then deliver by email or in the widget, in the customer's language.

Gate 1: mask the same things, in the same way

The outgoing direction uses the exact same locking step as the incoming one. The agent's reply almost always repeats the figures that matter — "we'll refund the $42.00 to your account" or "your order #4471 ships today." Those get swapped for placeholders before translation and restored after, so the translation can rephrase the sentence around them but can never turn `$42.00` into `$420` or drop a digit from the order number. Part 5 covers the masking in full; here it's enough to know the reply is protected the same way the message was.

Gate 2: translate the reply out

With the figures locked, the masked reply is translated into the customer's language — the one saved on the thread back in Part 2. It's the same machinery as translate-in, just pointed the other way: Claude Haiku 4.5 does the everyday work and returns a confidence read per sentence, and any sentence that comes back shaky is re-run on Claude Sonnet 4.6. A reply is usually cleaner to translate than an incoming message — the agent wrote it plainly, on purpose — so the stronger model fires even less often here than on the way in.

Gate 3: the round-trip check

This is the gate that makes the agent comfortable sending something in a language they can't read. After translating the reply into the customer's language, the relay translates that result *back* into the staff language and shows it next to what the agent originally wrote. If the two read the same, the meaning carried. If the read-back says something subtly different — “we might refund” where the agent wrote “we will refund” — the agent catches it before the customer ever sees it. The restored figures are shown too, side by side with the originals, so the agent can confirm at a glance that `$42.00` is still `$42.00`.

A round-trip translation isn't a perfect proof — some meaning can survive the trip out and back even if the outgoing text is slightly off — but in practice it catches the mistakes that matter: flipped negatives, changed amounts, a softened commitment, a missing “not.” It turns “trust the machine” into “read this plain check,” which is a much easier thing to ask of a busy agent.

Gate 4: a human presses send

Nothing has left the building yet. The agent now sees three things together: the customer-language reply that's ready to go, the plain read-back of what it means, and the restored figures. They can press send, or they can edit their original reply and re-prepare — maybe they'll simplify a sentence the round-trip showed was getting mangled, or rephrase an idiom that doesn't travel. Only when a person presses send does the reply actually go out.

This is the core rule of the whole system, and it's why the relay is safe to put in front of real customers: it never sends a machine-written or machine-approved message. It does the translation, it shows its work, and it hands the decision to a

person. The customer gets a fluent reply in their own language; the business keeps a human accountable for every word that goes out under its name.

Then it ships, and the thread remembers

Once approved, an email thread goes out through SES outbound and a widget thread appears right in the customer's chat window — in the customer's language, with the figures exact, reading like a fluent speaker wrote it. Both the original (what the agent typed) and the translation (what the customer received) are saved on the thread, so a bilingual colleague who picks up the conversation later can read exactly what was said in both languages, and the audit trail shows every machine translation and every human edit.

Next post: the guardrail that runs underneath both directions — how names, numbers, and prices are locked before translation and put back after, so the machine can never quietly change a figure or rename your product.

PART 5 OF 7

MAY 31, 2026 PART 5 OF 7 · TRANSLATION RELAY SERIES ~5 MIN READ

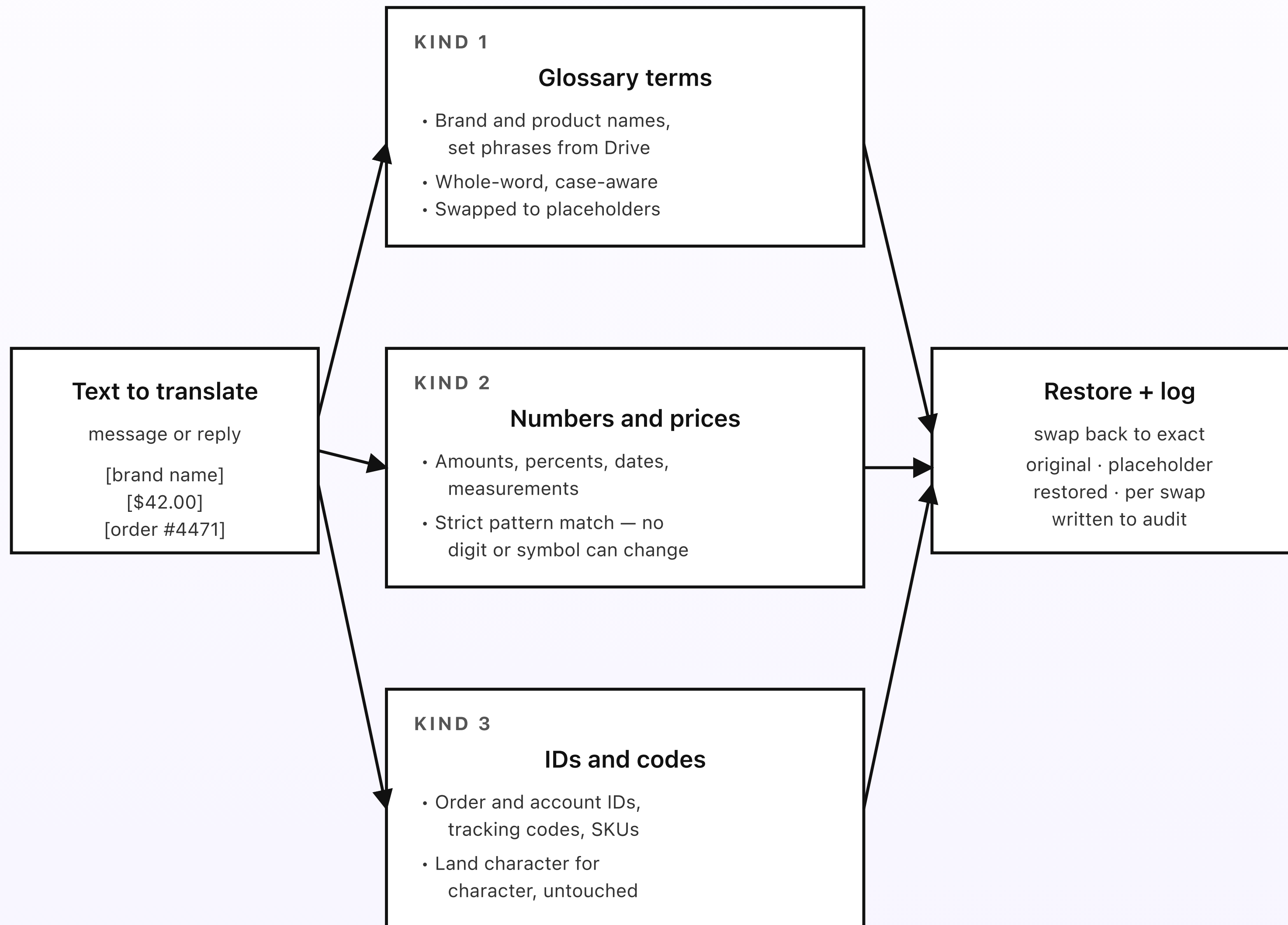
How names and prices stay exact

A translation that’s 99% right but turns “$42.00” into “$420” is worse than no translation at all. The same goes for an order number with a flipped digit, or a product name the model helpfully “translated” into a generic word. These are the mistakes that cost real money and real trust, and a model on its own will occasionally make every one of them. So the relay never lets the model near them. This post is about the single guardrail that runs underneath both directions: lock the exact bits, translate the rest, put the exact bits back.

KEY TAKEAWAYS

- Three kinds of content are protected: glossary terms, numbers and prices, and IDs and codes.
- Each protected bit is swapped for a placeholder before translation, then restored to the exact original after.
- The model translates the words around the placeholders but never sees the real names or figures.
- Every swap is logged, so you can prove a figure was never altered in translation.
- This same lock runs on the way in and the way out — both directions, identical rules.

Three kinds of protected content



The model translates the words, never the figures — and every swap is logged.

Fig 5. Three kinds of protected content, locked before translation and restored after. Glossary terms, numbers and prices, and IDs and codes are each swapped for placeholders; the model translates the words around them; the restore step puts the exact originals back and logs every swap.

Kind 1: glossary terms (the things only you know)

Some words must never be translated because they're names, not words. Your company is called "Northwind," not "north wind." Your product is "FreshKeep," not "keeps fresh." A model with no context will sometimes translate these into their literal meaning, which reads as wrong to anyone who knows your brand. The glossary — the Google Sheet from Part 1 — lists every one of these, and the matcher swaps each for a placeholder before translation.

The matching is careful on purpose. It's case-aware and matches whole words only, so a glossary term won't accidentally get matched inside a longer, unrelated word. Each entry can also carry a note — "FreshKeep is a product line, never translate" — that's shown to the agent on hover, so the human review has context too. Because the glossary lives in Drive, adding a new product name the day you launch it is a thirty-second edit, no deploy required; the sync job picks it up within fifteen minutes.

Kind 2: numbers and prices (the things that cost money to get wrong)

Every number in a message is found by a strict pattern and locked before translation: currency amounts ("$42.00", "€39,90"), plain numbers, percentages,

dates, and measurements. The relay doesn't trust the model to carry a figure faithfully, because it occasionally won't — it'll round, it'll re-format, it'll convert "1,000" (one thousand) into "1.000" in a language where the comma and period swap roles, and the meaning quietly changes. By locking the exact characters and restoring them verbatim, the figure that leaves is character-for-character the figure that arrived.

There's one deliberate subtlety: the relay does *not* auto-convert currencies or units. If a customer writes "€40" the agent sees "€40," not a guessed dollar amount — converting money is a business decision with a live exchange rate, not a translation, and silently doing it would be exactly the kind of "helpful" error this guardrail exists to prevent. If a conversion is needed, the agent does it on purpose, and the relay protects whatever figure they write.

Kind 3: IDs and codes (the things that must land exactly)

Order numbers, account IDs, tracking codes, SKUs, reference numbers — anything that's an identifier rather than a word — must land character for character or it's useless. A tracking code with one wrong character points at nothing. An order number with a flipped digit pulls up someone else's order. These are matched by shape (mixes of letters and digits, hash-prefixed numbers, known ID formats) and locked the same way. They're the lowest-risk to detect and the highest-cost to get wrong, which makes them the easiest call in the whole system: never let the model touch them.

Restore, then prove it

After the model returns its translation — with the placeholders still sitting in place, untouched — the restore step swaps each placeholder back to its exact original value. Then it does the part that makes the whole thing trustworthy: it writes every swap to the audit log. For each protected bit, the log records the original value, the placeholder it was given, and the value it was restored to. Because the restored value is always the original value, you can prove — not hope, prove — that no figure, name, or code was altered in translation. If a customer ever disputes “you quoted me $42, the email said $52,” the log settles it in seconds.

This same lock-and-restore runs identically on the way in (translating the customer’s message for staff) and on the way out (translating the staff reply for the customer). One guardrail, both directions, no exceptions. It’s the quiet reason the relay is safe to put in front of real money: the model makes the language fluent, and a deterministic step — not the model — keeps the facts exact.

Next post: the cost breakdown. The whole relay runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the cheap model carries almost all of it.

PART 6 OF 7

MAY 31, 2026 PART 6 OF 7 · TRANSLATION RELAY SERIES ~3 MIN READ

What the translation relay costs

The relay is cheap because it does the expensive thing as rarely as possible. Every message gets one or two small model calls; the stronger, pricier model only touches the handful of passages that need it. There's no always-on compute, no API Gateway, no NAT Gateway. At typical SMB volume the bill is a few dollars a month, and the fixed cost is essentially zero.

KEY TAKEAWAYS

- Around $3/month at typical SMB volume (about 600 messages a month, both directions).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- Most of the bill is the Bedrock translate calls — and the cheap model carries almost all of it.
- The stronger Claude Sonnet 4.6 only fires on the tricky passages, so it stays a small sliver.
- At 2,000 messages the bill is around $9. At 5,000 it's around $20.

Cost at three volumes

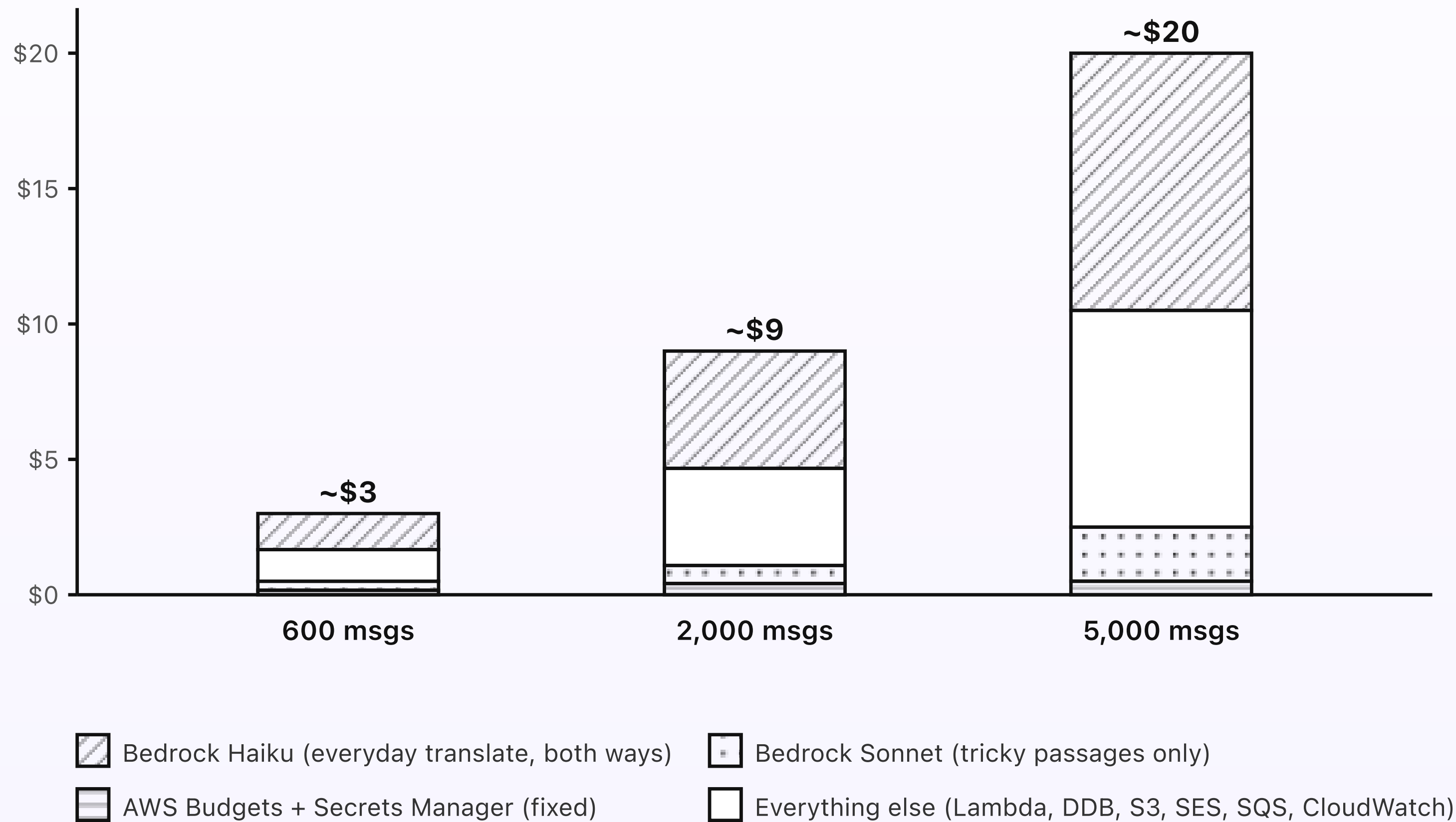


Fig 6. Monthly cost at three message volumes. Bedrock Haiku — the everyday translate calls in both directions — is the dominant slice. Bedrock Sonnet stays a sliver because it only fires on the passages the cheap model flags as unsure.

Where the dollars actually go

Bedrock Haiku (the bulk). Every message gets translated in, and every reply gets translated out, on Claude Haiku 4.5. Each call is a few hundred input tokens and a few hundred output tokens — a fraction of a cent. Detection's careful fallback (for short or mixed notes) and the round-trip check add a couple more small Haiku calls per conversation. Multiply by a few hundred messages a month and it's a couple of dollars. This is the part that scales with how busy you are, and it scales gently.

Bedrock Sonnet (only the tricky passages). The stronger model never translates a whole message — only the handful of clauses the cheap model flagged as shaky. At typical volume that's a small share of passages, so Sonnet stays a sliver. A business with a lot of slangy, mixed-language messages will see this slice grow, but it's still bounded by "only the hard bits, only when needed."

Lambda runtime. Short functions: intake, detect, translate-in, translate-back, send, and the drive-sync job every fifteen minutes. Each runs for well under a second on arm64. Pennies a month at all three volumes.

DynamoDB on-demand. The conversation threads, the swap log, and the audit trail. A few small reads and writes per message. Pennies.

S3 + storage. The mirrored glossary, the voice note, and the raw inbound email MIME. A few hundred KB at SMB volume. Effectively free.

SES. Inbound for the email lane: $0.10 per thousand received messages. Outbound for the replies: $0.10 per thousand sent. Negligible at this scale.

SQS. The queue that sits between detect and translate so a burst of messages can't overwhelm the model calls. A million requests a month is under a dollar; you'll use a tiny fraction of that.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the web widget and the send-reply endpoint.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Everything runs only when a message moves through.
- **A Knowledge Base.** The relay translates; it doesn't answer from documents. No embeddings, no vector store, no S3 Vectors.
- **A second model on the easy path.** The cheap model handles the everyday work; the stronger one is reserved for the passages that earn it.

How the cost scales

Bedrock Haiku grows roughly linearly with message volume, because every message is translated both ways. Sonnet grows too, but only with the share of tricky passages, which is small and fairly stable. Lambda, DynamoDB, and SES all grow linearly and stay tiny. So the bill at 10,000 messages a month is around $40, and at 20,000 it's around $80 — still far cheaper than one part-time bilingual hire, and it covers every language at once. Past those volumes you'd start caching common phrases and batching the round-trip checks, but those are optimizations for a busy support desk, not redesigns.

Set an AWS Budgets alarm at $25/month so anything unusual pages you before the bill matters. The relay's normal-volume cost stays well under that ceiling.

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

PART 7 OF 7

MAY 31, 2026 PART 7 OF 7 · TRANSLATION RELAY SERIES ~8 MIN READ

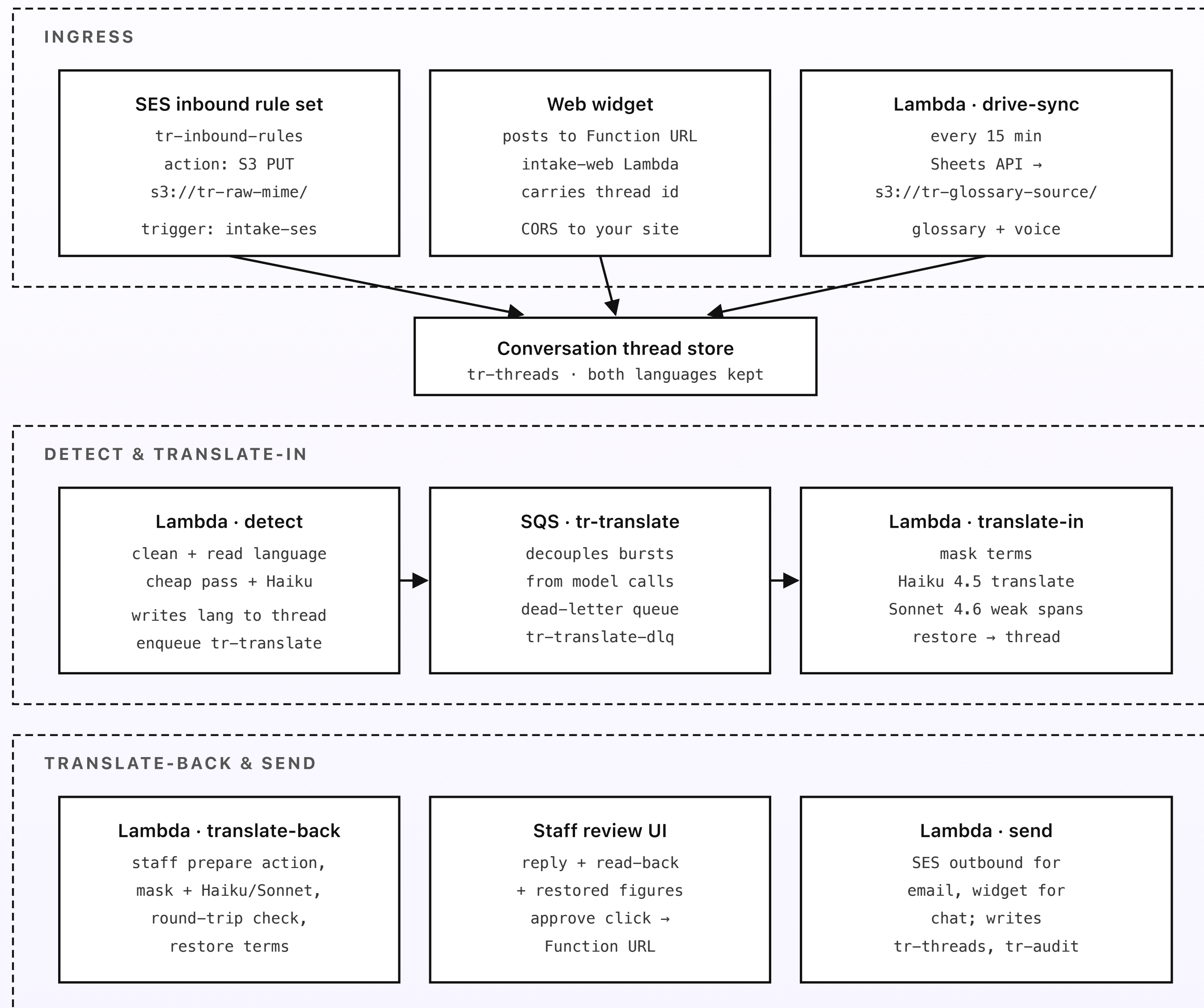
Engineering reference: the translation relay architecture

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

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound, Bedrock cross-Region inference, and Lambda Function URLs are all in good shape there, and it keeps round-trip latency low for an Asia-Pacific customer base; swap to whichever region is closest to your buyers. Bedrock calls go through the Global cross-Region inference profile, so capacity isn't pinned to one region. One AWS account dedicated to the relay keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. No VPC; every Lambda runs with default networking and reaches AWS service endpoints and the Google APIs over the public internet.

Topology



A person always approves the send — every translation and edit is logged to tr-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (two channels in plus the glossary sync), detect and translate-in (the queue decouples bursts from model calls), translate-back and send (the human approves and the reply ships). Every Lambda is event-, queue-, or request-driven; nothing is synchronous-chained across model calls.

Lambda functions

All Lambdas use the `arm64` architecture, the smallest memory size that meets latency targets (typically 256 MB, 512 MB for the model-calling functions), 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-ses`** — S3 PUT trigger on `s3://tr-raw-mime/`. Parses the MIME, extracts the plain-text body (falls back to HTML stripped to text), strips the signature, quoted history, and footers, resolves the thread by sender + normalized subject, and writes the cleaned message to `tr-threads`. Then invokes `detect` asynchronously. Memory: 256 MB. Timeout: 30 s.
- **`intake-web`** — Lambda Function URL, `AuthType: NONE` with a per-site signed widget token verified in-handler and CORS locked to your domains. Accepts `{thread_id, text}` from the chat widget, applies the same cleanup, writes to `tr-threads`, and invokes `detect`. Memory: 256 MB. Timeout: 15 s.
- **`drive-sync`** — EventBridge Scheduler target, every 15 minutes. Uses the Google Drive + Sheets API (service-account credentials in Secrets Manager under `tr/drive/sa`) to export the glossary sheet as CSV and the voice note as text, writing to `s3://tr-glossary-source/` only when changed. Memory: 256 MB. Timeout: 30 s.

- **`detect`** — invoked by the intake Lambdas. Runs a cheap script-and-statistics language check; for short, mixed, or code-heavy text, calls Bedrock Haiku 4.5 as a fallback. Writes `language` and `language_confidence` to the thread. If confidence stays below threshold, marks the turn `language_unclear` for human review instead of enqueuing. Otherwise sends a message to the `tr-translate` SQS queue. Memory: 512 MB. Timeout: 30 s.
- **`translate-in`** — SQS event source on `tr-translate` (batch size 1, partial-batch responses on). Loads the glossary from `s3://tr-glossary-source/`, masks protected terms, numbers, prices, and IDs (Part 5), calls Bedrock Haiku 4.5 for the per-sentence translation + confidence, re-runs sub-threshold passages on Bedrock Sonnet 4.6, restores the masked spans, and writes the staff-facing translation and per-passage confidence to `tr-threads`. Logs each mask/restore swap to `tr-audit`. Memory: 512 MB. Timeout: 60 s.
- **`translate-back`** — Lambda Function URL, invoked by the staff *prepare* action (Slack-style signed request or the internal review UI session). Masks the reply, translates into the thread's customer language with Haiku 4.5 + Sonnet 4.6, runs the round-trip check (translate the result back to the staff language), restores terms, and stores the prepared reply + read-back on the thread. Does not send. Memory: 512 MB. Timeout: 60 s.
- **`send`** — Lambda Function URL, invoked only by the human *approve* action. Verifies the prepared-reply hash matches what the agent approved (so an edit-after-prepare can't slip through unreviewed), then delivers: SES `SendRawEmail` for an email thread, or returns the reply to the widget poll for a chat thread. Writes the final reply (both languages) to `tr-threads` and an `action: sent` row to `tr-audit`. Memory: 256 MB. Timeout: 15 s.

- `summary` — EventBridge Scheduler target, weekly. Reads the week's `tr-threads` and `tr-audit`; calls Bedrock Haiku 4.5 to write a short report (volume by language, share of passages that needed Sonnet, count of human-flagged messages); emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB** · `tr-threads` — one item per conversation turn. PK `thread_id`; sort key `turn_ts`; attributes: `channel` (email/web), `direction` (in/out), `customer_lang`, `team_lang`, `original_text`, `translated_text`, `passage_confidence` (list), `status`. On-demand.
- **DynamoDB** · `tr-swaps` — one row per masked span. PK `(thread_id, turn_ts)`; sort key `placeholder`; attributes: `kind` (glossary/number/id), `original`, `restored`. On-demand. Proves no figure changed in translation.
- **DynamoDB** · `tr-audit` — one row per action of any kind (translate, prepare, edit, approve, send, flag). PK `(thread_id, ts)`; attributes: `action`, `by_user`, `model`, `before`, `after`. On-demand. No TTL — long-term audit trail.
- **S3** · `tr-glossary-source` — mirrored glossary CSV and voice note as plain text. Versioning enabled, so a bad glossary edit rolls back in one click.
- **S3** · `tr-raw-mime` — raw inbound email MIME. Lifecycle to Glacier at 30 days; expiry at 2 years.

Bedrock

- **Cheap path.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Callsites: `detect` fallback, `translate-in`, `translate-back`, the round-trip check, and `summary`.
- **Heavy path.** `anthropic.claude-sonnet-4-6-20250115-v1:0` via `global.anthropic.claude-sonnet-4-6-20250115-v1:0`, called only on sub-threshold passages from `translate-in` and `translate-back` — never on a whole message.
- **Embeddings.** Not used. The relay translates; it doesn't retrieve. No Knowledge Base, no S3 Vectors. (Titan Text Embeddings V2 would be the choice if a future phrase-memory cache needed semantic lookup, but exact-match caching covers the common case at lower cost.)
- **Prompts.** Strict and short: preserve placeholders verbatim, translate faithfully and plainly, return per-sentence confidence as JSON. Temperature near zero for determinism.

SQS lanes

- **`tr-translate`** — standard queue between `detect` and `translate-in`. Decouples a burst of inbound messages from the rate of model calls, so a spike never throttles Bedrock or drops a message. Visibility timeout 90 s (six times the consumer's typical runtime). Max receive count 3.
- **`tr-translate-dlq`** — dead-letter queue for `tr-translate`. Anything that fails three times lands here with the full context for inspection; a CloudWatch alarm on queue depth > 0 pages the on-call admin. Most DLQ entries are a malformed message or a transient Bedrock throttle, both safe to replay.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `support.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `tr-inbound-rules`: one rule with recipient `support@your-company.com` → spam scan → S3 PUT to `s3://tr-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses`.
- SES outbound for replies: verify a sender identity at `support@your-company.com` with DKIM and SPF on the parent domain, and set a custom `Reply-To` so the customer's next message threads back in. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **detect role:** `dynamodb:UpdateItem` on `tr-threads`; `sqs:SendMessage` on `tr-translate`; `bedrock:InvokeModel` on the Haiku ARN only. *No* Sonnet, *no* SES.
- **translate-in role:** `sqs:ReceiveMessage` + `DeleteMessage` on `tr-translate`; `s3:GetObject` on the glossary bucket; `bedrock:InvokeModel` on the Haiku and Sonnet ARNs; `dynamodb:PutItem` on `tr-threads`, `tr-swaps`, `tr-audit`.
- **translate-back role:** same Bedrock + glossary access as translate-in; `dynamodb:PutItem` on `tr-threads`, `tr-swaps`, `tr-audit`. *No* SES — it cannot send, only prepare.

- **send role:** `ses:SendRawEmail` from the verified sender identity; `dynamodb:PutItem` on `tr-threads` and `tr-audit`; `dynamodb:GetItem` to verify the approved-reply hash. *No* `bedrock:*` — the send path never calls a model.
- **drive-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the glossary bucket; outbound network to `www.googleapis.com`.

Masking pipeline

The mask/restore code is a shared library imported by `translate-in` and `translate-back`, so both directions behave identically. Order matters: IDs and codes are matched first (most specific), then currency and number patterns, then glossary terms (longest match first to avoid partial hits). Each match is replaced by a typed placeholder — `[[ID_1]]`, `[[NUM_2]]`, `[[TERM_3]]` — and recorded in an in-memory map. After the model returns, the restorer validates that every placeholder it emitted still exists exactly once (a dropped or duplicated placeholder fails the turn and routes to human review), then swaps each back to the original value and writes the swap to `tr-swaps`. Number and ID patterns are unit-tested against a fixture of locale edge cases (comma/period decimals, hash-prefixed orders, alphanumeric SKUs) so a regex change can't silently weaken the guardrail.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"placeholder_mismatch"` to a CloudWatch metric for alerting.
- **Alarms:** `tr-translate-dlq` depth > 0; `translate-in` error rate > 1% in 24h; `placeholder_mismatch` > 0 (a masking bug is a money bug); Bedrock throttle count rising.
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `tr-cost-alarm` subscribed to the on-call admin's email.

Config and secrets

Google service-account credentials for Drive and Sheets live in Secrets Manager under `tr/drive/sa`. The widget signing key is under `tr/widget/key`; the SES sender identity lives in IAM and the verified-domain config. The team's working language, the confidence thresholds for Sonnet escalation and for human-flagging, the list of glossary categories, and the customer-facing "from" name all live in Parameter Store under `/tr/config/`. Lambdas fetch config on cold start and cache it for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) and AWS SAM for the stack. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for `tr-glossary-source` so a bad glossary edit can be rolled back in one click, and gate deploys on the masking

library's unit tests passing — that test suite is the thing standing between a regex tweak and a changed price in a customer's inbox. Total deployable surface: around eight Lambdas, three DynamoDB tables, two S3 buckets, one SQS queue plus its DLQ, one SES rule set, a few Function URLs, 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.