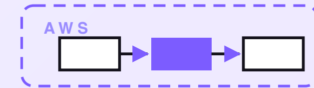
7-PART SERIES · FREE COMPANION



Missed-call text-back

When a call to the business goes unanswered, the customer is left listening to a dial tone — and a good number of them just ring the next shop down the list. This is the design of a small serverless system that turns a missed call into an instant, friendly text back: it matches the caller to any known customer, sends one SMS in the business's voice with a booking link, threads any reply to the right person, and escalates anything urgent or unanswered to a human. It sends exactly one text per missed call, honours quiet hours and STOP opt-outs, and never spams. 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/missed-call-text-back

CONTENTS

Missed-call text-back

01 A missed-call text-back on AWS for a few dollars a month

02 How a missed call gets caught

03 How a text-back gets composed

04 How a reply gets routed

05 How an unanswered call-back gets escalated

06 What the missed-call text-back costs

07 Engineering reference: the missed-call text-back architecture

PART 1 OF 7

JULY 1, 2026 PART 1 OF 7 · MISSED-CALL TEXT-BACK SERIES ~10 MIN READ

A missed-call text-back on AWS for a few dollars a month

A missed call is the quietest way a small business loses a customer: the phone rings out, the caller hangs up, and more often than not they ring the next shop down the list. This post walks through the design of a small serverless system that turns that missed call into an instant, friendly text back — matched to the caller, with a booking link — and quietly hands the hard ones to a person.

KEY TAKEAWAYS

- A missed call fires a webhook from your telephony provider; within seconds the caller gets one friendly text back.
- The caller's number is matched to any known customer or order, so the text can use their name and the right booking link.
- One Bedrock call writes the wording in your voice. Everything else — the trigger, the match, the routing — is plain Python.
- It sends exactly one text per missed call, honours quiet hours and STOP, and never cold-texts or chases.
- Designed on AWS for about $2.20/month at roughly 150 missed calls a month. It only ever texts back — a human handles the rest.

The whole system on one page

Before any code, here's the shape of what we're designing. Every small business that takes calls misses some of them — the till is busy, it's after hours, both lines are engaged — and a missed call is almost invisible. There's no voicemail half the time, no record anyone looks at, and the caller simply rings the next shop down the list. The system below catches that moment and turns it into a conversation: a missed call comes in, and a friendly text goes back out, matched to the caller, within seconds.

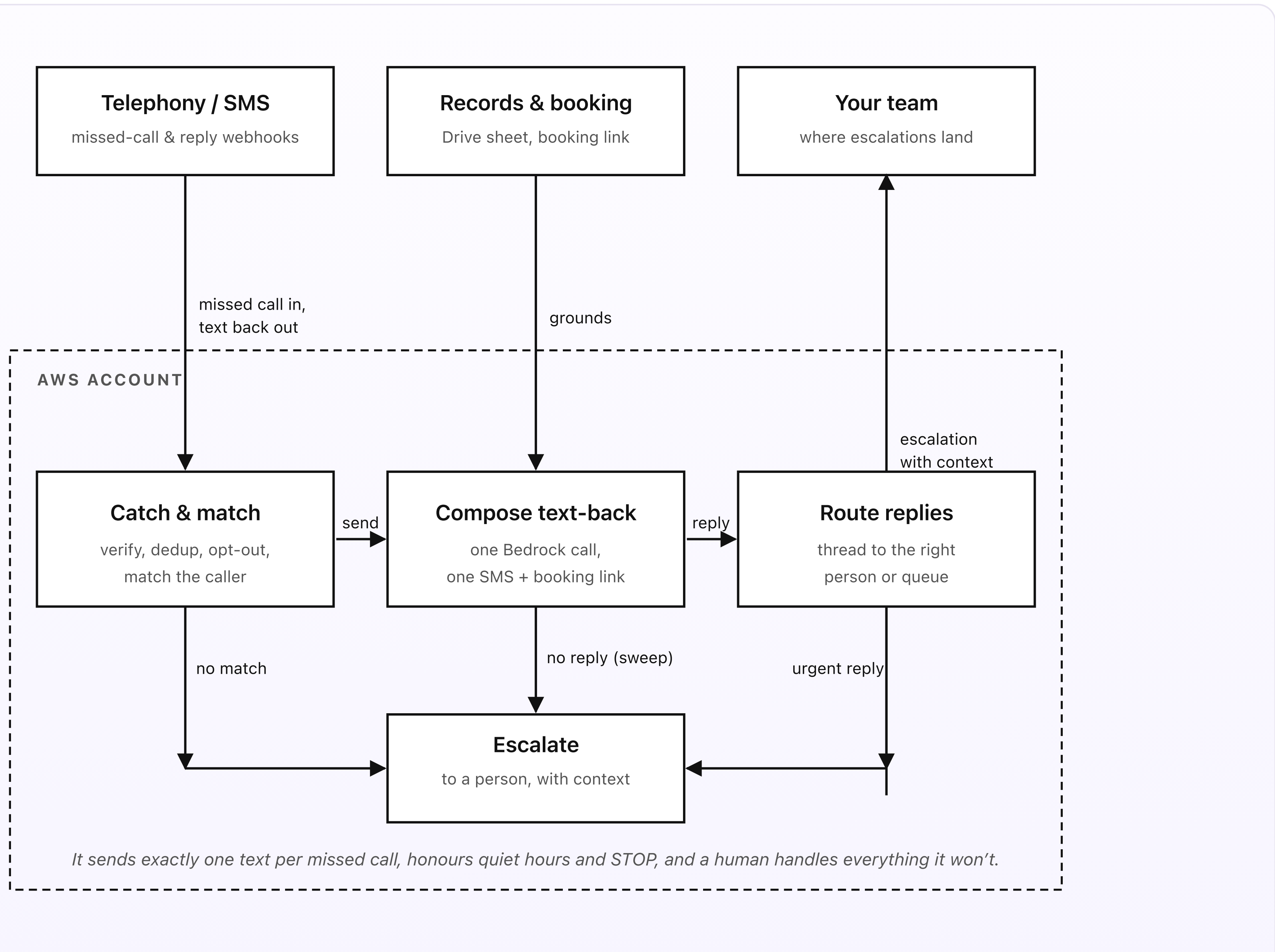

Telephony / SMS
missed-call & reply webhooks
Records & booking
Drive sheet, booking link
Your team
where escalations land
missed call in,
text back out
grounds
AWS ACCOUNT
escalation
with context
Catch & match
verify, dedup, opt-out,
match the caller
send
Compose text-back
one Bedrock call,
one SMS + booking link
reply
Route replies
thread to the right
person or queue
no match
no reply (sweep)
urgent reply
Escalate
to a person, with context
It sends exactly one text per missed call, honours quiet hours and STOP, and a human handles everything it won't.

Fig 1. Three things outside, four pieces inside AWS. A missed call comes in from the provider; Catch & match decides whether to reply and to whom, Compose sends one text-back with a booking link, and Route threads any reply to a person. Anything unmatched, urgent, or unanswered branches to Escalate.

What you set up once (the outside)

- **Telephony and SMS provider.** Whatever already carries your business number — a VoIP or cloud-phone provider that can do two things: fire a webhook the moment a call goes unanswered, and send and receive SMS on that same number. The missed-call webhook is the trigger for everything; the SMS side is how the text-back goes out and how replies come back. You point the provider's webhook at one AWS URL and store its keys in Secrets Manager. This is the only moving part you don't own, and it's covered in Part 2.
- **Customer records and a booking link.** A Google Sheet in a Drive folder with one row per customer or recent order: name, phone, last visit or order, and a note or two. You already keep something like this; this just puts it where the system can read it, so a text can open with the caller's name instead of "Hi there". Alongside it sits the one thing every text points to — your booking link (or a short FAQ page) — and a small settings doc for the voice, the opening hours that define "quiet", and the escalation rules.
- **Your team.** The person who picks up everything the system deliberately won't handle on its own — usually whoever answers the phone. They get a message (email, or a row in a shared inbox) with the caller's number, the customer it was matched to, the text that went out, and the reply so far. The system never books, quotes, cancels, or promises anything; it opens the conversation and a human carries it on.

What runs on every missed call (the inside)

- **Catch and match.** The provider posts a missed-call event to one Lambda Function URL. The function verifies the signature, collapses repeat rings from the same number into a single job, checks the number against the STOP opt-out list and the current quiet hours, and matches the caller against the customers mirrored from the sheet. Only then does it decide a text-back is warranted. This is Part 2.
- **Compose text-back.** One Bedrock Haiku 4.5 call takes the handful of facts it's allowed — the caller's first name if known, the business name, the booking link — and writes a single short, friendly SMS in your voice. The model writes words and nothing else; it never decides whether to send, to whom, or which link to use. This is Part 3.
- **Route replies.** When the caller texts back, that reply lands on the same webhook, is matched to the original missed call, and is threaded to the right person or queue. The system doesn't try to hold the conversation; it makes sure a human sees the reply with the full thread. This is Part 4.
- **Escalate.** The lane for everything that shouldn't be left to an automatic text: an unmatched caller, a reply that reads as urgent, and — via a scheduled sweep — any text-back that's gone unanswered past a set window. Each lands with a person, with the call and the thread attached. This is Part 5.

In plain words

It's 1:10pm and the salon's only stylist is mid-colour when the phone rings out. The provider fires a missed-call webhook for `+44 7700 900412`. The system matches that number to Priya, who was in three weeks ago, and within about ten seconds

her phone buzzes: “Hi Priya — sorry we missed your call at Bloom! We’re with a client right now. Text us here and we’ll sort you out, or grab a slot at bloomhair.uk/book.” She taps the link, books a cut for Friday, and never had to ring back. Nobody at the salon touched a thing until the booking dropped in.

An hour later a different number rings out twice in a minute, then texts back: “my mum’s appointment is in 20 mins and we’re stuck in traffic, are we ok?” That isn’t a booking nudge — it’s time-critical. The system has already sent its one text-back; it does *not* try to answer the question. It threads the reply, flags it *urgent*, and pushes it straight to the front desk’s phone so a person can ring them in the next minute. One text-back, then a human, with the whole thread already in hand.

DESIGN RULES THAT SHAPED EVERY DECISION

- One missed call, one text. Repeat rings from the same number collapse to a single reply — it never spams.
- It opens the door, it doesn't walk through it. The system texts back and threads replies; it never books, quotes, or promises.
- A confident match or a plain greeting. Known callers get their name; unknown numbers get a warm generic text, and anything odd escalates.
- The model only writes words. The trigger, the match, the dedup, and the routing are deterministic; Bedrock just phrases the SMS.
- Quiet hours and STOP are sacred. No texts overnight, and anyone who texts STOP is suppressed for good.
- It never goes quiet on a person. Urgent replies and unanswered text-backs are escalated to a human with full context.

Why this shape

Most small businesses handle missed calls one of three ways: they don't know about them at all, they glance at the call log at the end of the day and ring a few back, or they pay for a receptionist or an answering service. The first loses the customer silently — a missed call rarely calls twice. The second is hours too late; by the evening the caller has already booked elsewhere. And a full answering service is real money every month for something a single text would have solved. The gap is the first sixty seconds, and nothing in a typical small shop is watching them.

The shape above fills exactly that gap and nothing more. It leans on the phone provider you already pay for as the trigger, keeps the customer sheet you already maintain as the source of names, and adds a small system that sends one good text the instant a call is missed. The common case — “I just wanted to book” — turns into a tap on a booking link with no human involved. The few that are genuinely urgent or off-script are pulled out early and put in front of a person, with the call and the conversation already gathered.

The next four posts walk through each piece in turn: how a missed call gets caught and matched, how a text-back gets composed, how a reply gets routed, and how an unanswered call-back gets escalated. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JULY 1, 2026 PART 2 OF 7 · MISSED-CALL TEXT-BACK SERIES ~7 MIN READ

How a missed call gets caught

Before a single text goes out, the system has to be sure it should send one at all: that the call really was missed, that it hasn't already replied to this caller, that it isn't the middle of the night, and that the number hasn't opted out. This post is about that gate — how a raw webhook becomes a safe, matched, send-once job.

KEY TAKEAWAYS

- The trigger is a missed-call webhook from your telephony provider, posted to one Lambda Function URL — no API Gateway.
- The webhook is verified by its signature before anything else runs, so a stranger can't make the system text people.
- Repeat rings from the same number inside a short window collapse to one job, so a nervous caller gets one text, not five.
- Quiet hours and the STOP opt-out list are checked before sending — nobody is texted overnight, and an opt-out is permanent.
- The caller's number is matched to a known customer for the name and booking link; an unknown number still gets a warm generic text.

From a ring-out to a job

Everything starts with one event the business never normally sees: a call that rang out. Telephony providers expose this as a webhook — when a call ends without being answered (and without going to a voicemail the caller actually used), the provider sends a small HTTP POST with the caller's number, the time, and a call id. That POST lands on a single Lambda Function URL. There's no API Gateway in front of it; a Function URL is a plain HTTPS endpoint on the function itself, which is all a webhook needs and the cheapest way to receive one.

The catch function's job is not to send a text. Its job is to decide whether a text is warranted at all, and to do that safely and exactly once. Most of this post is about the checks that sit between "a webhook arrived" and "a text-back is queued", because that gap is where a missed-call system either earns trust or becomes a nuisance.

Prove it's real, then prove it's new

The first check is authenticity. A Function URL is public, so the first thing the function does is verify the provider's signature — a hash of the request body signed with a shared secret held in Secrets Manager. If the signature doesn't match, the request is dropped with a `401` and nothing else happens. Without this, anyone who found the URL could make your system text strangers all day; with it, only your provider can trigger a send.

The second check is duplication, and it's the one that keeps the system from becoming spam. A worried caller often rings two or three times in quick succession; some providers also retry a webhook they think failed. Either way, the

system must send exactly one text per missed call. So the function writes a record keyed on the caller's number with a short dedup window — a few minutes — using a conditional write to DynamoDB. The first ring in that window wins and creates the job; any further rings or retries inside the window see the existing record and stop. One text-back, however many times the phone rang.

Should we text at all?

Two more gates decide whether sending is even allowed, regardless of who called. **Quiet hours** come from the settings doc: the business's opening hours, plus a rule for after-hours calls. A call missed at 2am shouldn't buzz someone's phone at 2am, so the text is held and released when quiet hours lift — the caller still hears back first thing, before they've chased anyone else. **Opt-out** is the STOP list: anyone who has ever replied STOP to a text is recorded as suppressed, and the system will never text that number again. Both checks run before a single word is composed, because the cheapest message to handle is the one you correctly decide not to send.

Only once a call is verified, de-duplicated, inside allowed hours, and not opted out does the function look up *who* called.

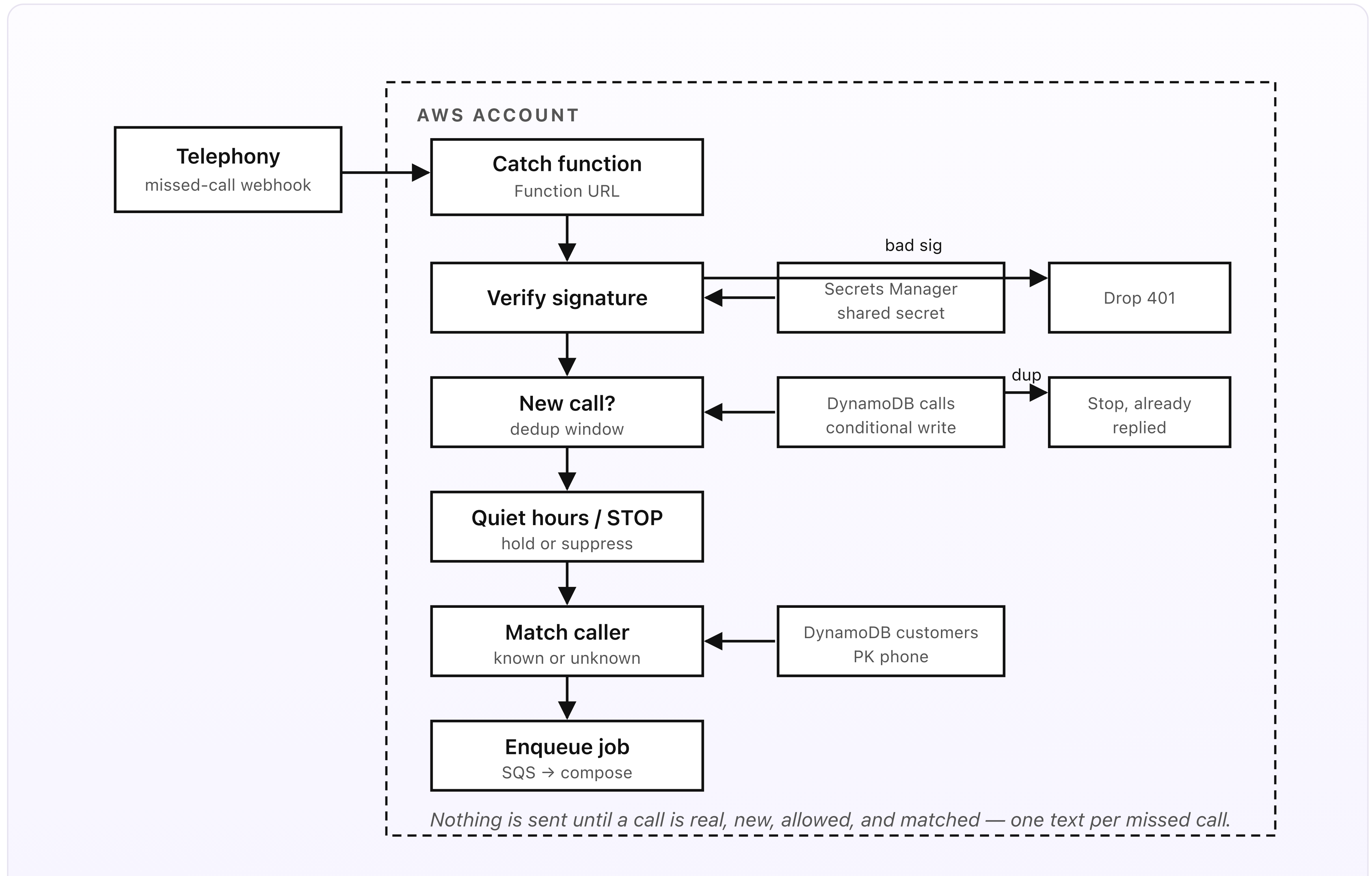

AWS ACCOUNT
Telephony
missed-call webhook
Catch function
Function URL
Verify signature
bad sig
Secrets Manager
shared secret
Drop 401
New call?
dedup window
DynamoDB calls
conditional write
dup
Stop, already replied
Quiet hours / STOP
hold or suppress
Match caller
known or unknown
DynamoDB customers
PK phone
Enqueue job
SQS → compose
Nothing is sent until a call is real, new, allowed, and matched — one text per missed call.

Fig 2. The catch gate. A missed-call webhook is verified by signature, de-duplicated with a conditional write so repeat rings make one job, checked against quiet hours and the STOP list, and matched to a customer — only then is a text-back job enqueued.

Matching the caller

Matching here is simpler than it sounds, because a phone call gives you the cleanest possible key: the caller's number. The function normalises it to a single canonical form (E.164, the `+44…` international format) so that `07700 900412` and `+447700900412` are the same person, then looks it up in the customers table mirrored from your Drive sheet. A hit returns the customer's first name and, if you tag rows that way, the right booking link — a regular at the barber gets a different link from a first-time enquiry. A miss is not a failure: an unknown number still gets a warm, generic text-back inviting them to book or reply, just without a name.

The only case that doesn't flow straight to compose is the one that's genuinely ambiguous or risky — a withheld or malformed number the provider couldn't give you, or a number flagged in your sheet as "do not auto-text" (a supplier, say, or a known nuisance caller). Those are recorded and, if they look like a real missed customer, dropped into the escalation lane for a person to glance at, rather than guessed about. Everything else — the overwhelming majority — becomes a clean job on the SQS queue: caller number, matched name if any, booking link, and the original call id, ready for Part 3 to phrase.

DESIGN RULES THAT SHAPED THE CATCH STEP

- One public surface. A single Lambda Function URL receives the webhook; there is no API Gateway and no other way in.
- Verify before you trust. A bad signature is dropped with a 401 before any work — the URL being public is fine because the secret isn't.
- One call, one text. A conditional write over a short window collapses repeat rings and provider retries into a single job.
- Decide not to send, cheaply. Quiet hours and the STOP list are checked before composing — the held or suppressed message costs nothing.
- The number is the key. Match on the normalised caller number; a known caller gets a name, an unknown one still gets a warm text.
- When in doubt, a person. Withheld, malformed, or flagged numbers escalate rather than getting an automatic guess.

PART 3 OF 7

JULY 1, 2026 PART 3 OF 7 · MISSED-CALL TEXT-BACK SERIES ~8 MIN READ

How a text-back gets composed

By the time this step runs, the system knows it should reply, and to whom. All that's left is to say it like the business would. This post is about the only place a model is allowed near the system: the single Bedrock call that turns a few known facts into one warm, on-brand SMS with a booking link — and the fences that keep it to one honest message.

KEY TAKEAWAYS

- One Bedrock Haiku 4.5 call per missed call writes a single short SMS in the business's voice — the only place a model runs.
- The model is handed only the facts it may use: the caller's first name if known, the business name, and the booking link.
- The booking link is injected by code after the model writes, never typed by the model, so it can never be mangled or invented.
- The draft is checked for length, a single link, and an opt-out hint before it's sent — a bad draft falls back to a fixed template.
- The model decides wording only. Whether to send, to whom, and which link to use were all settled in Part 2.

All that's left is to say it

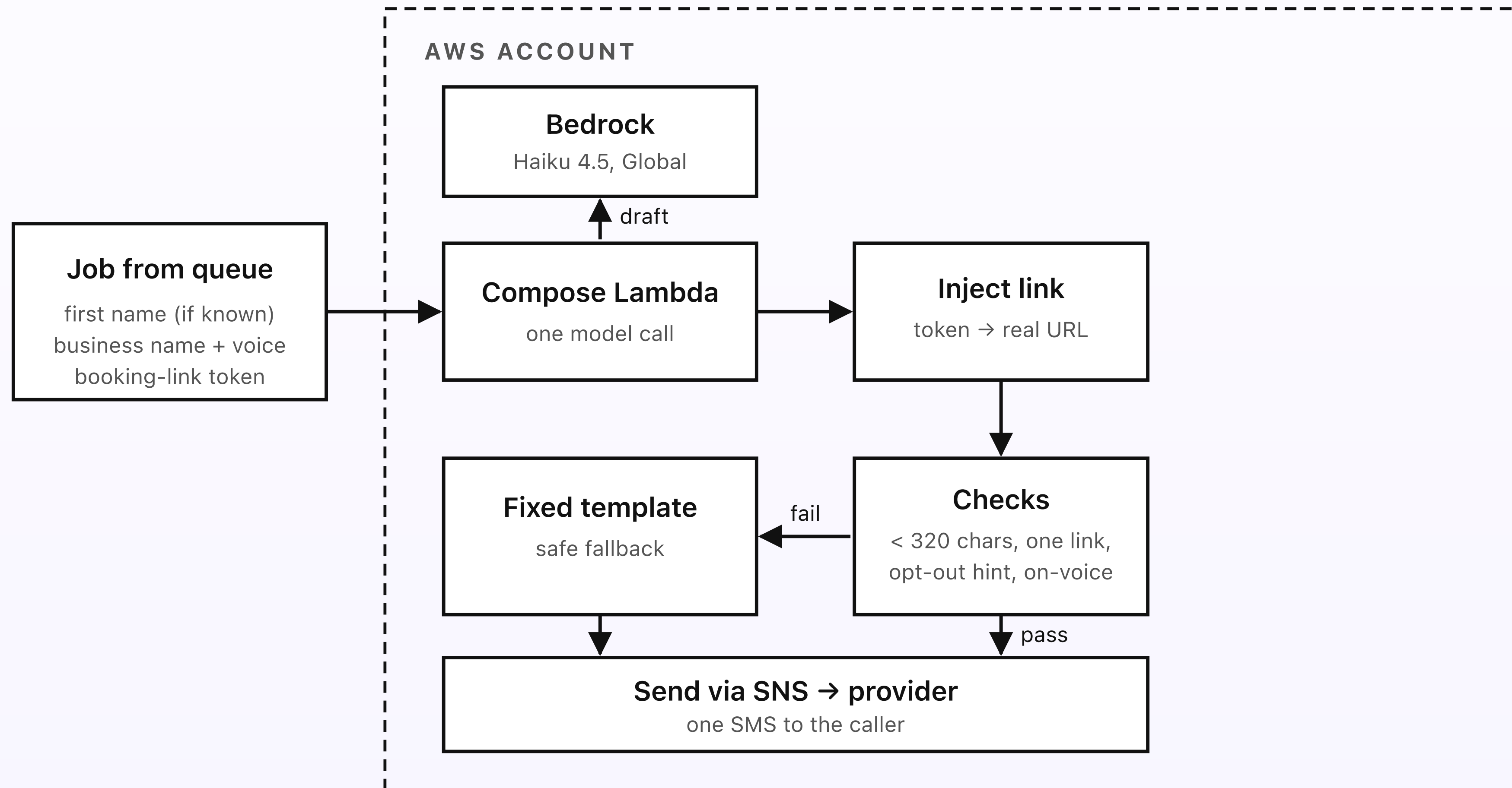
By the time this step runs, every real decision has already been made. The catch function in Part 2 proved the call was real and new, cleared it against quiet hours and the opt-out list, and matched the caller. The job on the queue already carries the caller's number, their first name if we know it, and the booking link that suits them. Nothing about *whether* to text, or *whom* to text, is open any more. The only thing left is the wording — and wording is the one thing a small business actually cares about, because a text that sounds like a robot is worse than no text at all.

So this is the single place in the whole system where a model runs. One Bedrock call, using Claude Haiku 4.5, takes a few facts and writes one short, warm SMS that sounds like the business wrote it. Haiku is the right tool here precisely because the task is small: a sentence or two, in a set tone, with no reasoning to do. It's fast and it costs a fraction of a penny, which matters when it runs on every missed call.

Handed only the facts it may use

The prompt is deliberately narrow. The model is given the business name and voice notes from the settings doc ("friendly, first-person plural, no exclamation-mark soup"), the caller's first name if the match found one, and a placeholder where the booking link will go. It is told, in plain terms, to write one SMS under 320 characters that apologises for the missed call, invites a reply, and points to the booking link — and to use *only* those facts. It has no tools, no web access, and no knowledge of the customer beyond the first name it was handed, so there is nothing for it to over-share or invent.

The booking link itself is never written by the model. The model emits a token like `{{booking}}`, and the code substitutes the real URL afterwards. This is a small thing that matters a lot: language models are perfectly capable of subtly mistyping a URL or “improving” it, and a broken booking link is the one error that quietly wastes the whole text. By keeping the link out of the model’s hands and injecting it deterministically, the link is always exactly right.



The model writes words only; the link is injected by code, every draft is validated, and a bad draft falls back to a fixed template.

Fig 3. Composing one text-back. A single Bedrock call drafts the wording with a link placeholder; code injects the real booking URL, the draft is validated for length, one link, and an opt-out hint, and anything that fails falls back to a fixed template before the one SMS is sent.

Checked before it goes

A model writing a one-off SMS is low-risk, but "low" is not "none", so the draft is validated by code before it leaves the building. An SMS is billed and split by length, so the text must come in under a single sensible limit — 320 characters — or it's trimmed or rejected. It must contain exactly one link, the one we injected, and no others. It must carry the small opt-out hint that keeps the business on the right side of messaging rules ("reply STOP to opt out"), and it must not contain anything that reads as a price, a promise, or a confirmed booking, because the system is opening a conversation, not closing a sale.

If the draft fails any of those checks — or if Bedrock is slow or errors — the composer doesn't retry forever or send something half-formed. It falls back to a fixed, hand-written template: "Hi{name}, sorry we missed your call at {business}. Reply here or book at {link}. Reply STOP to opt out." The fallback is plain, but it's always correct, always on time, and always within the rules. The model makes the common case sound human; the template guarantees the system never goes silent just because a model had a bad moment.

Why let a model near it at all

It's fair to ask why this needs a model when a template already exists as the fallback. The honest answer is tone and fit. A regular who's been in five times

reads differently from a first-time caller; a 9am missed call reads differently from a 6pm one; a dog groomer and a solicitor want very different warmth. A single template can't flex to all of that without becoming a pile of `if` statements, and a stiff template is exactly what makes customers feel they're talking to a machine. One cheap Haiku call gives every text-back a little of the business's actual character, while the deterministic link injection, the validation, and the template fallback make sure that character never comes at the cost of correctness. The model gets the warmth; the code keeps the guarantees.

DESIGN RULES THAT SHAPED THE COMPOSER

- One call, one text. A single Haiku call writes one SMS — no chains, no retries that could fan out into more messages.
- Only the facts it's handed. First name, business, voice, and a link token — no tools, no lookups, nothing to over-share.
- The link is the code's job. The model emits a token; the real booking URL is injected afterwards so it can never be mangled.
- Validate every draft. Length, a single link, an opt-out hint, no prices or promises — checked before it's sent.
- A template is always ready. If the model fails or drifts, a fixed, correct text goes out instead — the system never goes quiet.
- The model decides nothing. Whether, to whom, and which link were all settled upstream; the model only chooses words.

PART 4 OF 7

JULY 1, 2026 PART 4 OF 7 · MISSED-CALL TEXT-BACK SERIES ~7 MIN READ

How a reply gets routed

The point of the text-back isn't the text; it's the reply it earns. When the caller texts "yes please, can you do Thursday?" that message has to reach a real person with the full thread attached. This post is about routing: how an inbound reply is matched to its missed call, threaded, and put in front of the right human.

KEY TAKEAWAYS

- When the caller texts back, that reply lands on the same Function URL the missed call used — one inbound surface for everything.
- The reply is matched to its missed call by the caller's number and threaded, so a person sees the whole conversation, not a stray text.
- The system never auto-answers a reply. Its job is to put the right human in front of the caller with full context.
- Anything that reads as urgent — or any reply from an unmatched number — is escalated immediately rather than queued.
- A STOP reply is honoured the instant it arrives: the number is suppressed, no acknowledgement is sent, and no one is texted again.

The reply is the point

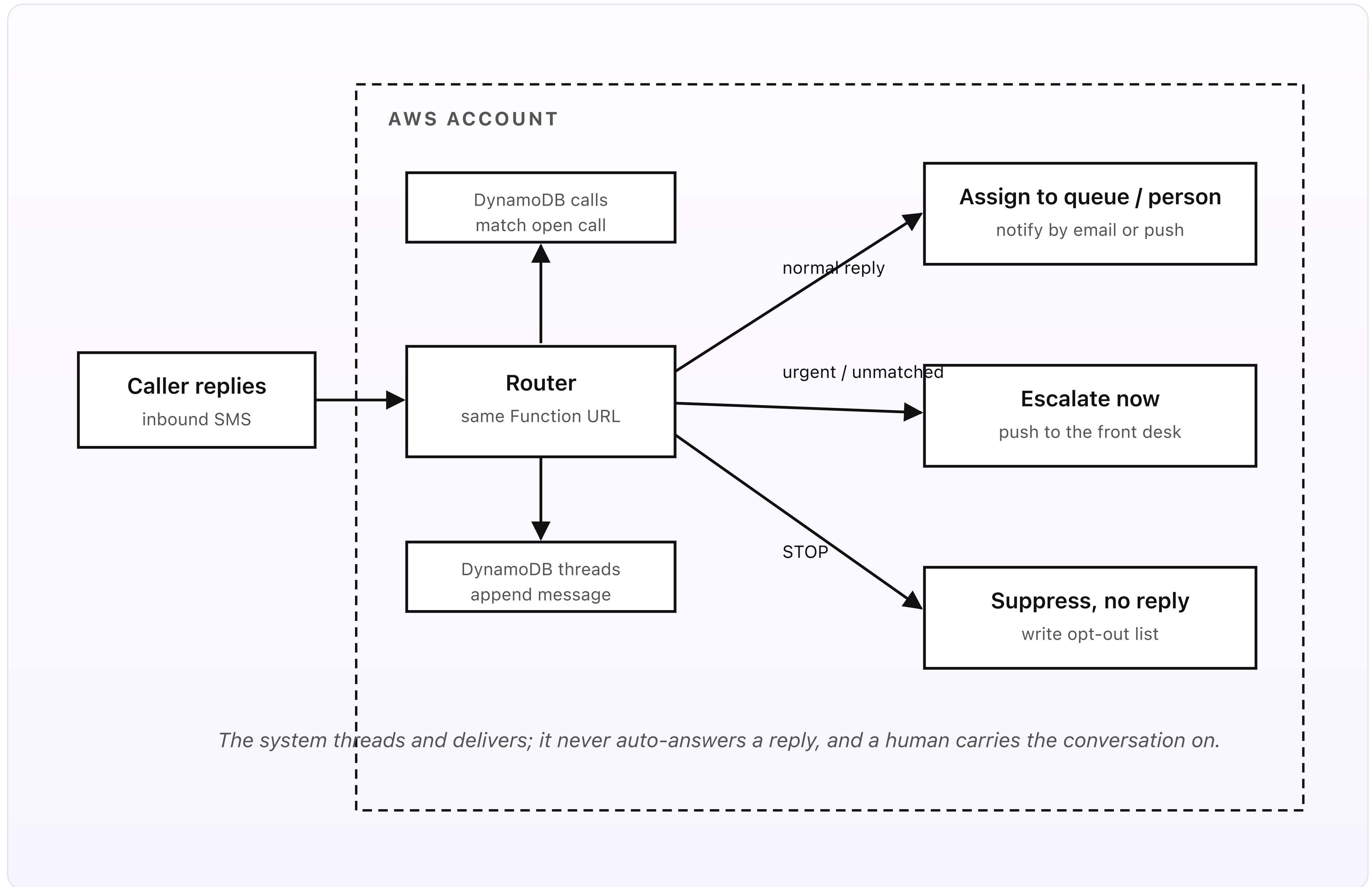
The text-back is only ever the opening line. What actually saves the customer is the reply it earns — “yes please, can you do Thursday?” or “do you take card?” or “actually I need to cancel”. That message has to reach a real person at the business quickly and with the context already attached, because a reply that lands in a void is worse than no text-back at all: it raises the customer’s hopes and then drops them. So routing gets its own piece of the system, and a firm rule: the system threads and delivers replies, it does not answer them.

That rule is deliberate. It would be tempting to let the model carry on the conversation — check the diary, offer a slot, confirm a booking. But that’s exactly where an automated texter starts making promises the business can’t keep, double-booking the Thursday slot, or confidently quoting a price that’s wrong. The system opened the door; a human walks through it. What routing does is make sure the human is the *right* one and that they’re looking at the whole thread.

One inbound surface, matched and threaded

The provider posts inbound SMS to the same Lambda Function URL that receives missed-call events; the function tells the two apart by the event type in the payload. An inbound reply carries the sender’s number, which is the same key the missed call used, so matching the reply back to its originating call is a direct lookup: find the most recent open call record for that number. That record links to a **thread** — a row in the threads table that holds the conversation state: which missed call started it, what text-back went out, who (if anyone) is handling it, and whether it’s still open.

Threading matters because it’s what turns a loose SMS into something a person can act on. Without it, the front desk sees “Thursday works” from an unknown number and has no idea what it’s about. With it, they see: *missed call at 1:10pm from Priya (last in 3 weeks ago), we texted back offering a booking link, she replied ‘Thursday works’.* Every inbound message is appended to its thread, so the history is always in one place, and a reply to a reply stays in the same conversation rather than starting a new one.



The system threads and delivers; it never auto-answers a reply, and a human carries the conversation on.

Fig 4. Routing a reply. The inbound SMS hits the same Function URL, is matched to its open missed call and appended to the thread, then triaged: normal replies go to a person or queue, urgent or unmatched ones escalate immediately, and a STOP suppresses the number with no acknowledgement.

Triage: who sees it, and how fast

Once a reply is threaded, the router decides where it goes. Most replies are ordinary — a booking question, a “yes that works”, a quick query — and those are assigned to the right person or shared queue and surfaced however the business already works: an email to the support address, a row in a shared inbox, or a push to the front-desk phone. The settings doc decides the mapping; a two-person shop sends everything to one number, a larger one might split by service.

Two kinds of reply jump the queue. The first is anything that reads as **urgent or time-critical** — “outside now”, “running late”, “emergency”, “in 20 minutes”. Those are pushed straight to a human immediately rather than sitting in a queue, because a slow reply to a time-critical text is the same as no reply. The second is a reply from a number the system can’t tie to any recent missed call — someone texting the business number cold, or a thread that’s gone stale. Rather than guess, the router escalates it as an unmatched message for a person to read. Detecting “urgent” is kept deliberately simple and deterministic — a keyword and recency check, erring on the side of escalating — rather than asking a model to judge tone, because the cost of under-reacting here is a missed appointment.

STOP means stop

One reply is handled before any routing at all: `STOP` (and its common variants — "unsubscribe", "opt out"). The instant the router sees one, it writes the sender's number to the opt-out list that Part 2 checks, and it does nothing else — no "you've been unsubscribed" confirmation, because that would be one more unwanted text. From that moment the number is suppressed everywhere: no text-backs, no escalation texts, nothing. Honouring STOP immediately and silently is both the polite thing and the compliant thing, and keeping the opt-out list as the single source of truth — checked before every send — is what makes the whole system safe to point at real customers.

DESIGN RULES THAT SHAPED ROUTING

- One inbound surface. Replies and missed calls hit the same Function URL; the event type tells them apart.
- Match by the number, thread by the call. Every reply is tied to its open missed call and appended to one conversation.
- Thread, don't answer. The system delivers the reply to the right human with full context; it never auto-replies.
- Urgent jumps the queue. A simple keyword-and-recency check pushes time-critical replies straight to a person.
- Unmatched goes to a human. A reply with no matching call is escalated to be read, never guessed about.
- STOP is instant and silent. Suppress the number at once, send nothing back, and check the list before every send.

PART 5 OF 7

JULY 1, 2026 PART 5 OF 7 · MISSED-CALL TEXT-BACK SERIES ~7 MIN READ

How an unanswered call-back gets escalated

Most text-backs do their job: the caller replies and books. But some don't — the message lands, nobody answers, and without a nudge that caller is exactly the one who quietly drifts away. This post is about the safety net: the scheduled sweep that notices a text-back has gone unanswered and puts the caller in front of a person to follow up.

KEY TAKEAWAYS

- Some text-backs are never answered — and an unanswered text-back is a customer quietly slipping away, so the system watches for it.
- An EventBridge Scheduler rule runs a sweep on a fixed cadence, looking for calls that were texted but have had no reply within a set window.
- Each stale call is handed to a person to ring back, with the call time, the matched customer, and the text that went out attached.
- The sweep escalates a call exactly once, respects quiet hours, and never sends the caller a second automated text.
- It's the difference between "we texted them" and "we got them back" — the net under the whole system.

The ones that go quiet

Most text-backs work: the caller reads it, taps the booking link or texts back, and Part 4 takes over. But some don't. The text lands, the caller means to reply, and then the bus comes or the kettle boils and they forget. From the business's side this looks identical to success — a text was sent — which is exactly why it's dangerous. That caller had a need a few minutes ago and now they're drifting, and nobody at the shop has any reason to notice. A system that only ever fires on incoming events would never see this, because the thing that needs acting on is an event that *didn't* happen: a reply that never came.

Catching an absence needs something that runs on a clock rather than on a trigger. That's the escalation sweep: a small scheduled job whose entire purpose is to find text-backs that have gone unanswered past a sensible window and put the caller in front of a person before they're gone for good.

A job on a clock

The sweep is driven by EventBridge Scheduler — a managed cron that invokes a Lambda on a fixed cadence, with no always-on compute and effectively no cost. A sensible cadence is every 15 minutes during opening hours. Each run does the same simple thing: query the calls table for records that are *texted but not replied*, where the text-back went out longer ago than the no-reply window (say, 30 minutes) and which haven't already been escalated. A short window catches people while their intent is still warm; making it configurable lets a business tune it — a restaurant taking bookings wants minutes, a tradesperson quoting jobs might be happy with a couple of hours.

For each call the sweep finds, it builds a handover and hands it to a person to ring back: the time of the missed call, the customer it was matched to (and their history, if any), the exact text-back that went out, and the fact that there's been no reply. Then — and this is the important part — it marks the call *escalated* on the same record, so the next run 15 minutes later doesn't flag the same call again. A call is escalated exactly once. The sweep raises a human, it does not send the caller a second automated text; the whole design rests on one text per missed call, and a chasing text would break that promise and read as nagging.

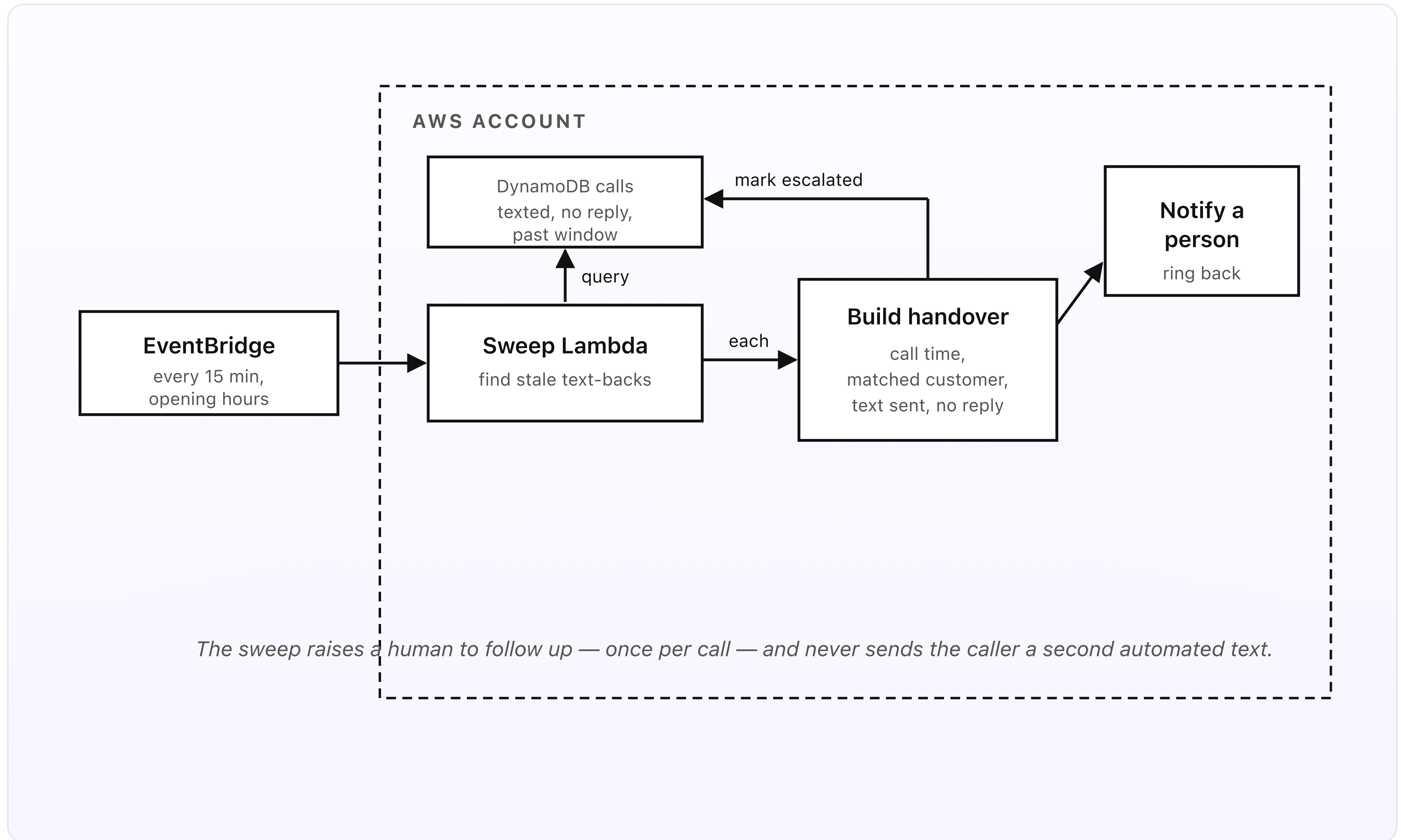


Fig 5. The no-reply sweep. EventBridge Scheduler runs the sweep on a fixed cadence; it queries for text-backs older than the no-reply window with no reply, builds a handover for each, notifies a person to ring back, and marks the call escalated so it's only ever

flagged once.

Why a sweep, and not a timer per call

You could imagine scheduling a one-off check for each individual text-back — set a 30-minute timer when the text goes out, fire it, see if there's a reply. It works, but it means a scheduled entity per missed call, more moving parts to create and clean up, and a pile of near-simultaneous invocations on a busy morning. A single periodic sweep is simpler and cheaper: one rule, one function, one query that scales to whatever volume the day brings, and idempotent by design because the *escalated* flag means re-running it never double-notifies. At this scale the 15-minute granularity is plenty — the difference between a 30- and a 42-minute callback is nothing next to the difference between a callback and none.

The sweep also respects the same guardrails as everything else. It runs only during opening hours, so it doesn't ping the owner's phone overnight; a call missed at 11pm waits and is swept in the morning. It checks the opt-out list, so a number that texted STOP after the text-back is never escalated. And it touches only its own state — it reads calls and writes the escalated flag, nothing more — which keeps it safe to run unattended every quarter of an hour, forever.

DESIGN RULES THAT SHAPED ESCALATION

- Watch for the event that didn't happen. A reply that never came is the signal; only a scheduled sweep can see an absence.
- A clock, not a trigger. EventBridge Scheduler runs the sweep on a fixed cadence with no always-on compute.
- Escalate once. The *escalated* flag on the call record makes the sweep idempotent — no call is ever flagged twice.
- Raise a human, don't chase the caller. The sweep notifies a person to ring back; it never sends a second automated text.
- The same guardrails apply. Opening hours and the opt-out list are honoured by the sweep exactly as by the live path.
- One sweep beats many timers. A single periodic query is simpler, cheaper, and idempotent than a scheduled entity per call.

PART 6 OF 7

JULY 1, 2026 PART 6 OF 7 · MISSED-CALL TEXT-BACK SERIES ~6 MIN READ

What the missed-call text-back costs

A system that costs more than the business it saves is a gadget. This post is the cost breakdown: every AWS service this design touches, what each adds up to at around 150 missed calls a month, and why the total lands near $2.20 — plus what happens to the bill when the volume goes up tenfold.

KEY TAKEAWAYS

- About $2.20/month at roughly 150 missed calls, and the fixed cost is almost nothing — nothing runs when the phone isn't ringing.
- The two biggest lines are the outbound SMS and one small Bedrock call per text-back. Everything else is cents.
- The only real fixed cost is Secrets Manager: two provider secrets at $0.40 each, billed whether or not a call comes in.
- At ten times the volume (around 1,500 missed calls) the bill lands near $14 — it scales with use, not with idle time.
- SMS carrier fees vary by country and provider; the numbers here are a UK-leaning estimate, not a fixed AWS price list.

Where the money goes

The system is serverless end to end, so there's no instance ticking over when the shop is shut and no idle bill. You pay for a missed call only when one happens. At a typical small-business volume — call it 150 missed calls a month, each getting one text-back, with a handful of replies and escalations on top — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two provider secrets — webhook signing key, SMS API key ($0.40 each)	$0.80
SNS (SMS)	One outbound text-back per missed call (~150), plus a few escalation texts	$0.60
Bedrock (Claude Haiku 4.5)	One compose call per text-back (~150)	$0.45
DynamoDB (on-demand)	Calls, threads, customers mirror, audit — small reads and writes	$0.12
CloudWatch Logs	Function logs, 7-day retention	$0.10
SES	Escalation and handover email to your team	$0.05
Lambda (Python 3.14, arm64)	Webhook, replier, router, escalator, sweep, drive-sync	$0.05

AWS service	What it does here	Monthly
SQS + DLQ	Buffering between the webhook and the slower model and SMS calls	$0.02
EventBridge Scheduler	The 15-minute no-reply sweep and the Drive sync	$0.01
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~150 missed calls/month**	**$2.20**

The shape of that bill is the point. The only line that costs money while the system sleeps is Secrets Manager — two secrets at $0.40 each, $0.80 a month no matter what, which is over a third of the total at this volume. Everything else is genuinely usage-priced and rounds to zero at idle. The two lines that do move with volume are the ones that actually reach the customer: the SMS itself and the one Haiku call that phrases it. The Lambdas, the queue, the tables, and the schedules — all the machinery doing the real work of catching, matching, and routing — together cost less than the SMS bill alone.

The line that isn't purely AWS

The SMS line deserves a caveat. AWS prices outbound SMS per message, and the exact rate depends on the destination country and the mobile carrier — a UK mobile is a few pence, other countries differ, and some routes add carrier surcharges. The $0.60 here is a UK-leaning estimate for around 150 outbound

texts; your real number will track your country and your provider. If your telephony provider (rather than SNS) carries the outbound SMS, that cost moves onto their invoice instead and off this table — but it's the same handful of pence per text either way. Either way, SMS is the one line worth watching as volume grows, which is exactly why the AWS Budgets alarm sits on top of the whole thing.

What ten times the volume costs

Push this to a busy business — 1,500 missed calls a month, ten times the volume — and the bill lands near $14, not $22. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules stay at a cent, and AWS Budgets stays free. What scales is the genuinely usage-priced work — roughly $6 of SMS for ten times the texts, about $4.50 of Bedrock, and a few dollars more spread across DynamoDB, logs, SES, and Lambda. Even then, the two things that reach the customer dominate, and all the machinery in between stays close to free.

Fig 6. The monthly bill at ten times the base volume, about 1,500 missed calls. SMS and Bedrock are the bulk of it; the fixed lines stay put, so the total grows sub-linearly — near $14, not ten times $2.20.

The honest way to read this: the AWS bill is rounding error against what a missed call is worth. A single missed customer at a salon, a garage, or a dental practice is worth far more than $2.20, and this design catches them by the dozen. Even at $14 a month for a busy shop, the system pays for itself the first time it turns one ring-out into a booking — and the few callers who don't reply still get a human ringing them back, with the whole story already gathered.

DESIGN RULES THAT SHAPED THE COST

- Pay per missed call, not per hour. No always-on compute means no idle bill.
- Spend the model sparingly. One Haiku call per text-back, and only to phrase — never to decide or to route.
- Cheap work stays cheap. Catching, matching, routing, and sweeping are plain Lambda and DynamoDB, cents at this scale.
- Know your one fixed cost. Secrets Manager is the only line that bills while the system sleeps.
- Watch the SMS line. It’s the part that scales and the part whose price varies, so the Budgets alarm sits right on top of it.

PART 7 OF 7

JULY 1, 2026 PART 7 OF 7 · MISSED-CALL TEXT-BACK SERIES ~7 MIN READ

Engineering reference: the missed-call text-back architecture

This is the missed-call text-back with the friendly labels removed: the real resource names, the runtime, the table key schemas, the single public Function URL, the escalation schedule, and the IAM scope. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Six Lambda functions, all Python 3.14 on arm64, wired through one SQS queue with a dead-letter queue.
- One public surface: a single Lambda Function URL on `mctr-webhook` that takes both missed-call events and inbound replies — no API Gateway.
- Five DynamoDB tables, all on-demand: calls, threads, customers mirror, opt-out list, and an append-only audit log.
- One EventBridge Scheduler with two rules: the 15-minute no-reply sweep and the customer-sheet sync.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by the replier. Single region, `eu-west-2`.

The architecture, for engineers

This is the same system as Part 1 with the friendly labels removed and the real resources named. Everything is in one region, `eu-west-2` (London), in one account. There is no API Gateway, no NAT Gateway, and nothing always-on; the only inbound surface is one Lambda Function URL, outbound SMS goes through SNS, and work is buffered on a single SQS queue.

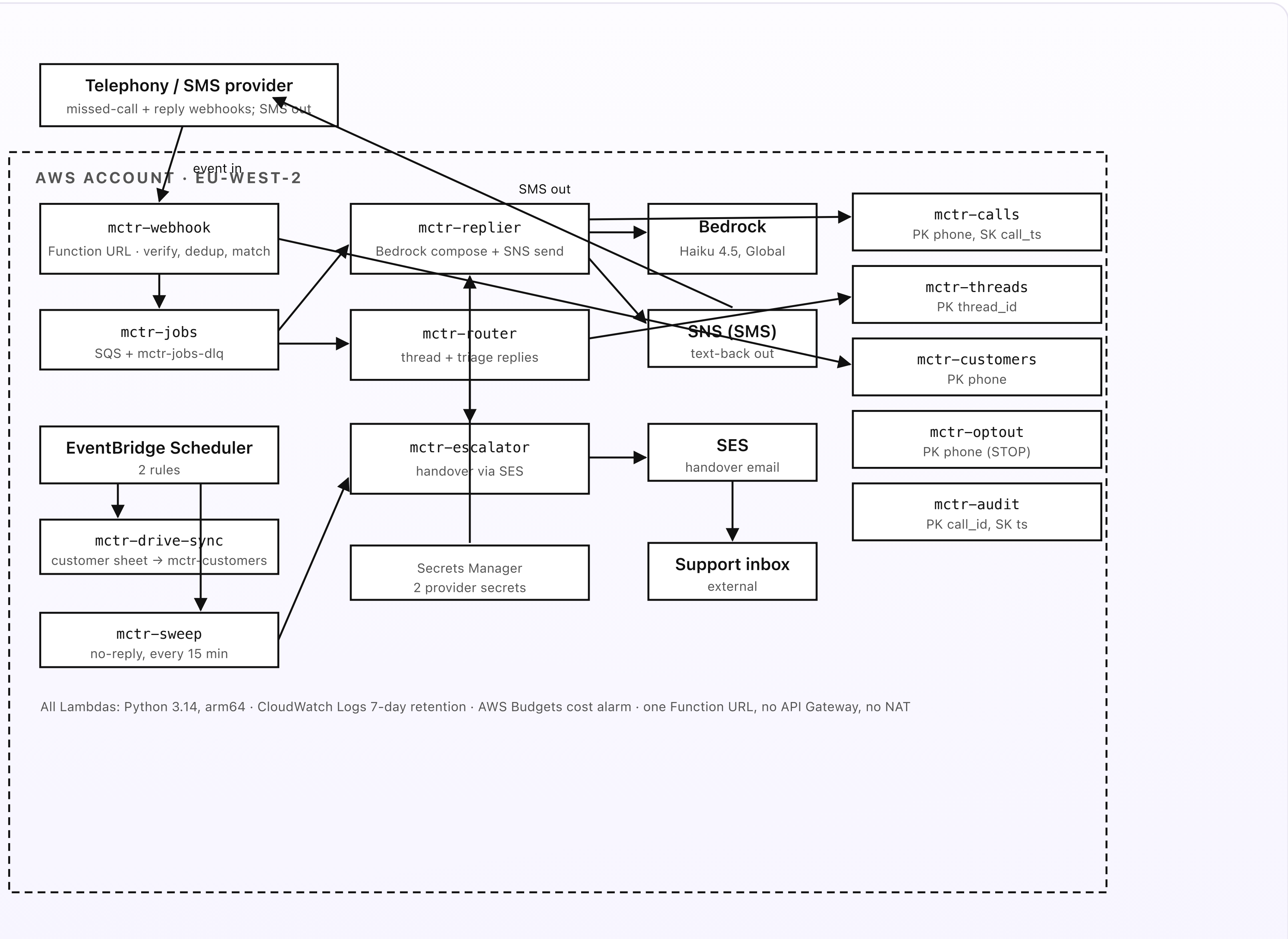

Telephony / SMS provider
missed-call + reply webhooks; SMS out
event in
AWS ACCOUNT · EU-WEST-2
SMS out
mctr-webhook
Function URL · verify, dedup, match
mctr-replier
Bedrock compose + SNS send
Bedrock
Haiku 4.5, Global
mctr-calls
PK phone, SK call_ts
mctr-jobs
SQS + mctr-jobs-dlq
mctr-router
thread + triage replies
SNS (SMS)
text-back out
mctr-threads
PK thread_id
mctr-customers
PK phone
EventBridge Scheduler
2 rules
mctr-escalator
handover via SES
SES
handover email
mctr-optout
PK phone (STOP)
mctr-drive-sync
customer sheet → mctr-customers
Secrets Manager
2 provider secrets
Support inbox
external
mctr-audit
PK call_id, SK ts
mctr-sweep
no-reply, every 15 min
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm · one Function URL, no API Gateway, no NAT

Fig 7. The missed-call text-back drawn for engineers: one Function URL on `mctr-webhook`, an SQS-buffered set of six Lambdas, five DynamoDB tables, Bedrock called only by the replier, SNS for SMS and SES for handover, and two scheduled jobs. One region, one account, no API Gateway.

Lambda functions

Six functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. Each does one job and hands off; the SQS queue (`mctr-jobs`, with `mctr-jobs-dlq` as its dead-letter queue after five attempts) decouples the public webhook from the slower model and SMS calls.

- `mctr-webhook` — the only public surface. Backs the single Lambda Function URL and handles both event types from the provider. For a missed call: verifies the signature, de-duplicates against `mctr-calls` with a conditional write, checks `mctr-optout` and quiet hours, matches the caller against `mctr-customers`, and enqueues a compose job. For an inbound reply: enqueues a routing job. Nothing slow happens here.
- `mctr-replier` — SQS-triggered on compose jobs. Makes the single Bedrock call, validates the draft, injects the booking link, sends one SMS via SNS, and writes the result to `mctr-calls` (state `texted`) and `mctr-audit`.
- `mctr-router` — SQS-triggered on routing jobs. Matches the reply to its open call in `mctr-calls`, appends to `mctr-threads`, handles `STOP` by writing `mctr-optout`, and assigns or escalates by keyword-and-recency triage.
- `mctr-escalator` — builds the handover (call + matched customer + text-back + reason), de-duplicates against open threads, and emails the support inbox via

SES.

- `mctr-drive-sync` — scheduled. Pulls the customer sheet from Drive and upserts rows into `mctr-customers`.
- `mctr-sweep` — scheduled. Queries `mctr-calls` for `texted` calls past the no-reply window with no reply and not yet escalated, hands each to `mctr-escalator`, and marks the call `escalated` so it's only flagged once.

Data stores, schedules, and messaging

- **DynamoDB (all on-demand).** `mctr-calls` — PK `phone` (E.164), SK `call_ts`; one item per missed call with its dedup marker, text-back, and state (`texted` / `replied` / `escalated`), queried newest-first to match replies. `mctr-threads` — PK `thread_id`, the conversation and assignment state. `mctr-customers` — PK `phone`, the directory mirrored from the sheet. `mctr-optout` — PK `phone`, the STOP suppression list checked before every send. `mctr-audit` — PK `call_id`, SK `ts`, append-only, holding each text-back and the facts it was built from.
- **Function URL.** One, on `mctr-webhook`, with provider signature verification in-function; `AuthType NONE` at the edge because authenticity is enforced by the shared secret, not by IAM. No API Gateway.
- **SNS and SES.** SNS sends the outbound text-back (or the provider's own SMS API does, if you route SMS through them); SES sends escalation and handover email to the team from a verified domain with DKIM.
- **EventBridge Scheduler.** Two rules — `mctr-sweep` at `rate(15 minutes)` gated to opening hours, and `mctr-drive-sync` at `rate(15 minutes)`.

- **Secrets Manager.** Two secrets — the webhook signing secret and the SMS/provider API key — fetched at call time, never in env vars or the sheet.
- **Bedrock.** Model id `anthropic.claude-haiku-4-5` via the Global cross-Region inference profile, invoked only by `mctr-replier`.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `mctr-webhook` can read `mctr-customers` and `mctr-optout`, conditionally write `mctr-calls`, read the signing secret, and send to `mctr-jobs` — it cannot call Bedrock or SNS. `mctr-replier` is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile; it can publish to SNS and write `mctr-calls` and `mctr-audit`, but cannot delete from any table. `mctr-router` can write `mctr-threads` and `mctr-optout` and invoke the escalator, nothing more. `mctr-escalator` can send via SES and read its inputs only. The scheduled functions hold the narrow Drive and table permissions they need and no inbound surface at all. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call for capacity and is not a data store. An AWS Budgets alarm watches the monthly spend — with SMS the line most likely to move, it's the cheapest early warning that volume (or a loop) is running hot.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Six small Lambdas beat one that does everything; the queue decouples the slow calls from the webhook.
- One public surface. Only `mctr-webhook` is reachable from outside, on a single Function URL, authenticated by a shared secret.
- Least privilege, per role. Only the replier can call Bedrock and SNS; only the webhook and router touch the opt-out list.
- State in DynamoDB. Tables for calls, threads, customers, opt-out, and audit; the calls table's state field drives the sweep.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called once per text-back.
- A budget alarm is a smoke detector. With SMS the variable line, a Budgets alert is the cheapest way to catch a runaway.