7-PART SERIES · FREE COMPANION

Invoice chaser

A serverless chaser that watches which invoices are unpaid and past due; sends a polite reminder sequence to the right contact at the right cadence; escalates to the account owner when an invoice goes seriously late; and stops the moment payment lands. A human stays in control — nothing rude or wrong ever goes out. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Invoice chaser

PART 1 OF 7

MAY 6, 2026 PART 1 OF 7 · INVOICE CHASER SERIES ~5 MIN READ

An invoice chaser on AWS for a few dollars a month

A small business sends more invoices than anyone keeps in their head. The net-30 that went out three weeks ago and is now quietly two days late. The big project invoice the client keeps meaning to pay. The repeat customer who always pays, just always a week late. The one nobody followed up on because the person who would have was on holiday. Chasing money you are owed is awkward, easy to forget, and the first thing that slips when the week gets busy. This post walks through the design of a small chaser that watches every unpaid invoice, sends a polite reminder to the right person at the right time, and escalates if it goes seriously late — and stops the second the money lands.

KEY TAKEAWAYS

- Three sources for invoices: a Drive sheet from your accounting tool, an inbox forwarding lane, and a webhook lane.
- Every invoice ends in one of four moves on each tick: current, first nudge, follow-up, or escalate.
- Per-terms cadences: net-30 gets a nudge at 3 days late, a follow-up at 10, an escalation at 21.
- Reminders climb a friendly-to-firm tone ladder, respect quiet hours and weekends, and carry a pay-link.
- Designed on AWS for about $2.40/month at typical small-business volume.

The whole system on one page

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

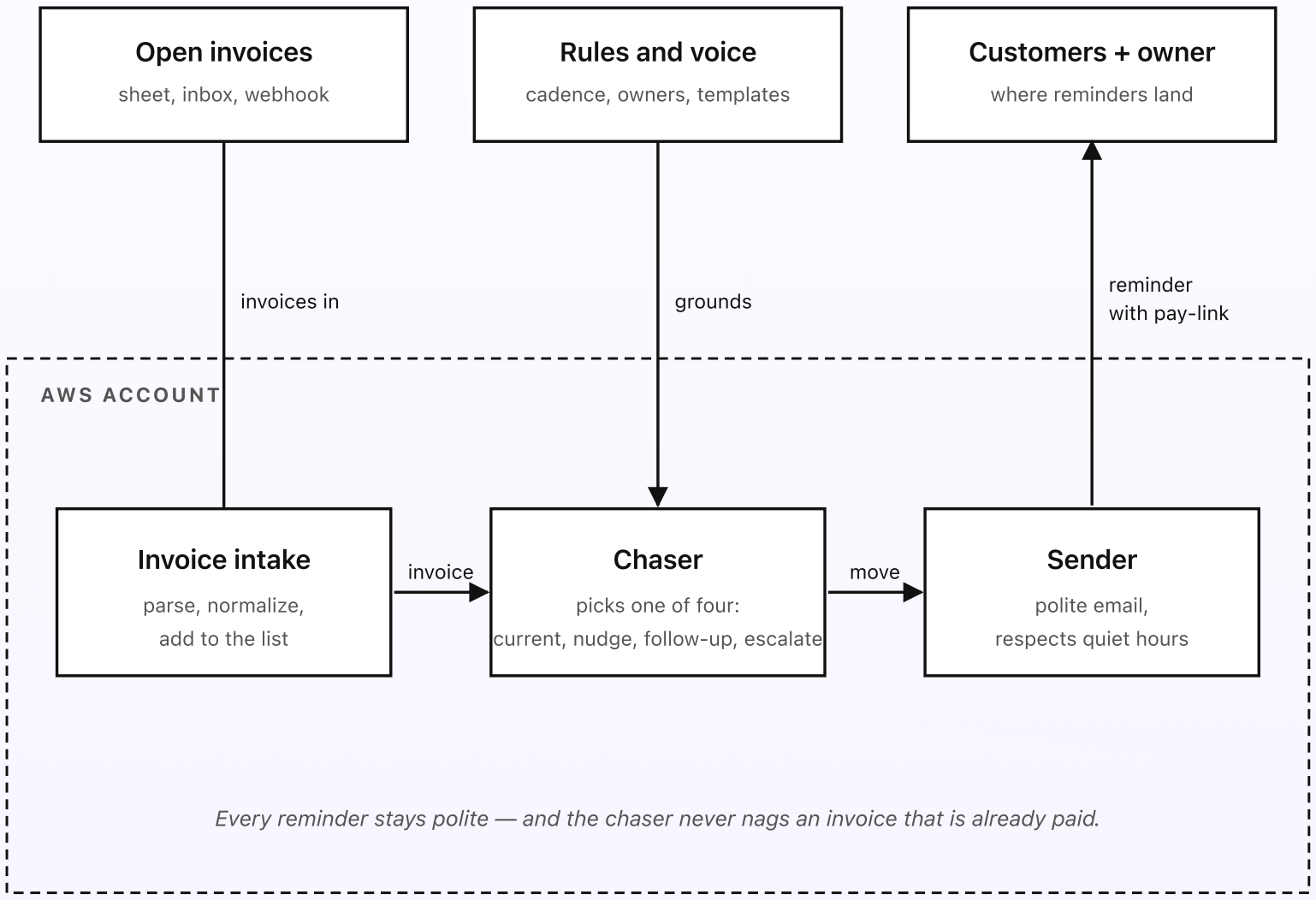

Open invoices
sheet, inbox, webhook
Rules and voice
cadence, owners, templates
Customers + owner
where reminders land
invoices in
grounds
reminder
with pay-link
AWS ACCOUNT
Invoice intake
parse, normalize,
add to the list
invoice
Chaser
picks one of four:
current, nudge, follow-up, escalate
move
Sender
polite email,
respects quiet hours
Every reminder stays polite — and the chaser never nags an invoice that is already paid.

Fig 1. Three sources outside, three pieces inside AWS. Invoices flow in from a Drive sheet, an inbox forwarding lane, and a webhook lane. The Chaser runs daily and picks one of four moves. The Sender emails the right reminder to the right person at the right time.

What you set up once (the outside)

- **Open invoices.** A Google Sheet in a Drive folder, one row per invoice: number, customer, billing contact email, amount, issue date, due date, terms (net-15, net-30, due-on-receipt), a paid flag, and a link to the invoice PDF. You export this from your accounting tool once and let new invoices flow in from two other lanes covered in Part 2 — an inbox-forwarding lane (forward an invoice PDF to a dedicated address and the chaser proposes a row for one-tap approval) and a webhook lane (your accounting tool calls a small endpoint the moment an invoice is issued).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the reminder cadence for each set of terms — how many days past due the chaser should send, and how many times. Net-30 typically gets a nudge at 3 days late, a follow-up at 10, and an escalation at 21; due-on-receipt gets a tighter 1/7/14. The doc also lists the account owner per customer (or per invoice, if it overrides), the escalation target if the invoice stays unpaid, the quiet hours, and any holiday days to skip. The *voice* doc holds one reminder template per step of the tone ladder — the friendly nudge, the firmer follow-up, the final notice.
- **Customers and owner.** The billing contact at each customer receives the reminders. The internal account owner receives the escalation when an invoice goes seriously late. Reminders land with the invoice number, amount due, days

past due, a link to the invoice PDF, and a pay-link the customer can click to settle on the spot.

What runs on every tick (the inside)

- **The invoice intake.** Three sources feed the list. The Drive sheet is the canonical store. New invoices can also be added via the inbox forwarding lane (forward a PDF to `invoices@your-company.com`, the chaser uses Textract to read the PDF and Bedrock Haiku 4.5 to extract number, customer, amount, due date, then drops a one-tap approval card in the bookkeeper's Slack to confirm before the row is added) and the webhook lane (your accounting tool posts new invoices to a small endpoint the moment they are issued).
- **The chaser.** Runs once a day at 9am local. Reads the open invoices. For each one, computes days-past-due. Compares against the per-terms cadence in the rules doc. Picks one of four moves. *Current*: not yet due, or paid — do nothing. *First nudge*: just crossed the first cadence step — send a friendly reminder with the pay-link. *Follow-up*: crossed a later step with no payment — send a firmer reminder, mention when the previous one went out. *Escalate*: hit the final step with no payment — tell the account owner named in the rules doc; log it. The chaser itself doesn't call a model on the daily tick — the move logic is plain Python.
- **The sender.** Reads the voice doc, formats the reminder message for the chosen move and tone, and sends it. Email goes through SES outbound. It honors quiet hours (no sends between 6pm and 8am local by default) and skips weekends and holidays. Every send writes a row in DynamoDB so the next day's tick can tell whether a reminder already went out. A weekly digest summarizes everything that went out that week, plus what's coming up. A monthly summary

writes a one-paragraph cash-flow narrative: total outstanding, oldest invoices, biggest at-risk amounts.

In plain words

You invoiced Acme Co. $6,400 on net-30 terms. The due date passes and nobody at Acme has paid. On day 3 past due, the chaser emails their billing contact a friendly nudge: “Just a quick reminder — invoice #1042 for $6,400 was due on May 1. Here’s a link to pay or to the PDF if you need it.” No reply. On day 10 it sends a firmer follow-up that mentions the first one. Still nothing. On day 21 it escalates to your account owner, Sam, with the full history attached, so a human can pick up the phone. Meanwhile, the moment Acme pays — whether from the pay-link or a bank transfer your accounting tool records — the chase stops on the next tick and a short thank-you can go out. Nobody at Acme gets nagged about money they have already sent.

The cost of running this is about $2.40 a month at SMB volume. The cost of *not* running it is the invoice that ages 90 days because everyone was busy, the cash-flow crunch that follows, and the awkward call that gets harder the longer it waits.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every reminder ships with full context — invoice number, amount, days past due, pay-link, PDF. The customer never has to dig.
- Four moves, always. Current, first nudge, follow-up, escalate. There is no fifth.
- The tone ladder climbs from friendly to firm. Nothing rude ever goes out, even on the final notice.
- Quiet hours, weekends, and holidays are respected. A reminder is a finite resource; bad timing burns it.
- Payment stops the chase on the next tick. The chaser never nags an invoice that is already paid.
- Every send and every owner action is logged. Audit a collection next year and you can see every reminder that went out.

Why this shape

Most teams chase invoices in one of three ways: a person who remembers to do it when they have time, a spreadsheet of “who owes us” that nobody opens, or not at all. The person works until they are busy — and the weeks they are busiest are exactly the weeks cash is tightest. The spreadsheet is a list, not a system: it tells you who is late but does nothing about it. And “not at all” is how a healthy business ends up quietly lending its customers money for free.

The setup above keeps the invoice list in a sheet the team already exports, but adds a small system that *looks at* that list every day and acts only when something is actually overdue. Reminders go out early enough to matter. They are polite by default and only firmer when an invoice is genuinely late. They carry a pay-link so paying is one click. They escalate to a human cleanly when it is time for a phone call. And they stop the moment the money arrives. The chaser is invisible on the days nothing is owed; it only shows up when someone owes you money and has forgotten.

The next four posts walk through each piece in turn: how an invoice gets loaded, how overdue gets detected, how a reminder reaches the customer, and how a chase stops on payment. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 6, 2026 PART 2 OF 7 · INVOICE CHASER SERIES ~4 MIN READ

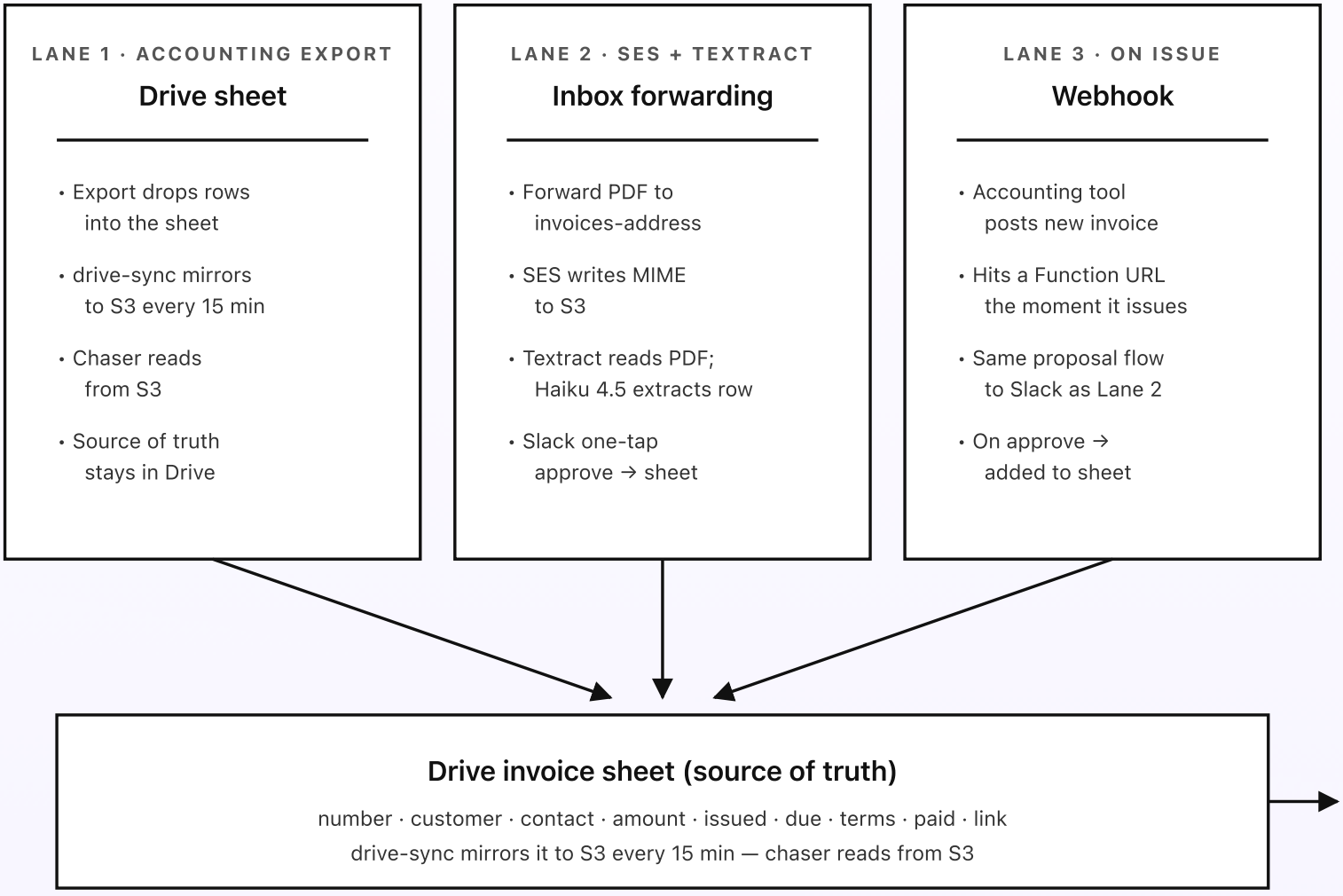
How an invoice gets loaded

The chaser only chases what's on the list. So the first job is making sure the list actually reflects what your customers owe you. There are three ways an invoice gets on: it comes across from your accounting tool into the Drive sheet, somebody forwards an invoice PDF to a dedicated address, or your accounting tool calls a small endpoint the moment the invoice is issued. The first one is the backbone. The other two exist because in real life invoices get created in a dozen places and nobody types a row in a sheet for the one they just sent.

KEY TAKEAWAYS

- Three intake lanes feed one list: the Drive sheet, an inbox-forwarding lane, and a webhook lane.
- Inbound PDFs are parsed by Textract; Bedrock Haiku 4.5 reads the text and proposes a row.
- Every parsed row goes to the bookkeeper’s Slack for one-tap approval before it lands on the list.
- The webhook lane picks up new invoices from your accounting tool the moment they are issued.
- The Drive sheet stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one list



The Drive sheet stays the source of truth — the other lanes are conveniences that propose rows for it.

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the inbox lane and the webhook lane are conveniences that propose rows for approval. The drive-sync Lambda mirrors the sheet to S3 so the chaser can read it without hitting Drive on every tick.

Lane 1: the Drive sheet from your accounting tool

The backbone lane. Most accounting tools can export an open-invoices report to a Google Sheet, or you paste one in on a schedule. The columns are short: number, customer, billing contact email, amount, issue date, due date, terms, a paid flag, and a link to the invoice PDF. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://ic-invoices-source/invoices.csv` if the sheet has changed since the last sync. The chaser reads from S3, not Drive directly. That keeps Drive API calls predictable and gives you S3 versioning for free, so a bad bulk-edit can be rolled back in one click.

This lane covers the steady state: your accounting tool is the system of record, and the sheet is just the window the chaser looks through. Most invoices arrive this way.

Lane 2: inbox forwarding (for invoices that live outside the tool)

Set up a dedicated inbound address — something like `invoices@your-company.com` — via Amazon SES. Anyone on the team forwards an invoice PDF to that address and the chaser takes it from there. SES writes the raw MIME to `s3://ic-raw-mime/`. The S3 PUT triggers a parser Lambda. The Lambda walks the MIME tree to the PDF attachment, runs Amazon Textract on it (Textract reads PDF, PNG, JPEG, and TIFF natively; if somebody forwards a Word document, the

parser falls back to `python-docx`), and gets back the extracted text plus any tables.

Then a Bedrock Haiku 4.5 call reads the text and emits a structured row: invoice number, customer, amount, issue date, due date, terms, and a billing-contact guess based on the "To" line of the original forward. The model prompt is short: "Extract a row for the invoice list. Return JSON only. Mark each field with a confidence score. Do not invent an amount or a date that isn't in the text." The output goes to a small Slack interactive message that pings the bookkeeper: the proposed row, the confidence per field, and three buttons — *approve*, *edit*, *discard*. On *approve*, a Lambda writes the row to the Drive sheet via the Sheets API. On *edit*, the bookkeeper gets a fillable modal pre-populated with the proposal. On *discard*, the message is logged and the PDF moved to a discarded prefix in S3 for audit.

The reason every parsed row goes to a human first is simple: chasing a customer for the wrong amount is far worse than an invoice that never made it onto the list. A misread amount or a wrong contact turns a polite reminder into a complaint — and an awkward apology.

Lane 3: the webhook lane

Most accounting tools can call a URL when an invoice is created. That is the cleanest lane of all: the invoice lands on the list the same minute it is issued, with no export delay and no forwarding. Your accounting tool posts the new invoice to a Lambda Function URL — the same kind of small endpoint used for the pay-links later in the series. The endpoint checks a shared secret, maps the tool's fields to the chaser's row shape, and runs the same Slack proposal flow as Lane 2.

For a trusted source you can skip the approval step and let webhook rows land directly, since the data comes straight from your own system of record rather than a forwarded PDF that had to be read. The webhook lane is the most opt-in of the three: a team whose tool can't call a URL just leans on Lane 1 and loses nothing; a team whose tool can gets near-instant loading for free.

Why the sheet stays the source of truth

Three lanes in, but only one place where the chaser actually looks. That's a deliberate constraint. If two lanes both wrote directly to the chaser's state, every "why did this reminder go out?" question would mean checking three places. Funneling everything through the Drive sheet means there is exactly one row per invoice, and any team member can read or edit any of it without learning a new tool. The convenience lanes are first-class for getting invoices in, but they always pass through the sheet on the way.

Next post: how the chaser actually reads the list, computes days-past-due, and picks one of four moves.

PART 3 OF 7

MAY 6, 2026 PART 3 OF 7 · INVOICE CHASER SERIES ~5 MIN READ

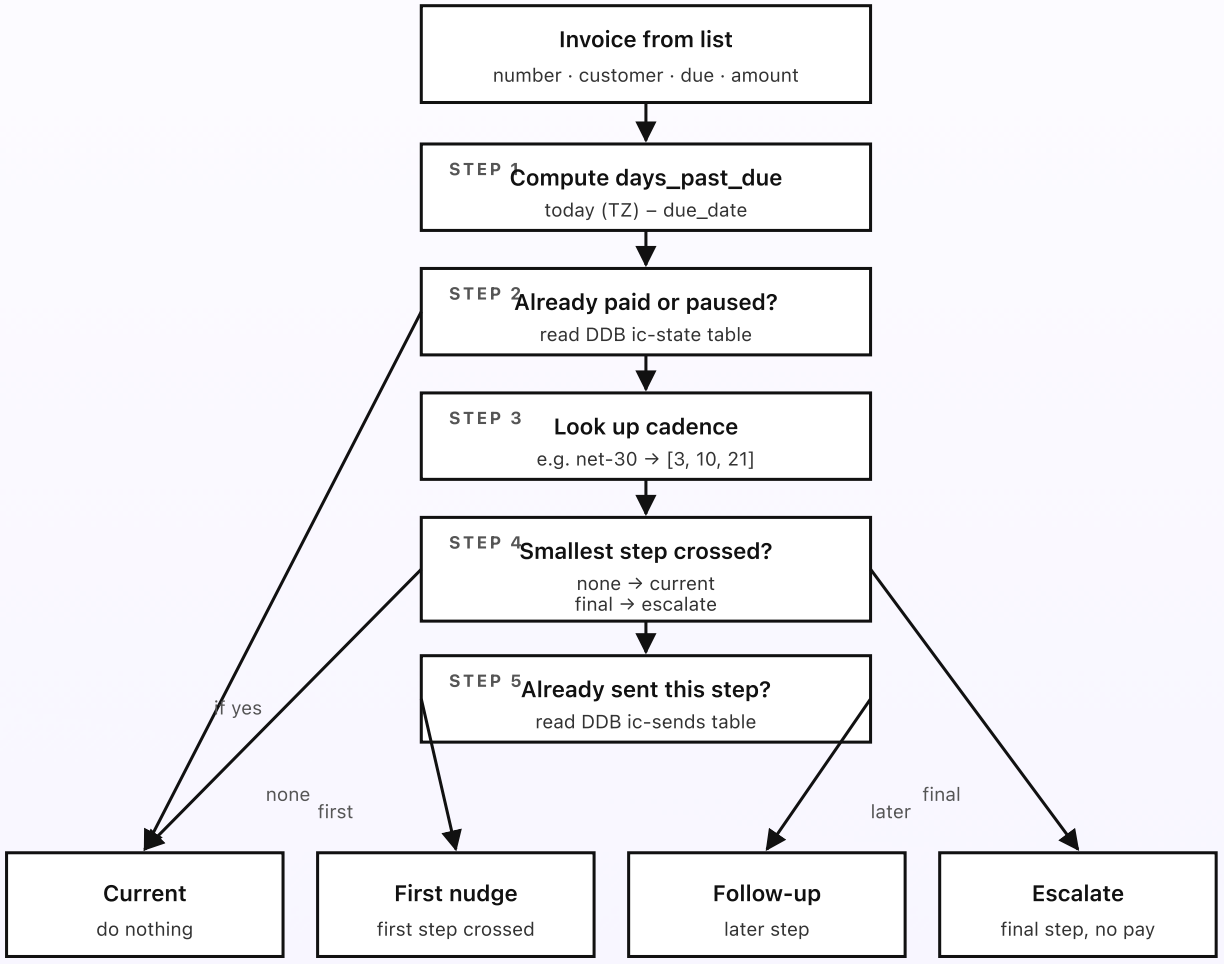
How overdue gets detected

Once a day, at 9am local time, an EventBridge Scheduler rule fires the chaser Lambda. The Lambda reads the invoice list, looks at one row at a time, computes how many days past due it is, and decides whether to do nothing or to send a reminder — and if so, which kind. The whole decision is plain Python. No model. No vector retrieval. Every threshold lives in the rules doc, where the bookkeeper can edit it without a deploy.

KEY TAKEAWAYS

- The chaser runs once a day via EventBridge Scheduler at 9am local time.
- Per-terms cadences live in the rules doc — net-30 gets 3/10/21, net-15 gets 2/7/14, due-on-receipt gets 1/7/14.
- Four moves per invoice, every tick: current, first nudge, follow-up, escalate.
- DynamoDB tracks last-send and paid/pause state per invoice so reminders aren't duplicate spam.
- The chaser itself never calls a model. The decision is entirely deterministic.

The decision flow, per invoice



The rules doc holds every threshold — change a cadence and tomorrow's tick uses the new value.

Fig 3. The chaser's decision tree, per invoice, per daily tick. Five steps decide which of four moves applies. The rules doc holds every threshold; the chaser only enforces them.

Cadences: 3/10/21 isn't magic, it's in the doc

The rules doc has one short section per set of terms. Each section names the cadence in plain prose: "Net-30: nudge at 3 days late, follow up at 10, escalate at 21. Net-15: 2, 7, 14. Due-on-receipt: 1, 7, 14. Net-60: 7, 21, 45." The numbers are days past due when the reminder fires. The first number is the friendly nudge. The last number is the escalation point — if the invoice still isn't paid by then, the account owner gets pulled in.

The cadences exist for a reason. A 3-day net-30 nudge is early enough to feel like a courtesy, not a demand — most invoices that age a few days are just sitting in someone's approval queue. A 10-day follow-up is firm enough to get noticed without burning the relationship. A 21-day escalation is where a human should be picking up the phone. Different terms imply different patience; the cadences reflect that.

Per-invoice overrides exist too. The invoice sheet has an optional column called `cadence_override`. Type a comma-separated list of days there and the chaser uses your numbers instead of the terms default for that one row. This is the right escape hatch for the big client you've agreed to give a longer leash, or the chronic late-payer who needs a tighter one.

Four moves, always

Every invoice, every tick, lands in exactly one of four buckets. The names are simple on purpose.

- **Current.** The invoice isn't past the first cadence step yet, or it has been paid or paused. Do nothing. Most invoices, most days, are current.
- **First nudge.** The invoice just crossed the first cadence step and there's no payment yet. Send a friendly reminder with the pay-link. Write a row to the `ic-sends` DynamoDB table marking that the first step has fired.
- **Follow-up.** A later step crossed without payment. Send a firmer reminder that names the previous reminder's date so the customer doesn't feel like it's the first they're hearing of it. Write the new send to `ic-sends`.
- **Escalate.** The final step in the cadence crossed without payment. Tell the account owner named in the rules doc — usually the salesperson or finance lead for that customer — with the full history attached so a human can take it from here. Mark the invoice as escalated in DynamoDB. After escalation the chaser stops auto-sending to the customer; this is the point where a person, not a system, should be in the conversation.

State that makes the decision deterministic

The chaser reads two DynamoDB tables every tick. `ic-sends` records every reminder that's gone out: `(invoice_id, step_index, send_date, dispatched_via)`. `ic-state` records the live status of each invoice: `(invoice_id, status)` where status is open, paid, paused, disputed, or written-off, plus a `paused_until` when relevant. With those two tables, the move-decision logic is a few dozen lines of Python and zero magic. A given invoice with

a given due date, a given cadence, and a given send/status history always produces the same move. Re-running the tick produces no extra reminders, because the state in DynamoDB shows what already fired.

Marking an invoice paid is an explicit stop: rows for the cadence are kept for audit, and the live state flips to paid so no further reminders go out. Part 5 covers the stop-on-payment flow in detail.

Why the daily tick uses no model

The chaser could call a model on the tick to write a smarter reminder, or to decide whether to send at all. It doesn't. Two reasons. First, the daily tick should be the one part of the system that is utterly predictable — if the rules doc says nudge at 3 days late and there's no payment, the nudge fires. A model in that loop introduces variance the team can't reason about, and chasing money is exactly where surprises are unwelcome. Second, model calls cost money, and most days most invoices are current, so the call would be wasted nine days out of ten.

Bedrock fires elsewhere — on the inbound parsing lane in Part 2, and on the monthly cash-flow summary mentioned in Part 6. Not on the daily tick. The chaser itself is plain Python that reads a doc and writes events.

Next post: how a reminder reaches the customer, how quiet hours and weekends are honored, and how the tone ladder keeps every message polite.

PART 4 OF 7

MAY 6, 2026 PART 4 OF 7 · INVOICE CHASER SERIES ~5 MIN READ

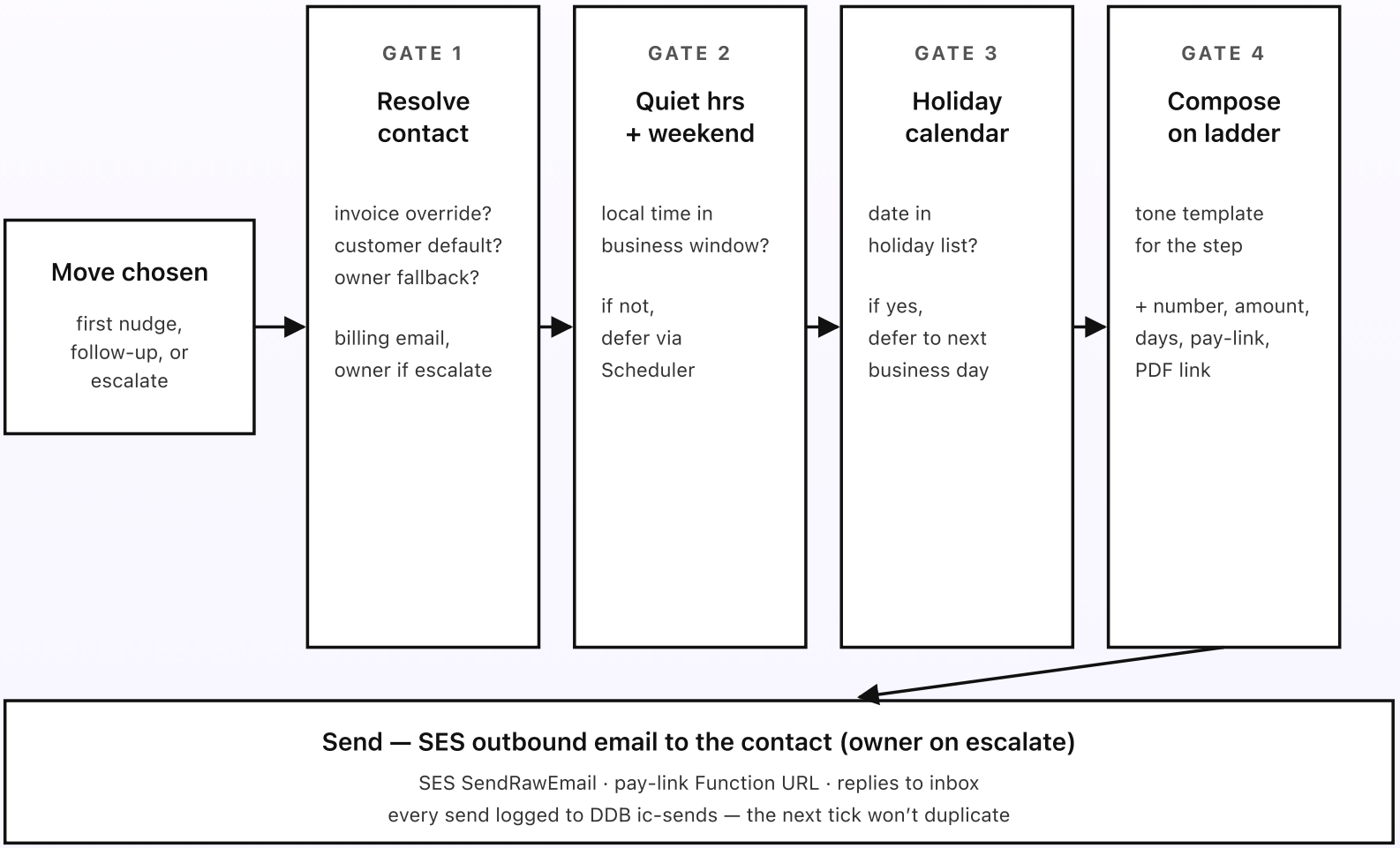
How a reminder reaches the customer

The chaser picked a move — first nudge, follow-up, or escalate. Now the sender Lambda has to figure out who to email, in what tone, at what time of day, and with what attached. Get any of those wrong and the reminder is worse than no reminder: a Saturday email, a blunt “you owe us money,” a reminder to a contact who left the customer last year. Four small guardrails sit between the move and the actual send.

KEY TAKEAWAYS

- Contact resolution: per-invoice override beats per-customer default beats fallback to the account owner.
- The tone ladder climbs from friendly to firm — the right template for the step, never rude.
- Quiet hours, weekends, and holidays defer a send to the next available business hour.
- Every reminder ships with the invoice number, amount, days past due, a pay-link, and the PDF.
- Escalation goes to the account owner with the full history; the customer isn't auto-emailed again.

Four guardrails on every send



Every gate is a deterministic check — no model calls, no rude surprise on a Saturday in April.

Fig 4. Four guardrails between the move and the sent reminder. Resolve the contact. Honor quiet hours and weekends. Skip holidays. Compose on the tone ladder with full context. Then send via SES and log it so the next tick doesn't duplicate.

Gate 1: resolve the contact

Three places the sender Lambda looks for who to email, in order. First, the invoice sheet's per-invoice `contact_email` column — if a row has a specific billing contact, that person gets the reminder regardless of the customer default. Second, the per-customer default in the rules doc ("all Acme Co. invoices go to ap@acme.com"). Third, the account owner fallback — the internal person who owns the relationship and gets every reminder that has no customer contact. The fallback should never fire in steady state; if it does, the weekly digest names every invoice that hit the fallback so the rules doc can be updated.

For an *escalate* move, the resolution flips: the reminder goes to the internal account owner, not the customer. That's the whole point of escalation — once an invoice has been chased to the end of its cadence with no payment, a person should take over the conversation, not the system.

Gate 2: quiet hours and weekends

The chaser itself runs at 9am local time, so the first time a move fires it's already in business hours on a weekday. But deferred sends from previous gates, and one-off computed sends, can land outside the window. Gate 2 reads the rules doc's quiet-hours setting (default 6pm to 8am, configurable per business) and the weekend rule (skip Saturday and Sunday by default).

If the current local time is in the quiet window or it's a weekend, the sender creates a one-off EventBridge Scheduler rule that fires at the next business-hour minute and exits without sending. The Scheduler invokes the same sender Lambda with the same payload at the deferred time, where Gate 2 will let it through. A payment reminder that lands at 2am Sunday reads as careless; one that lands at 9am Monday reads as a business doing its job.

Gate 3: holiday calendar

The rules doc lists the holidays you observe — either a static list ("Christmas Day, New Year's Day, Independence Day…") or a reference to a Google Calendar that holds them. Gate 3 checks the current local date against that list and, if it's a configured holiday, defers the send to the next non-holiday business day.

The list is on purpose — the chaser won't auto-detect a country's public holidays for you. The failure modes are very different. A holiday you forgot to add sends a reminder into a closed office — harmless but pointless. A holiday in the list that's no longer observed just delays a reminder by one business day, which is fine. The trade-off favors keeping the list explicit.

Gate 4: compose on the tone ladder, then send

The voice doc has one email template per step of the tone ladder. The *first nudge* is warm and short: "Just a friendly reminder that invoice #1042 for $6,400 was due on May 1 — here's a link to pay, and the PDF if you need it." The *follow-up* is firmer but still courteous, and names the date of the previous reminder. The *escalate* notice (which goes to the internal owner, not the customer) summarizes

the history so a human can pick up the phone. Every template carries the invoice number, amount due, days past due, a pay-link, and a link to the PDF. The sender fills the placeholders and ships the message via SES outbound.

The pay-link is a Function URL the customer can click to settle the invoice — it hands off to whatever payment surface your accounting tool exposes, and records that the link was opened so the chaser knows the customer engaged. The reply-to address is your real inbox, so if the customer writes back “already paid” or “can we split this,” a human sees it.

An escalate move doesn’t send the customer yet another email — the cadence is done, and piling on more automated reminders past that point reads as harassment, not diligence. Instead it hands the account owner the full picture: every reminder that went out, the dates, and the cumulative days overdue. From there it’s a person’s call.

Every send — nudge, follow-up, or escalation — writes a row to `ic-sends` in DynamoDB. The next day’s tick reads that row and knows not to send the same step again.

Why the guardrails exist

None of these gates are exotic. They’re the kind of small care a thoughtful person would take if they were chasing the invoices themselves — check who actually handles billing, don’t email at 11pm or on a Sunday, skip the day everyone’s off, stay polite even when an invoice is very late, and hand the awkward ones to a human. Putting them in code as four small sequential gates makes them part of

the design, not a feature you're trusting the writer of any one reminder to remember.

Next post: how a chase stops the moment payment lands — and the three things the owner can do when an invoice needs a human: pause, mark disputed, or write off.

PART 5 OF 7

MAY 6, 2026 PART 5 OF 7 · INVOICE CHASER SERIES ~5 MIN READ

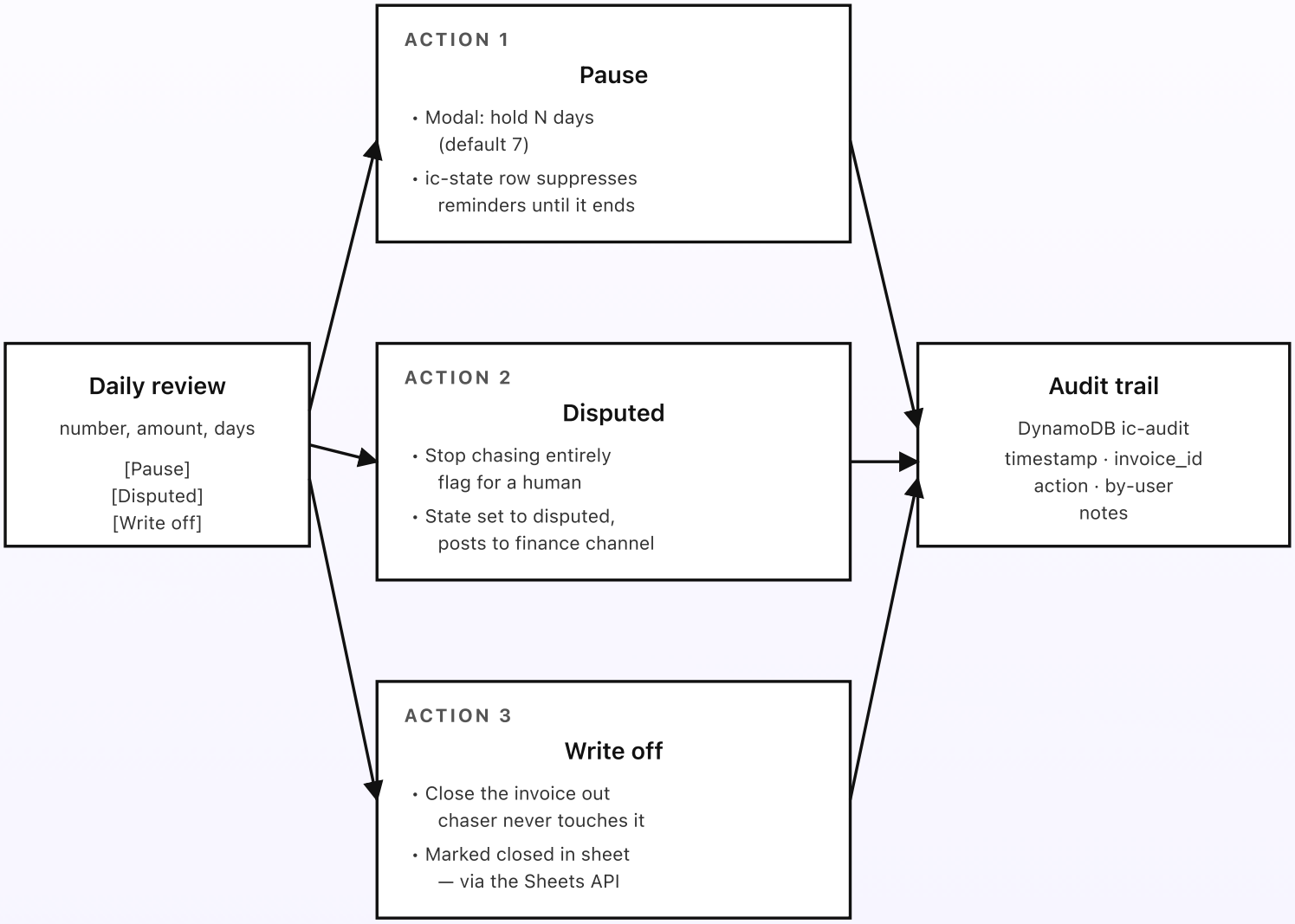
How a chase stops on payment

Acme Co. just paid invoice #1042. The most important thing the chaser can do now is the simplest: stop. No more reminders, no awkward “you owe us” email landing the morning after the money cleared. This post walks through how the chaser notices payment and shuts the chase down — and the three things the owner can do from the daily review when an invoice needs a human: pause, mark disputed, and write off.

KEY TAKEAWAYS

- Payment is detected from the paid column and the accounting feed; the chase stops on the next tick.
- Three owner actions on the daily review: *pause* (hold), *mark disputed* (stop and flag), *write off* (close out).
- Each action updates the invoice state and writes an audit row.
- Pause is bounded — you can only pause a few times before the chaser resumes anyway.
- The owner’s buttons are a Slack interactive message backed by a Function URL.

Payment stops the chase, then three owner actions



Payment stops the chase automatically — the buttons are for invoices that need a human.

Fig 5. Payment stops the chase automatically; three owner actions cover the rest. Pause holds without dismissing. Disputed stops chasing and flags a human. Write off closes the invoice. Every action — including the automatic paid stop — writes to the audit trail.

The automatic stop (the most common by far)

Most invoices end the happy way: the customer pays. The chaser learns about payment two ways. The Drive sheet has a `paid` column that your accounting export keeps up to date; when it flips to yes, the next tick reads it and sets the invoice state to paid. And if your accounting tool can post payment events to the same kind of webhook used for intake in Part 2, the state flips the same minute the money is recorded — no waiting for the next sync.

When an invoice goes to paid, three things happen, in order. First, the live cadence rows in `ic-sends` are copied to `ic-sends-archive` with a chase id, and the live chase is cleared so no further reminders fire. Second, a `action: paid` row is written to `ic-audit` with the date and the amount. Third, an optional short thank-you can go out via SES — a one-line "Thanks, we've received payment for invoice #1042" that closes the loop warmly. The customer never gets another reminder about an invoice they've already settled.

This is the path that matters most, and it needs zero human action. The buttons below are for the invoices that *don't* just get paid.

Action 1: pause (the deferral)

Some invoices aren't paid yet but also shouldn't be chased right now. The customer asked for a few extra days. The contact is on leave and the backup hasn't taken over. You're mid-conversation about the scope and don't want an automated nudge cutting across it. The owner isn't ready to stop chasing for good — they just need the chaser quiet for a week.

Pause opens a small modal asking for the number of days, with a 7-day default and a max of 14. On save, a row is written to `ic-state` with `(invoice_id, paused_until)`. The next day's tick reads that row in the "already paid or paused?" check from Part 3 and treats the invoice as current until the pause ends. When the pause ends, the chaser re-evaluates the cadence from where it was — if the invoice is now well past the escalation step, the next move is an escalation.

Pause is bounded. The rules doc has a configurable `max_pauses_per_chase` setting (default three). After that many pauses on the same invoice, further pause attempts are rejected with a "You've hit the pause cap on this invoice; please escalate or write it off" reply, and the next tick resumes normally. This is a soft constraint that exists because the most dangerous failure mode is repeatedly snoozing a debt into oblivion.

Action 2: mark disputed (stop and flag)

Sometimes the customer isn't late — they're unhappy. They say the work wasn't finished, the amount is wrong, or they never agreed to that line. Chasing a disputed invoice with cheerful reminders makes a tense situation worse and can cost you the relationship.

Mark disputed sets the `ic-state` status to disputed, which stops every reminder for that invoice immediately, and posts a note to the finance channel so a human picks it up. The expiry — sorry, the amount and due date — isn't changed; the invoice is simply taken out of the chaser's hands until somebody resolves the dispute and either marks it paid, writes it off, or clears the dispute flag so chasing can resume. A disputed invoice that later gets cleared resets cleanly on the next tick.

Action 3: write off (close it out)

Some invoices are never going to be paid. The customer folded. The amount is too small to be worth a fight. A goodwill gesture was agreed. Whatever the reason, the owner wants the invoice off the chase list for good without pretending it was paid.

Write off sets the `ic-state` status to written-off, marks the row closed in the Drive sheet via the Sheets API, and the chaser never touches it again. The amount is recorded in the audit trail as written off, not paid, so the monthly summary in Part 6 reports it honestly — a written-off invoice with a note ("written off by Sam on 2026-04-15, customer insolvent") so the trail isn't a mystery at year-end. Writing off is deliberately a distinct action from marking paid, because the two mean very different things to your books.

Every action is logged, every action is reversible

The `ic-audit` table records every paid, pause, disputed, and write-off with the user who took the action, the timestamp, and a snapshot of the row before and after. If a wrong status gets set (an invoice marked paid that wasn't, an invoice

written off in error), a team member can run an “undo last action” through a small admin command that reads the previous-state snapshot and restores the row. The undo is itself an audit row, so the trail of edits stays clean.

This kind of reversibility matters most for the money decisions you’ll be asked about later. When the accountant reconciles at year-end, or a customer insists they paid, the audit trail is the only memory anyone has of what actually happened to that invoice and who decided it.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why it’s one of the cheapest systems in the series.

PART 6 OF 7

MAY 6, 2026 PART 6 OF 7 · INVOICE CHASER SERIES ~3 MIN READ

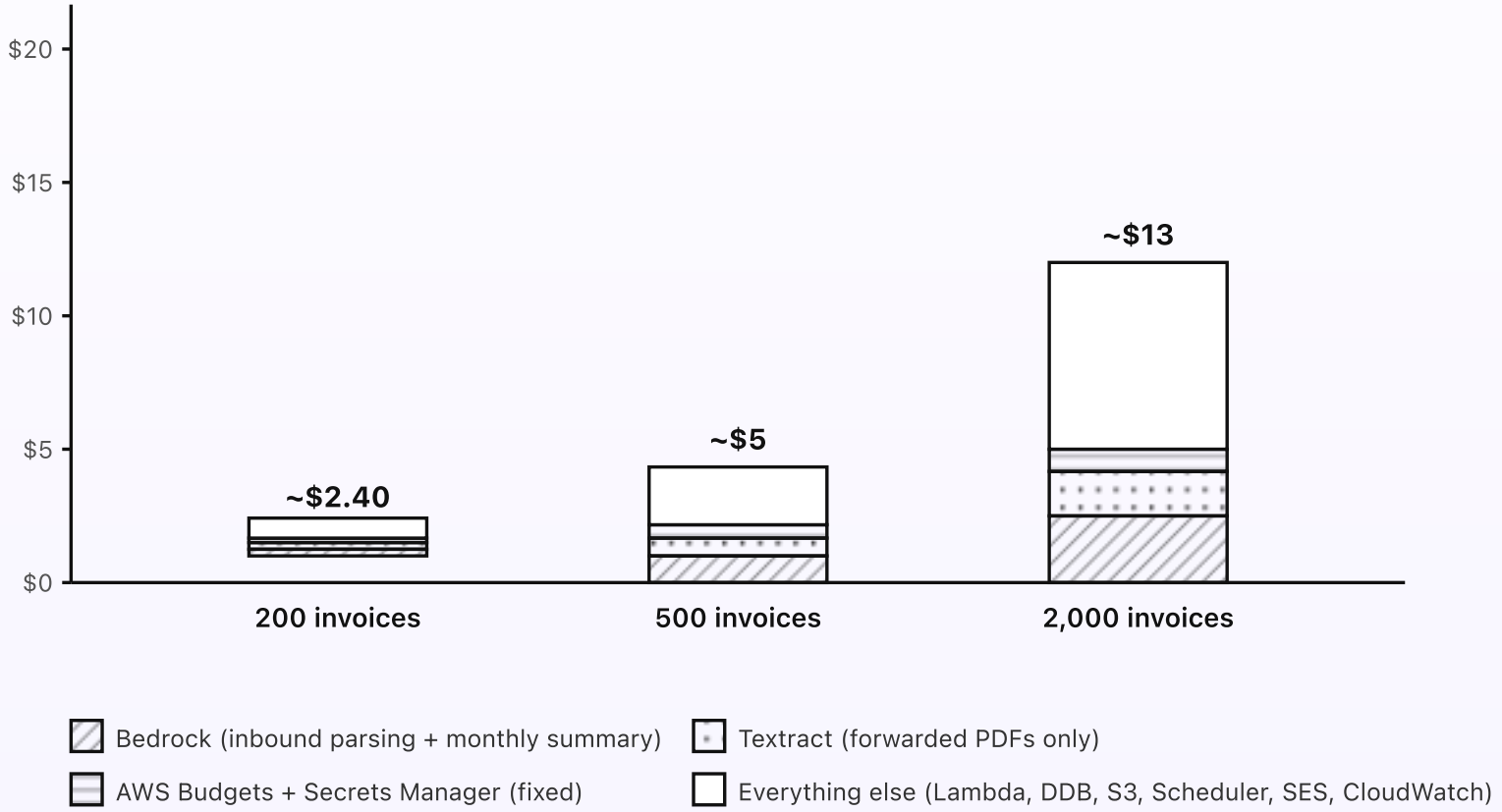
What the invoice chaser costs

The chaser is one of the cheapest systems in this whole series. The daily tick reads a CSV from S3, does some date arithmetic, writes a few rows to DynamoDB, and sends a handful of emails. It calls no models on the tick. Bedrock fires only when somebody forwards an invoice PDF and once a month for the cash-flow summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero — against the working capital it frees up by getting you paid faster.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (around 200 open invoices).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The daily tick costs pennies — no model calls.
- Bedrock fires only on inbound PDF parsing (a few times a month) and the monthly summary.
- At 500 open invoices the bill is around $5. At 2,000 invoices it's around $13.

Cost at three volumes



The daily tick is the dominant cost — and even that is fractions of a cent per invoice per day.

Fig 6. Monthly cost at three open-invoice volumes. Bedrock and Textract are small slivers because they only fire on the inbound parsing lane and the monthly summary. The dominant cost is the everything-else bucket: the daily tick reading every invoice.

Where the dollars actually go

Lambda runtime (the bulk). The chaser runs once a day. Each tick reads the invoice CSV from S3, iterates the rows, computes `days_past_due` for each, and decides on a move. At 200 invoices, that's a few hundred milliseconds. At 2,000 invoices it's a couple of seconds. Either way it's pennies a month. Add the sender Lambda firing for each reminder (around twenty to sixty sends a month at 200 invoices, a few hundred at 2,000), the Function URL Lambda for pay-links and owner actions, the webhook Lambda for new invoices, and the drive-sync Lambda every fifteen minutes — the Lambda total still lands a little over a dollar at all three volumes.

DynamoDB on-demand. Three small tables: `ic-sends`, `ic-state`, `ic-audit`. Reads are dominant during the daily tick (one read per invoice per tick, plus chase history). Writes are send events and audit rows. Pennies a month at any of these volumes.

S3 + Storage. The mirrored invoice CSV plus the archived MIME from any forwarded invoices. A few hundred KB total at SMB volume. Effectively free.

EventBridge Scheduler. The daily tick rule plus deferred send rules from quiet-hours, weekend, and holiday gates. A few invocations a day. Pennies.

SES. Inbound for the forwarding lane: $0.10 per thousand received messages (so a couple of cents a year for an SMB). Outbound for reminder emails: $0.10 per thousand sent. Even at a few hundred reminders a month, this is cents.

Bedrock (only when something fires it). The daily tick uses no Bedrock. The inbound parsing lane fires Haiku 4.5 once per forwarded PDF: a few thousand

input tokens (the Textract output) and a few hundred output tokens (the proposed row JSON), so a fraction of a cent per parse. At a few forwarded invoices a month, Bedrock costs cents. The monthly summary is one larger call: write a paragraph that summarizes the month's outstanding balance, oldest invoices, and biggest at-risk amounts; a couple of cents.

Textract (only on forwarded PDFs). Per-page pricing; a typical invoice is one or two pages. A cent or two per parse. At a few PDFs a month, Textract is a few cents. At 2,000 open invoices with twenty PDFs forwarded per month, it lands around a couple of dollars.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the pay-link, webhook, and owner-action endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The chaser sleeps 23.99 hours a day.
- **A Knowledge Base.** The invoice list is structured rows, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the tick.** The daily decision is plain Python. Bedrock fires only on the inbound parsing lane and the monthly summary.

How the cost scales

Lambda runtime grows roughly linearly with invoice count, because every open invoice is evaluated on every tick. DynamoDB grows linearly too. Bedrock and Textract are uncorrelated with invoice count — they only fire when somebody forwards a PDF or it's the first of the month. So the bill at 5,000 open invoices is around $30; at 10,000 it's around $60. Past those volumes the daily-tick model probably stops being right (you'd switch to a partial-tick that only evaluates invoices at or past their due date), but those are optimizations for very large ledgers — not redesigns.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The chaser's normal-volume bill stays well under that ceiling — and well under the cost of one invoice slipping to 90 days late.

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

PART 7 OF 7

MAY 6, 2026 PART 7 OF 7 · INVOICE CHASER SERIES ~8 MIN READ

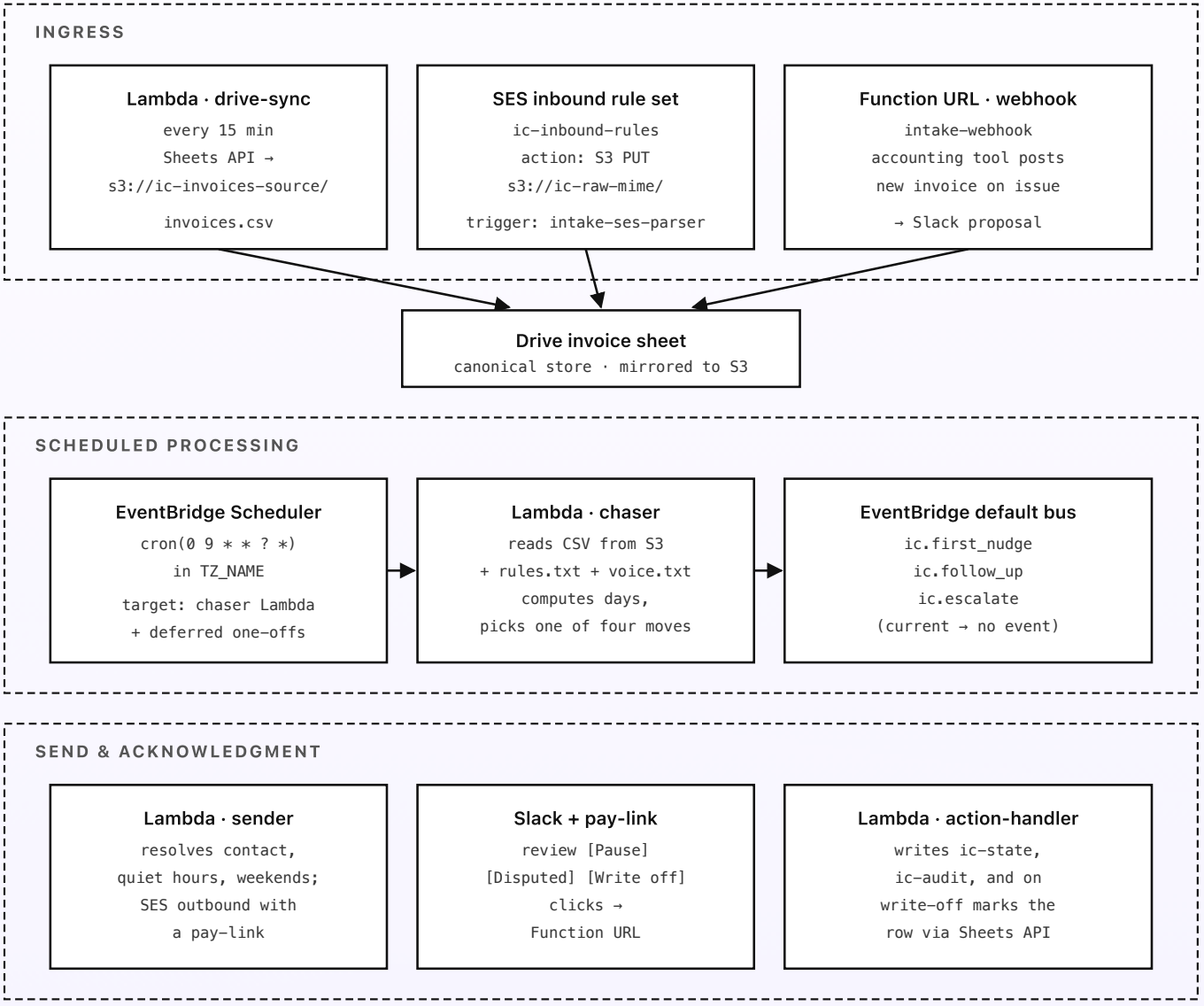
Engineering reference: the invoice chaser architecture

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

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound, Bedrock Global cross-Region inference, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is a reminder that goes out a day late, not a regional outage. One AWS account dedicated to the chaser (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every reminder leaves with a pay-link — and every interaction is logged to ic-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the invoice sheet), scheduled processing (the daily chaser tick emitting events), send and acknowledgment (the reminder ships and the owner's or customer's response is recorded). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

All Lambdas use the `arm64` architecture, the smallest memory size that meets latency targets (typically 256 MB), Python 3.14 runtime, and CloudWatch Logs at 7-day retention. Each function has its own least-privilege IAM role. None run inside a VPC.

- **`drive-sync`** — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `ic/drive/sa`) to export the invoice sheet as CSV and write to `s3://ic-invoices-source/invoices.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://ic-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`intake-webhook`** — Lambda Function URL, `AuthType: NONE` with a shared-secret check on the request (the secret lives in Secrets Manager under `ic/webhook/secret`). The accounting tool posts each newly issued invoice; the handler maps the tool's payload to the row shape and either posts a Slack proposal or, for a trusted source, writes the row directly via the Sheets API. Memory: 256 MB. Timeout: 15 s.
- **`intake-ses-parser`** — S3 PUT trigger on `s3://ic-raw-mime/`. Parses MIME, extracts the PDF attachment, runs Textract via `StartDocumentTextDetection`

+ `StartDocumentAnalysis` (asynchronously to handle multi-page invoices). On Textract completion (via SNS notification), reads the structured text and calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to propose an invoice row. Posts the proposal to Slack via `chat.postMessage` with Approve/Edit/Discard buttons. For DOCX attachments (Textract doesn't accept them), falls back to `python-docx` ; XLSX uses `openpyxl` . Both packages are stable and widely used in 2026, though their maintenance velocity is light — for an invoice-parsing path that only runs a few times a month, that's acceptable. If extraction precision becomes a concern, the active community fork `python-docx-oss` is a drop-in alternative. Memory: 512 MB. Timeout: 60 s.

- **`chaser`** — EventBridge Scheduler target, daily at 9am local time (the schedule expression runs in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). Reads `s3://ic-invoices-source/invoices.csv` and the rules and voice docs. For each row, computes `days_past_due` , reads chase state from `ic-sends` and `ic-state` , decides on a move. Emits one event per row that needs action: `ic.first_nudge` , `ic.follow_up` , or `ic.escalate` , with the invoice context as the event payload. Current invoices emit nothing. Reconciles the `paid` column against `ic-state` and archives chases for newly paid invoices. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`sender`** — EventBridge rule on the three move events. Resolves contact, checks quiet hours, weekends, and the holiday calendar, formats the reminder from the voice tone template, and ships via SES `SendRawEmail` . The escalate move resolves to the internal account owner instead of the customer. On a quiet-hours, weekend, or holiday defer, creates a one-off EventBridge Scheduler rule that re-invokes `sender` at the next available business minute.

Writes a row to `ic-sends` after a successful send. Memory: 256 MB. Timeout: 30 s.

- **`action-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on Slack-sourced requests and a signed token on pay-link requests. Triggered by Slack interactive button clicks (Pause/Disputed/Write-off) and by customer pay-link clicks. Writes to `ic-state` and `ic-audit`; on write-off, marks the row closed in the Drive sheet via the Sheets API and archives the chase in `ic-sends-archive`; on a recorded payment, flips state to paid and stops the chase. Memory: 256 MB. Timeout: 15 s.
- **`digest`** — EventBridge Scheduler target, weekly Monday 9am. Reads `ic-sends` for the past week and the invoice list; sends a digest message to a configured Slack channel summarizing reminders sent, invoices coming due, and the oldest open balances. No Bedrock; the message is a plain summary table. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `ic-sends`, `ic-state`, and `ic-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph cash-flow narrative (total outstanding, oldest invoices, biggest at-risk amounts, written-off total); emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB · `ic-sends`** — one row per reminder sent. PK `(invoice_id, step_index)`; attributes: `send_date`, `dispatched_via` (email), `recipient`, `move` (first_nudge/follow_up/escalate). On-demand. No TTL.

- **DynamoDB · `ic-state`** — one row per invoice. PK `invoice_id`; attributes: `status` (open/paid/paused/disputed/written-off), `paused_until` (if paused), `pause_count`, `paid_date` and `paid_amount` (if paid), `by_user`. On-demand.
- **DynamoDB · `ic-audit`** — one row per write action of any kind. PK `(invoice_id, ts)`; attributes: `action`, `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB · `ic-sends-archive`** — archived chases after payment or write-off. Same shape as `ic-sends`; PK `(invoice_id, chase_id, step_index)`. On-demand.
- **S3 · `ic-invoices-source`** — mirrored CSV from the Drive invoice sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3 · `ic-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3 · `ic-raw-mime`** — raw inbound MIME from forwarded invoices. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3 · `ic-source-pdfs`** — the parsed source invoices after the inbound parser handles them, kept for reference if the row links to one.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `intake-ses-parser` for the inbound invoice parsing, and `summary` for the monthly cash-flow narrative. Claude Sonnet 4.6 (`global.anthropic.claude-sonnet-4-6-20250930-v1:0`) is available as a

fallback for invoices whose layout the Haiku pass flags as low-confidence, but in practice invoices are structured enough that Haiku handles them.

- **Embeddings.** Not used. The invoice list is structured rows; deterministic lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The chaser itself doesn't call Bedrock; the parsing lane fires a few times a month at most.

EventBridge Scheduler config

- `ic-daily-tick` — `cron(0 9 * * ? *)` in the SMB's timezone. Target: `chaser` Lambda.
- `ic-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `ic-weekly-digest` — `cron(0 9 ? * MON *)` in TZ. Target: `digest` Lambda.
- `ic-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `sender` when a quiet-hours, weekend, or holiday defer is needed. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `invoices.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.

- SES inbound rule set `ic-inbound-rules`: one rule with recipient `invoices@your-company.com` → spam scan → S3 PUT to `s3://ic-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the reminder emails: verify a sender identity at `billing@your-company.com` with DKIM and SPF on the parent domain, and set the reply-to to a monitored inbox so customer replies reach a human. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **chaser role:** `s3:GetObject` on the invoices, rules, and voice keys; `dynamodb:Query` + `GetItem` on `ic-sends`, `ic-state`; `dynamodb:BatchWriteItem` for archiving paid chases to `ic-sends-archive`; `events:PutEvents` on the default bus. *No* `bedrock:*`.
- **sender role:** `events:ListSchedules` + `CreateSchedule` for the deferred-send one-offs; `ses:SendRawEmail` from the verified sender identity; `dynamodb:PutItem` on `ic-sends`; `secretsmanager:GetSecretValue` on the pay-link signing secret.
- **action-handler role:** `dynamodb:PutItem` on `ic-state` and `ic-audit`; `secretsmanager:GetSecretValue` on the Slack signing secret and the Sheets-API service-account secret; outbound network access to `sheets.googleapis.com`; `dynamodb:Query` for chase state lookup; on write-off or payment, `dynamodb:BatchWriteItem` for archiving the chase to `ic-sends-archive`.

- **intake-ses-parser role:** `s3:GetObject` on `ic-raw-mime`; `textract:StartDocumentTextDetection` + `StartDocumentAnalysis`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token.
- **drive-sync and intake-webhook roles:** `secretsmanager:GetSecretValue` on the Google service-account secret and the webhook secret; `s3:PutObject` on the invoices and rules buckets; outbound network to `www.googleapis.com`.

Slack interactive flow

The Slack incoming webhook is the simplest delivery surface but doesn't support interactive button responses. So the daily-review and intake-proposal messages are posted via the `chat.postMessage` Web API instead, with Block Kit blocks containing the action buttons. Button clicks are sent by Slack to the configured Interactivity request URL, which is the `action-handler` Function URL. `action-handler` verifies the Slack signing secret on the inbound request, parses the action_id (`pause`, `disputed`, `write_off`; or `approve`, `edit`, `discard` on intake), opens a modal if needed (Pause opens a modal; Disputed and Write-off are one-tap), and processes the response when the modal is submitted.

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

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch metric for alerting.
- **Alarms:** chaser Lambda failures > 0 in a day (the daily tick is the one piece that has to run); sender failure rate > 1% in 24h; action-handler signature-verification failures > 5/hour (might mean the Slack secret rotated); SES bounce or complaint rate above the SES reputation thresholds (a reminder to a stale contact that hard-bounces should page).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `ic-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive and Sheets APIs live in Secrets Manager under `ic/drive/sa` (one service account with scopes for both APIs). Slack bot token and signing secret under `ic/slack/*`. The webhook shared secret and the pay-link signing secret under `ic/webhook/*` and `ic/paylink/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, holiday list reference, quiet-hours window, weekend rule, and account-owner fallback all live in Parameter Store under `/ic/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) 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 both `ic-invoices-source` and `ic-rules-source` so a bad Drive edit can be rolled back in one click, and version the EventBridge Scheduler timezone setting so you don't accidentally start running the daily tick in UTC after a CI rotation. SAM fits cleanly; a CDK Python stack also works. Total deployable surface: around eight Lambdas, four DDB tables, four S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SES rule set, and one Budgets alarm.

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