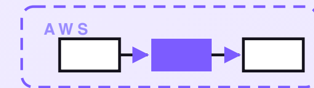
7-PART SERIES · FREE COMPANION



Churn predictor

A serverless predictor that spots customers who are quietly drifting away before they leave. It watches simple signals you already have — last order, support mood, login gaps — scores who looks at-risk with plain, explainable rules, and hands the owner a short weekly “reach out to these five” list with the reason for each. A human wins them back; the system only flags and explains, and never messages customers on its own. 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/churn-predictor

CONTENTS

Churn predictor

01 A churn predictor on AWS for a few dollars a month
02 How an at-risk customer gets watched
03 How a churn score gets calculated
04 How an at-risk list reaches the owner
05 How a win-back gets tracked
06 What the churn predictor costs
07 Engineering reference: the churn predictor architecture

PART 1 OF 7

MAY 26, 2026 PART 1 OF 7 · CHURN PREDICTOR SERIES ~5 MIN READ

A churn predictor on AWS for a few dollars a month

Customers rarely quit with a bang. They quit quietly. The regular who used to order every week now orders every month. The account that opened three angry support tickets and then went silent. The user who hasn't logged in since February. By the time someone notices, they're already gone — and winning a customer back after they leave costs far more than a friendly call while they're still wavering. This post walks through the design of a small predictor that watches the simple signals you already have, scores who looks at-risk in plain language, and hands the owner a short weekly list of people to reach out to — with the reason for each.

KEY TAKEAWAYS

- Three sources for the customer list: a Drive sheet, an order-feed import, and a support-inbox lane.
- Every customer lands in one of four bands each week: steady, watch, at-risk, or churning.
- The score is plain arithmetic — a weighted sum of a few signals, all spelled out in the rules doc.
- The owner gets a short weekly list, capped at five names, each with a plain reason. A human reaches out.
- Designed on AWS for about $2/month at typical small-business volume. It never messages customers itself.

The whole system on one page

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

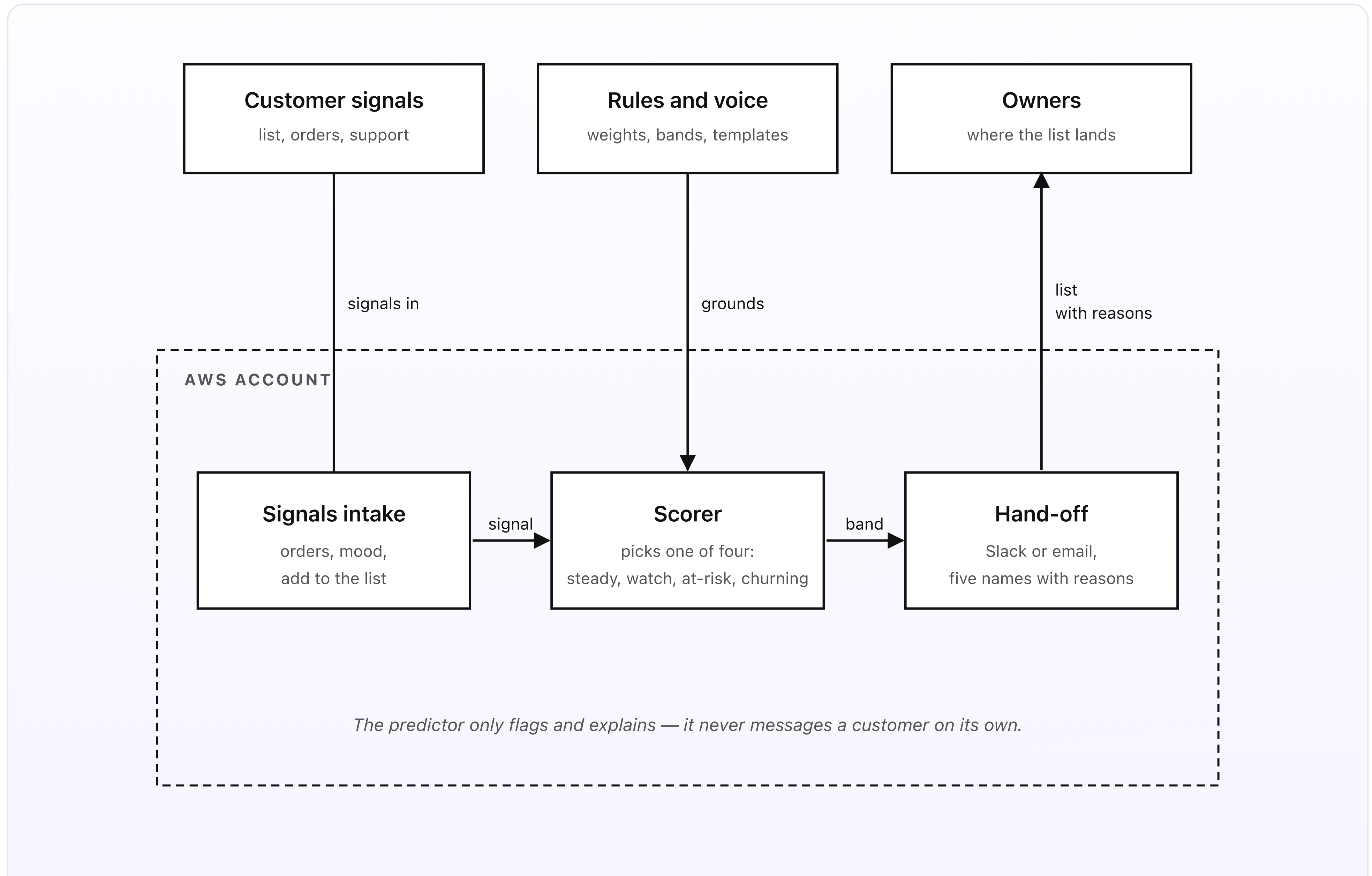

Customer signals
list, orders, support
Rules and voice
weights, bands, templates
Owners
where the list lands
signals in
grounds
list
with reasons
AWS ACCOUNT
Signals intake
orders, mood,
add to the list
signal
Scorer
picks one of four:
steady, watch, at-risk, churning
band
Hand-off
Slack or email,
five names with reasons
The predictor only flags and explains — it never messages a customer on its own.

Fig 1. Three sources outside, three pieces inside AWS. Signals flow in from a Drive list, an order-feed import, and a support-inbox lane. The Scorer runs weekly and picks one of four bands. The Hand-off sends a short list of at-risk customers, with a reason for each, to the right owner.

What you set up once (the outside)

- **Customer signals.** A Google Sheet in a Drive folder, one row per customer: name, owner email, last order date, order count, support mood (a number from sour to happy), last login date, plan, and monthly value. You fill it in once with what you have; the numbers that change often — last order, order pace, mood — get refreshed by two other lanes covered in Part 2: an order-feed import (your store or billing tool drops a daily export and the predictor updates the order columns) and a support-inbox lane (forward or copy support tickets to a dedicated address and the predictor reads each one's mood into a simple score).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc holds the signal weights — how many points a long gap since the last order is worth, how much a drop in order pace counts, how much a sour support mood adds, how much a login gap matters — and the cut-offs for the four bands. It also lists the owner per customer or per segment, the weekly cap (default five names), and any customers to never flag (your biggest account's owner may want to handle that one personally). The *voice* doc holds the templates for the weekly list and the one-line reason per customer.
- **Owners.** The account managers responsible for each customer. Each owner has a Slack member ID (so the list is a private DM, not a public post) or, if Slack isn't set up, an email address. The weekly list lands with each at-risk

customer's name, days since last order, the plain reason they were flagged, the monthly value at stake, and three buttons — *Reached out*, *Won back*, *Lost*.

What runs on every weekly run (the inside)

- **The signals intake.** Three sources feed the list. The Drive sheet itself is the canonical store. The order-feed import updates last-order and order-pace columns from a daily export your store or billing tool already produces. The support-inbox lane reads each forwarded ticket: a Bedrock Haiku 4.5 call rates the customer's mood on a simple scale (sour, flat, happy) and writes it back to the row, so a string of frustrated tickets nudges the score without anyone hand-typing it.
- **The scorer.** Runs once a week, Monday at 8am local. Reads the list. For each customer, it turns each signal into points using the weights in the rules doc — days since last order, drop in order pace versus their own history, sour support mood, login gap — and adds them up. The total lands the customer in one of four bands. *Steady*: low score, nothing to do. *Watch*: rising score, noted but not surfaced yet. *At-risk*: crossed the threshold — eligible for this week's list. *Churning*: high score, already drifting hard — top of the list. The scorer calls no model; the arithmetic is plain Python so anyone can check the math.
- **The hand-off.** Reads the voice doc, takes the highest-scoring at-risk customers up to the weekly cap, and writes a one-line plain reason for each — "no order in 47 days, down from weekly; two sour support tickets last month." Sends the list as a Slack DM or email to each owner. Every list and every outcome the owner records writes a row in DynamoDB so the next week's run knows who was already contacted. A monthly summary writes an owner-ready

paragraph: how many were flagged, how many were reached, how many were won back, how many were lost.

In plain words

Your customer Greenfield Cafe used to order beans every week. The order pace slipped to monthly in March, and last week they opened a support ticket that read as frustrated. The owner of that account is your sales rep, Dan. On Monday the predictor puts Greenfield near the top of Dan's weekly list: "Greenfield Cafe — $380/mo — no order in 34 days, down from weekly; one sour support ticket. *Reach out?*" with three buttons. Dan calls Greenfield on Tuesday, learns a delivery was late, fixes it, and taps *Won back*. The predictor resets Greenfield to steady and won't flag it again unless the signals slip once more. If Dan had done nothing, Greenfield would have quietly stopped ordering — and Dan would have found out two months later when someone asked why revenue dipped.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the steady, invisible leak of customers who would have stayed if anyone had called in time.

DESIGN RULES THAT SHAPED EVERY DECISION

- The system only flags and explains. A human reaches out. Nothing is ever sent to a customer automatically.
- The score is plain arithmetic, not a black box. Every flag ships with the exact reason that tripped it.
- Four bands, always. Steady, watch, at-risk, churning. There is no fifth.
- The weekly list is capped (default five). A list of forty names is a list nobody acts on.
- The customer list lives in Drive. Changing a weight, an owner, or a do-not-flag customer doesn't need a deploy.
- Every outcome is logged. Review a save or a loss next quarter and the trail is there.

Why this shape

Most small businesses find out about churn from the revenue chart, which is the worst possible place — it tells you a customer left weeks after the moment you could have done anything about it. The fancier answer, a machine-learning model trained on your data, is overkill for a small business and worse than useless if nobody trusts its output: a score of "0.83 risk" with no reason attached gets ignored, and rightly so.

The setup above keeps the data where the team already keeps it, but adds a small system that *looks at* the simple signals every week and does plain, checkable

arithmetic. The list is short enough to act on. Each name comes with a reason a human can agree or disagree with. And the act of reaching out — the part that actually saves the customer — stays with a person who can read the room. The predictor is the early-warning bell, not the firefighter.

The next four posts walk through each piece in turn: how an at-risk customer gets watched, how a churn score gets calculated, how the at-risk list reaches the right owner, and how a win-back gets tracked once they act. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 26, 2026 PART 2 OF 7 · CHURN PREDICTOR SERIES ~4 MIN READ

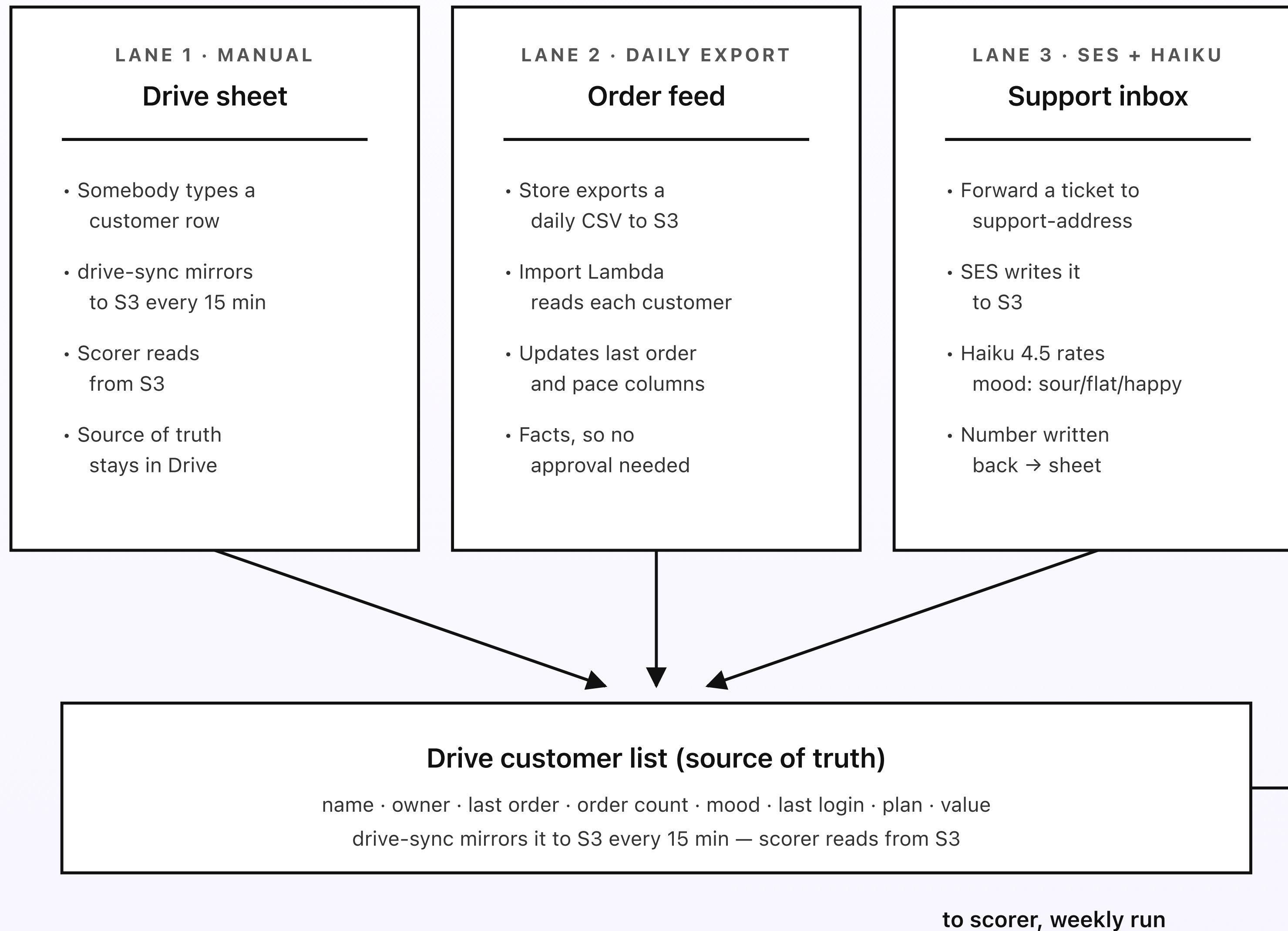
How an at-risk customer gets watched

The predictor only scores what's in the list. So the first job is making sure the list actually reflects how each customer is behaving right now. There are three ways the signals get in: somebody types them in the Drive sheet, a daily order export updates the order columns, or a support ticket gets read for its mood. The first one is obvious. The other two exist because in real life nobody hand-updates a sheet every time a customer orders late or sends a grumpy email.

KEY TAKEAWAYS

- Three intake lanes feed one list: the Drive sheet, a daily order-feed import, and a support-inbox lane.
- The order feed refreshes last-order date and order pace from an export your store already produces.
- Support tickets are read by Bedrock Haiku 4.5, which rates each one’s mood on a simple scale.
- The mood reader only writes a number back to the row — it never replies to the customer.
- 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 refresh the signals it holds.

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the order feed and the support inbox are conveniences that refresh the signals. The drive-sync Lambda mirrors the sheet to S3 so the scorer can read it without hitting Drive on every run.

Lane 1: the Drive sheet itself

The simplest lane. Open the list sheet in Drive, add or edit a row, save. The columns are short: name, owner email, last order date, order count, support mood, last login date, plan, and monthly value. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://cp-list-source/customers.csv` if the sheet has changed since the last sync. The scorer 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 initial setup — loading your customers and the things that don't change often, like owner and plan — and the occasional manual fix, like marking a customer "never flag."

Lane 2: the order feed (the one that keeps the score honest)

The strongest churn signal a small business has is usually the simplest: did they order, and how recently. Most stores and billing tools can already export a daily file — orders, dates, customers. Lane 2 picks that up. The export lands in `s3://cp-order-feed/` (your store drops it there on a schedule, or a tiny connector does). The S3 PUT triggers an import Lambda. The Lambda groups the rows by customer, finds each customer's most recent order date and their typical gap

between orders, and writes `last_order_date` and `order_pace` back to the Drive sheet via the Sheets API.

This lane needs no human approval, because it isn't making a judgement — it's copying facts. A customer either ordered on the 14th or they didn't. The order feed is what lets the predictor notice the quiet slowdown — weekly to monthly — that is the clearest early sign someone is drifting, long before they cancel anything.

Lane 3: the support inbox

The other early sign is mood. A customer who sends a string of frustrated tickets and then goes quiet is often on their way out. Lane 3 reads that without anyone hand-scoring it. Set up a dedicated address — something like `support-signals@your-company.com` — via Amazon SES, and forward (or auto-copy) support tickets there. SES writes the raw message to `s3://cp-raw-mime/`. The S3 PUT triggers a reader Lambda. The Lambda pulls the ticket text and calls Bedrock Haiku 4.5 with a short prompt: "Rate this customer's mood: sour, flat, or happy. Return one word. Do not reply to the customer." The result is turned into a number and written to the customer's `support_mood` column, blended with recent tickets so one bad day doesn't dominate.

The mood reader is the only place a model touches a customer's words, and it is strictly read-only. It rates a mood and writes a number. It never drafts or sends anything to the customer — the whole point of this system is that a human, not a model, decides what to say to a wavering customer.

Why the list stays the source of truth

Three lanes in, but only one place the scorer actually reads. That’s deliberate. If two lanes both wrote straight into the scorer’s state, every “why was this customer flagged?” question would mean checking three places. Funneling everything through the Drive sheet means there is exactly one row per customer, and any rep can read or edit any of it without learning a new tool. The convenience lanes keep the signals fresh, but they always land in the sheet on the way.

Next post: how the scorer actually reads the list, turns each signal into points, and lands every customer in one of four bands.

PART 3 OF 7

MAY 26, 2026 PART 3 OF 7 · CHURN PREDICTOR SERIES ~5 MIN READ

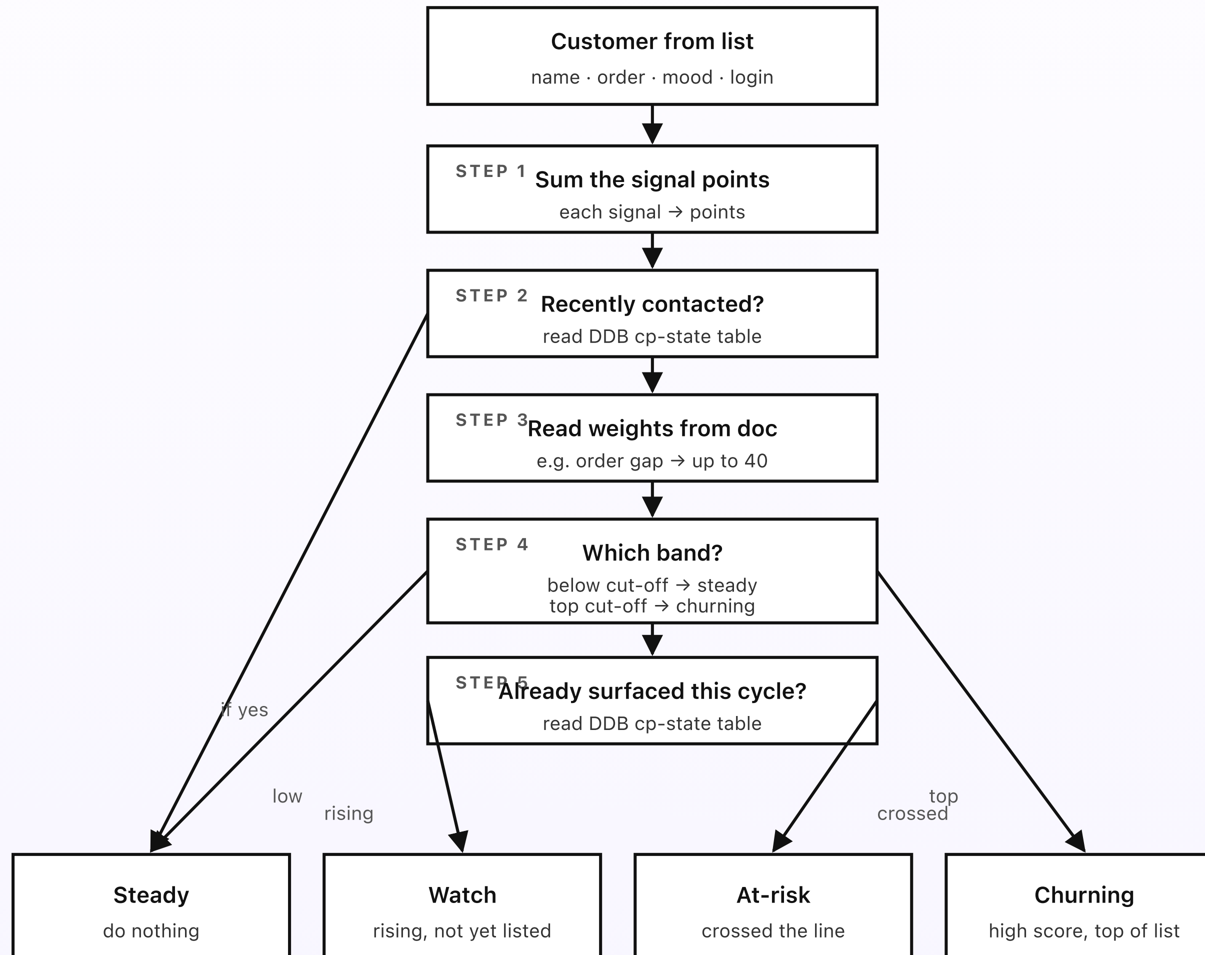
How a churn score gets calculated

Once a week, on Monday at 8am local time, an EventBridge Scheduler rule fires the scorer Lambda. The Lambda reads the list, looks at one customer at a time, turns each signal into points, adds them up, and lands the customer in one of four bands. The whole calculation is plain arithmetic. No model. No vector retrieval. Every weight and every cut-off lives in the rules doc, where a rep can change it without a deploy — and every flag carries the exact points that produced it.

KEY TAKEAWAYS

- The scorer runs once a week via EventBridge Scheduler at 8am local time.
- Each signal becomes points using weights in the rules doc — order gap, pace drop, sour mood, login gap.
- Four bands per customer, every run: steady, watch, at-risk, churning.
- DynamoDB tracks last week’s score and any recent contact so the same name isn’t flagged twice in a row.
- The scorer never calls a model. The math is fully deterministic and shown on every flag.

The scoring flow, per customer



The rules doc holds every weight — change one and next Monday's run uses the new value.

Fig 3. The scorer's flow, per customer, per weekly run. Five steps decide which of four bands applies. The rules doc holds every weight and cut-off; the scorer only adds them up.

Signals into points: the weights are in the doc

The rules doc has one short section per signal. Each names the rule in plain prose: "Order gap: zero points until they pass their usual order interval, then one point per extra day, up to forty. Order-pace drop: twenty points if their pace has halved versus their own history. Support mood: fifteen points for sour, five for flat, zero for happy. Login gap: ten points if no login in thirty days." The points add to a total out of one hundred. The cut-offs that follow — steady, watch, at-risk, churning — are also in the doc, as plain numbers like "watch at 30, at-risk at 50, churning at 75."

The weights exist for a reason, and they're tuned to your business. A coffee roaster whose customers order weekly should weight the order gap heavily. A software tool whose customers pay monthly but use it daily should weight the login gap more. The doc is where you say which signals matter for *your* customers, in numbers anyone can read.

Per-customer overrides exist too. The list sheet has an optional column called `never_flag`. Set it for a customer and the scorer still computes their score for the record, but never surfaces them — the right escape hatch for the marquee account whose owner insists on watching it personally.

Four bands, always

Every customer, every run, lands in exactly one of four bands. The names are simple on purpose.

- **Steady.** The total is below the watch cut-off, or the customer was contacted within the pause window. Nothing to do. Most customers, most weeks, are steady.
- **Watch.** The total crossed the watch cut-off but not the at-risk line. The score and reason are recorded to the `cp-state` DynamoDB table, but the customer is *not* put on this week's list. Watch is the "keep an eye on this" band — it shows up in the monthly summary, not the weekly nudge.
- **At-risk.** The total crossed the at-risk line. The customer is a candidate for this week's list, ranked by score. The reason — the specific points that pushed them over — is recorded so the hand-off can show it.
- **Churning.** The total is above the churning cut-off — already drifting hard. These go to the top of the weekly list. A churning customer with high monthly value is exactly who an owner wants to see first thing Monday.

State that makes the score fair

The scorer reads one DynamoDB table every run. `cp-state` records each customer's latest run: `(customer_id, score, band, reason, surfaced_date, last_contact)`. With that one table, the band decision and the "was this customer just contacted?" check are a few dozen lines of Python and zero magic. A given customer with given signals and a given contact history always produces the same band. And because the table remembers who was surfaced and

contacted, the same name doesn't crowd the list two weeks running while the owner is mid-conversation.

When an owner records an outcome — reached out, won back, lost — that writes back to `cp-state` and resets the relevant fields. Part 5 covers the outcome flow in detail.

Why the score uses no model

The scorer could call a model to produce a cleverer risk number. It doesn't. Two reasons. First, the score has to be something a human can check and argue with — if the doc says a sour ticket is worth fifteen points, then a sour ticket is worth fifteen points, and the owner can see exactly why a name is on the list. A model in that loop turns a number people trust into a number people ignore. Second, the whole list of customers gets scored every week, and most are steady, so a model call per customer would be money spent to reproduce arithmetic.

Bedrock fires elsewhere — on the support-mood lane in Part 2, where it reads a ticket's mood into a single number, and on the weekly reason write-up and the monthly summary in Part 6. Not on the score. The scorer itself is plain Python that reads a doc and does sums you could redo on paper.

Next post: how the at-risk list reaches the right owner, how the weekly cap and quiet timing are honored, and what a plain reason looks like next to each name.

PART 4 OF 7

MAY 26, 2026 PART 4 OF 7 · CHURN PREDICTOR SERIES ~5 MIN READ

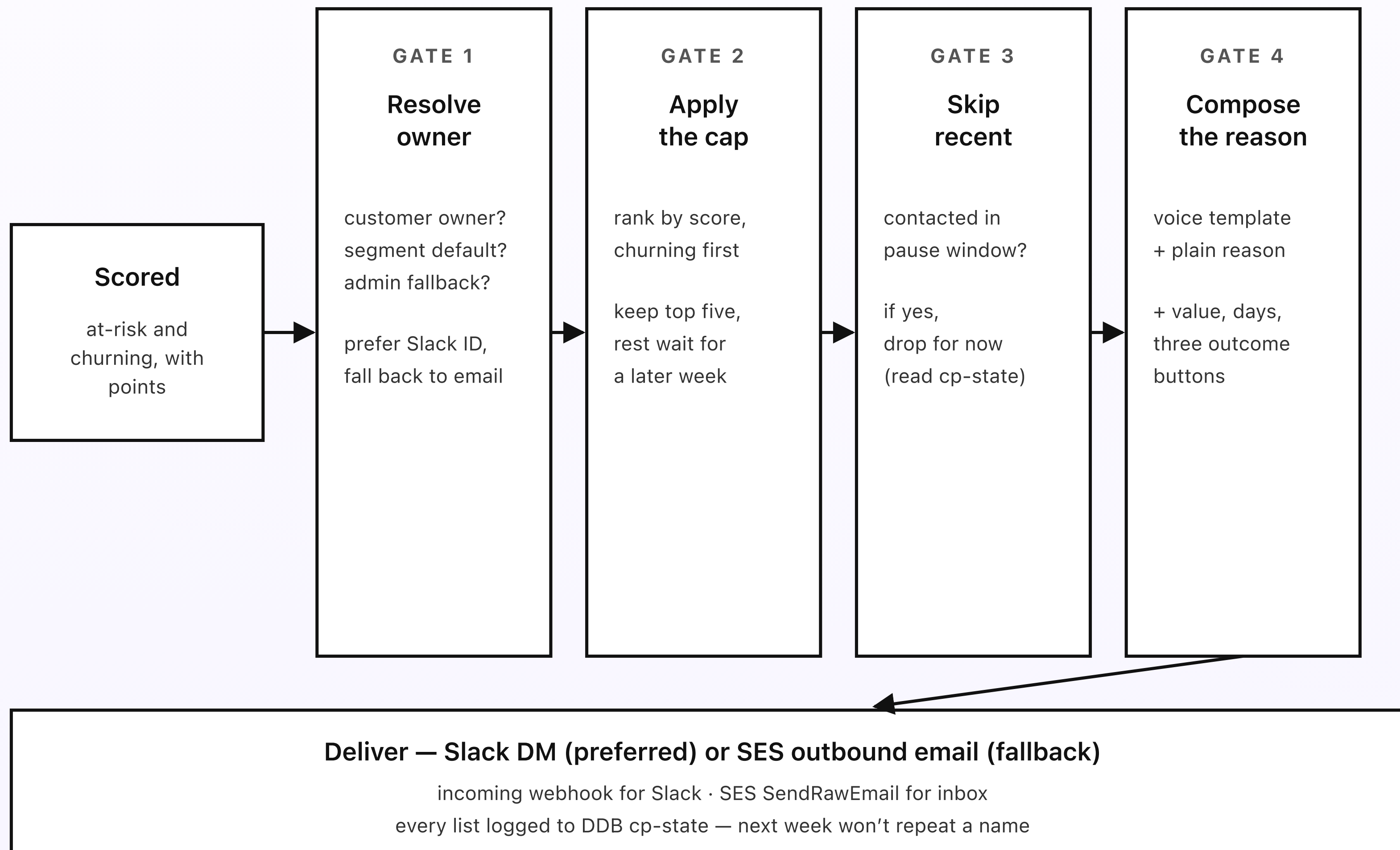
How an at-risk list reaches the owner

The scorer found this week's at-risk and churning customers. Now the hand-off Lambda has to figure out who owns each one, trim the list to something a person will actually act on, write a plain reason next to every name, and deliver it on the right channel. Get any of those wrong and the list is worse than nothing: forty names nobody reads, a customer routed to a rep who never met them, a score with no reason that gets waved away. Four small guardrails sit between the score and the list landing.

KEY TAKEAWAYS

- Owner lookup: per-customer owner beats per-segment default beats fallback to the configured admin.
- The weekly list is capped (default five per owner) and ranked by score, churning first.
- Each name gets a plain one-line reason built from the exact points that flagged it.
- Slack DMs are the default; email is the fallback if no Slack ID is configured.
- The list only suggests who to contact — the owner decides, and nothing is sent to the customer.

Four guardrails on every list



The list only suggests who to contact — the owner decides, and nothing is sent to the customer.

Fig 4. Four guardrails between the score and the delivered list. Resolve the owner. Apply the cap. Skip recent contacts. Compose a plain reason. Then ship via Slack or email and log the list so next week doesn't repeat a name.

Gate 1: resolve the owner

Three places the hand-off Lambda looks for the owner of a customer, in order. First, the list sheet's per-customer `owner_email` column — if a row has a specific account manager assigned, that person owns it regardless of segment. Second, the per-segment default in the rules doc ("all small-plan customers default to the success team"). Third, the configured admin fallback — the person who set up the predictor and gets every unowned name. The fallback should never fire in steady state; if it does, the monthly summary names every customer that hit the fallback so the rules doc can be updated.

Once the hand-off knows which person owns a name, it looks up their delivery preference. The voice doc maps each owner to a Slack member ID if one is set, otherwise to an email address. Slack is preferred because the list is a private DM with buttons the owner can tap as they work through it. Email is the fallback so nobody is left out.

Gate 2: apply the cap

This is the gate that makes the whole system usable. A scorer let loose on a few thousand customers will happily find forty at-risk names in a bad week — and a list of forty is a list an owner glances at and closes. Gate 2 ranks each owner's candidates by score, churning customers first, and keeps only the top few. The

default cap is five, set in the rules doc. The customers who didn't make the cut aren't lost — their score is still recorded, and they'll surface in a future week if they stay at-risk and the higher-scoring names get handled.

Five is a deliberate number. It's short enough that an owner can actually call all five in a week, which means the list is a to-do, not a wall. A churn predictor that surfaces everything surfaces nothing.

Gate 3: skip recent contacts

When an owner reaches out to a customer, they don't want that name back on the list next Monday while the conversation is still going. Gate 3 reads the `cp-state` table for each candidate's `last_contact` date and drops anyone the owner contacted inside the pause window (default two weeks, set in the rules doc). The customer is still being scored every week; they're just held off the list until the pause ends, at which point — if their signals haven't improved — they come back.

The pause exists because nagging an owner about a customer they're already working is the fastest way to get the whole list ignored. The right cadence is "here are new names, and the ones you haven't had a chance to act on," not "here is the same list as last week."

Gate 4: compose a plain reason, then ship

The voice doc has a short list template and a reason template. For each surviving name, the hand-off turns the scorer's points into one plain sentence: "no order in 47 days, down from weekly; two sour support tickets last month." This is the one

place a model earns its keep on the hand-off — a small Bedrock Haiku 4.5 call turns the raw points into a readable line, grounded strictly in those points so it can't invent a reason that isn't in the data. The Lambda fills the template, attaches three buttons — *Reached out*, *Won back*, *Lost* — and ships the list via the Slack DM. The webhook and bot token live in Secrets Manager.

For email fallback, the same list is wrapped in a small HTML email with the same reasons and links that, when clicked, hit a Function URL that records the outcome — the email equivalent of the Slack buttons.

The reason is the heart of the whole design. A score on its own ("Greenfield Cafe: 72") tells the owner nothing they can act on. The reason ("no order in 34 days, down from weekly; one sour ticket") tells them exactly what to ask about when they call. The list isn't "the computer says these people are leaving." It's "here's what changed for each of them — you decide."

Every list — Slack or email — writes a row to `cp-state` in DynamoDB. Next week's run reads it and knows which names were already surfaced and to whom.

Why the guardrails exist

None of these gates are exotic. They're the kind of small care a thoughtful manager would take if they were building the list by hand — check who actually owns each account, keep it short enough to action, don't re-list someone you're mid-conversation with, and say *why* next to every name. Putting them in code as four small sequential gates makes them part of the design, not something you're trusting a busy week to remember. And keeping the act of reaching out with a human is the most important guardrail of all: the system points, a person decides.

Next post: how a win-back gets tracked once an owner has acted — how the predictor records the contact, the save, or the loss, and keeps the trail straight.

PART 5 OF 7

MAY 26, 2026 PART 5 OF 7 · CHURN PREDICTOR SERIES ~5 MIN READ

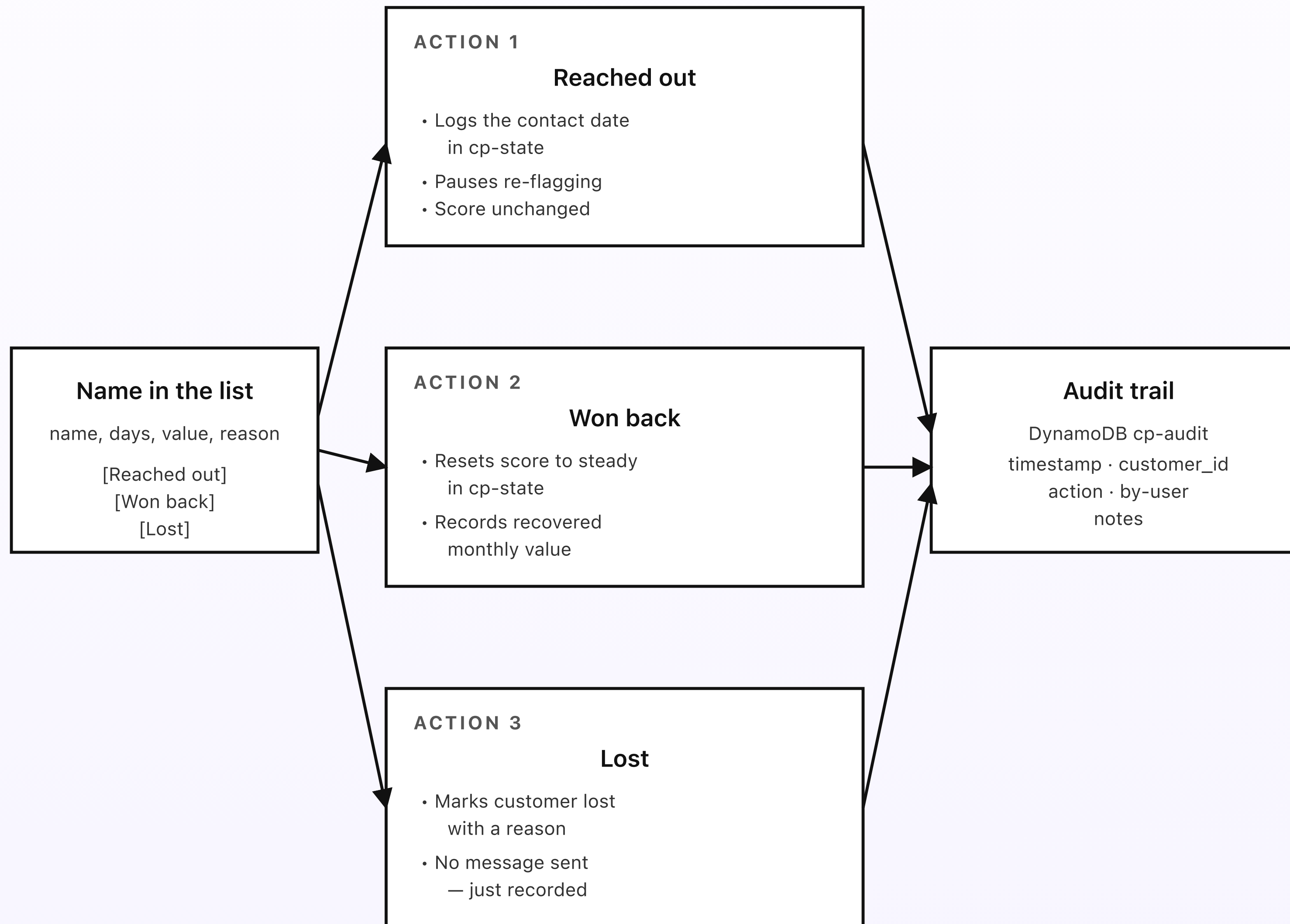
How a win-back gets tracked

The weekly list lands in Dan's Slack DM at 8:03am Monday. Greenfield Cafe is near the top: no order in 34 days, one sour ticket. There are three buttons next to the name. What happens when he taps one? The honest answer is "it depends on what actually happened with the customer." This post walks through the three things the predictor can record on a name — reached out, won back, lost — and how the state, the score, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Three actions per name: *reached out* (log and pause), *won back* (save and reset to steady), *lost* (record the churn).
- Each action updates the `cp-state` table and writes an audit row.
- A win-back resets the customer's score so the next slip starts a fresh cycle.
- A loss records a reason so the monthly summary can show why customers leave.
- The buttons are a Slack interactive message backed by a Function URL.

Three actions on a name



Marking a name doesn't message the customer — it records what the human did, so saves are countable.

Fig 5. Three actions per name, three different effects. Reached out logs the contact and pauses re-flagging. Won back resets the score and records the save. Lost records the churn with a reason. Every action writes to the audit trail.

Action 1: reached out (the most common)

Dan calls Greenfield, has a good conversation, and now wants the predictor to know he's on it. He taps *Reached out*. No modal needed — it's one tap. The button submits to a Function URL Lambda. Two things happen. First, the Lambda writes today's date to the customer's `last_contact` field in `cp-state`, which the hand-off's Gate 3 reads next week to keep Greenfield off the list during the pause window. Second, a `reached_out` row is written to `cp-audit` with the user and timestamp.

The score itself isn't changed — Greenfield is still genuinely at-risk until the signals improve. Reached out just means "a human is handling this; don't nag." If the customer starts ordering again, the order feed will lift their signals naturally and the score falls on its own. If they don't, Greenfield reappears on the list when the pause ends, which is exactly the prompt Dan needs to follow up.

Action 2: won back (the save)

A week later Greenfield places a big order. Dan taps *Won back*. The Function URL Lambda resets the customer's score in `cp-state` to steady, clears the surfaced and contact fields so the cycle starts fresh, and records the recovered monthly value (defaulting to the customer's value from the list, editable if the relationship

changed). A `won_back` row goes to `cp-audit` with the user, the timestamp, and the saved value.

Recording the save matters for more than tidiness. It's how the business finds out the predictor is worth running. The monthly summary in Part 6 adds up the recovered value across all the saves — "the team won back $3,400 of monthly revenue this month from eleven at-risk customers" — which is the number that justifies the two dollars of AWS spend many times over. A win-back the team made but never recorded is a save the business can't see.

Action 3: lost (the honest record)

Sometimes the customer is gone. They found a cheaper supplier, or their needs changed, or they went quiet and never came back. Dan taps *Lost*. An optional one-tap reason field offers a few choices — price, competitor, went quiet, other — so the loss isn't a mystery later. The Function URL Lambda marks the customer as lost in `cp-state` (so the scorer stops surfacing them) and writes a `lost` row to `cp-audit` with the reason.

The reasons add up into something useful. If half the month's losses are tagged "price," that's a pricing conversation, not eleven separate failures. If most are "went quiet," the predictor probably needs to flag earlier — a hint to tighten the weights in the rules doc. The lost record turns each departure from a number on the revenue chart into a small, honest note about why, and nothing about marking a customer lost ever reaches the customer — it's purely an internal record.

Every action is logged, every count is honest

The `cp-audit` table records every reached-out, won-back, and lost with the user who took the action, the timestamp, and a snapshot of the customer's score and band at the time. If a button gets tapped by mistake — *Lost* instead of *Reached out* on a fat-fingered Monday — a rep can run an "undo last action" through a small admin command that reads the snapshot and restores the state. The undo is itself an audit row, so the trail stays clean.

This trail is what makes the monthly summary trustworthy. When it says the team saved eleven customers and lost four, those aren't estimates — they're counted button taps, each tied to a person and a moment. The predictor's whole value rests on the team trusting the list, and the team trusts the list because the outcomes are recorded honestly and nothing happened behind their back.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the weekly run keeps it so cheap.

PART 6 OF 7

MAY 26, 2026 PART 6 OF 7 · CHURN PREDICTOR SERIES ~3 MIN READ

What the churn predictor costs

The predictor is one of the cheapest systems in this whole series. The weekly run reads a CSV from S3, does some plain arithmetic, writes a few rows to DynamoDB, and posts one short list per owner. It calls no model on the hot path. Bedrock fires only when a support ticket comes in, when the weekly list needs its plain reasons written, and once a month for the summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2/month at typical SMB volume (around 500 customers).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The weekly scoring run costs pennies — no model calls.
- Bedrock fires only on support-ticket mood, the weekly reasons, and the monthly summary.
- At 2,000 customers the bill is around $5. At 5,000 customers it's around $9.

Cost at three volumes

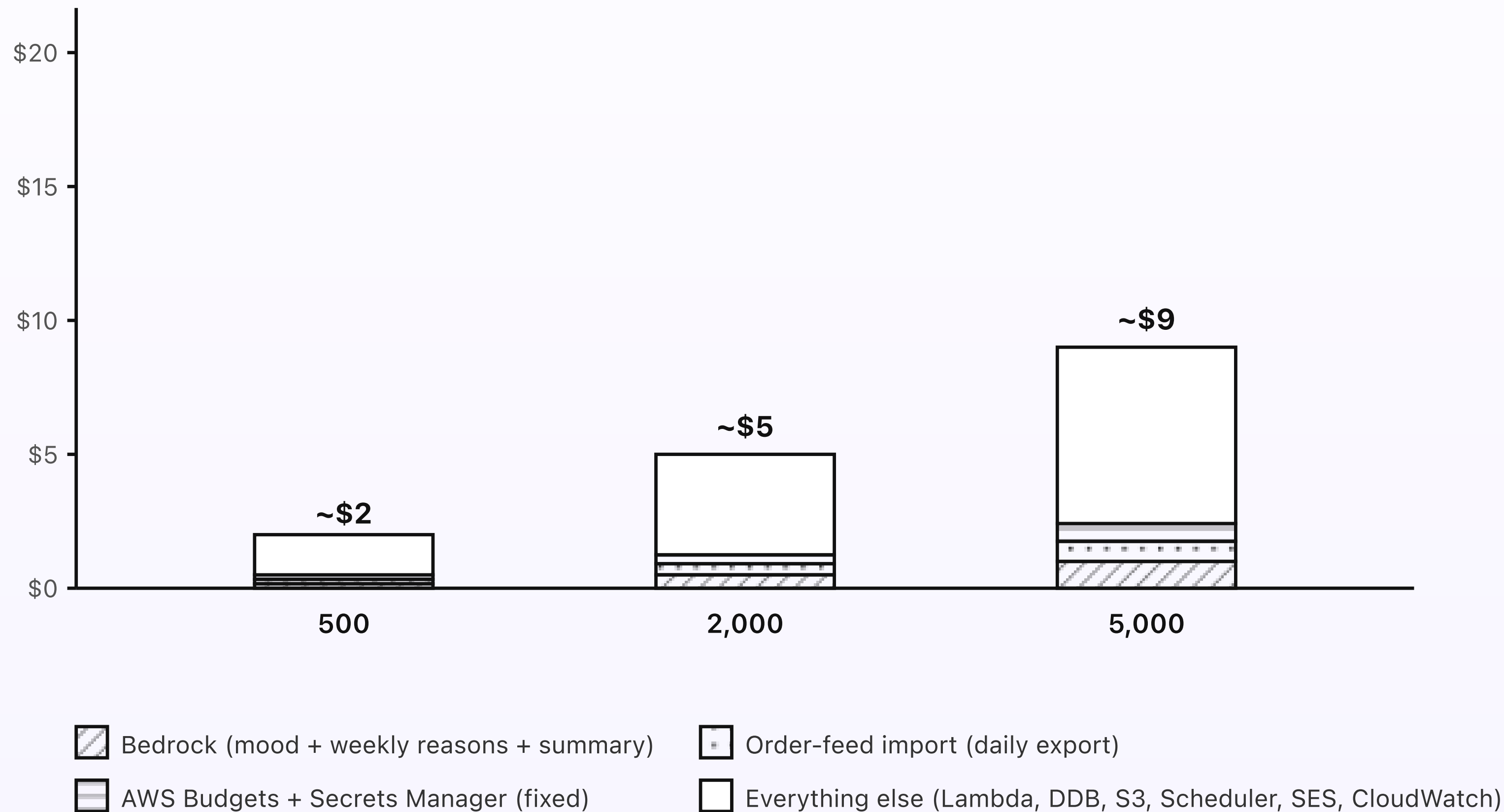


Fig 6. Monthly cost at three customer volumes. Bedrock and the order-feed import are small slivers because they only fire on incoming tickets, the weekly reasons, and the daily export. The dominant cost is the everything-else bucket: the weekly run scoring every customer.

Where the dollars actually go

Lambda runtime (the bulk). The scorer runs once a week. Each run reads the customer CSV from S3, iterates the rows, sums the signal points for each, and decides on a band. At 500 customers, that's a few hundred milliseconds. At 5,000 it's a couple of seconds. Either way it's pennies a month. Add the hand-off Lambda building one list per owner, the Function URL Lambda for outcome buttons, the order-import Lambda running daily, the support-mood Lambda firing per ticket, and the drive-sync Lambda every fifteen minutes — the Lambda total still lands under a dollar at all three volumes.

DynamoDB on-demand. Two small tables: `cp-state` and `cp-audit`. Reads are dominant during the weekly run (one read per customer, plus recent state). Writes are list events and outcome rows. Pennies a month at any of these volumes.

S3 + Storage. The mirrored customer CSV, the daily order exports, and the archived support MIME. A few MB total at SMB volume. Effectively free.

EventBridge Scheduler. The weekly scoring rule, the daily import trigger, and the hourly housekeeping. A handful of invocations a day. Pennies.

SES. Inbound for the support lane: $0.10 per thousand received messages (a few cents a month for an SMB). Outbound for email-fallback lists: $0.10 per thousand sent. Both are negligible at this scale.

Bedrock (only when something fires it). The weekly scoring run uses no Bedrock. The support-mood lane fires Haiku 4.5 once per forwarded ticket: a few hundred input tokens and one word out, so a tiny fraction of a cent each. The hand-off fires Haiku 4.5 once per name on the weekly list to write the plain reason

— at five names per owner, that's a handful of small calls a week. The monthly summary is one larger call. At SMB ticket and list volume, Bedrock costs cents.

The order-feed import. Reading the daily export and updating rows is plain Lambda and Sheets-API work — no model, no special service. Counted on its own because it runs every day, but it's still a sliver: a short run over one file, once a day.

What doesn't cost money

- **API Gateway.** Replaced by a Lambda Function URL for the outcome buttons.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The scorer wakes once a week and sleeps the rest.
- **A Knowledge Base.** The list is structured rows and the score is plain arithmetic — deterministic math beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the score.** The weekly decision is plain Python. Bedrock fires only on mood, the weekly reasons, and the summary.

How the cost scales

Lambda runtime grows roughly linearly with customer count, because every customer is scored on every run. DynamoDB grows linearly too. Bedrock is uncorrelated with customer count — it only fires when a support ticket arrives, when a name needs its weekly reason, or when it's the first of the month. So the bill at 10,000 customers is around $16; at 20,000 it's around $30. Past those

volumes the weekly-full-scan model probably stops being right (you'd switch to scoring only customers whose signals changed that week), but those are optimizations for very large lists — not redesigns.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The predictor's normal-volume bill stays well under that ceiling — and one saved customer usually pays for a year of it.

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 26, 2026 PART 7 OF 7 · CHURN PREDICTOR SERIES ~8 MIN READ

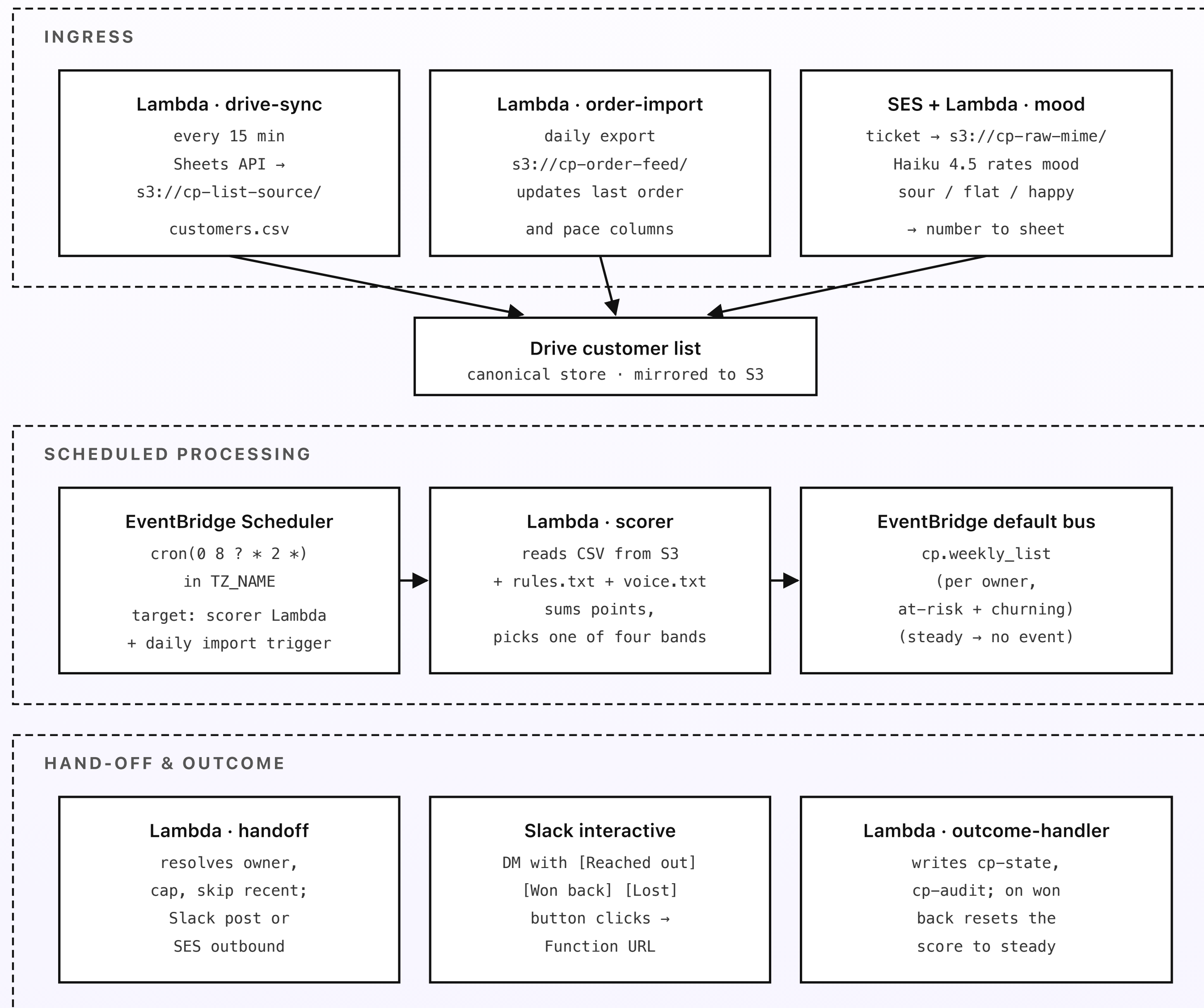
Engineering reference: the churn predictor architecture

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 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 one missed weekly list, not a regional outage. One AWS account dedicated to the predictor (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



The system only flags and explains — every interaction is logged to cp-audit, never sent to a customer.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the list), scheduled processing (the weekly scorer emitting a per-owner list event), hand-off and outcome (the list ships and the owner's outcome 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 `cp/drive/sa`) to export the customer sheet as CSV and write to `s3://cp-list-source/customers.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://cp-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`order-import`** — S3 PUT trigger on `s3://cp-order-feed/` (the store or billing tool drops a daily CSV; a small connector or a scheduled export populates it). Groups rows by customer, derives `last_order_date` and `order_pace` (median inter-order gap over a trailing window), and writes them back to the Drive sheet via the Sheets API `batchUpdate`. Idempotent on re-run of the same file. No model — these are facts. Memory: 256 MB. Timeout: 60 s.
- **`mood-reader`** — S3 PUT trigger on `s3://cp-raw-mime/`. Parses the MIME, extracts the ticket body and the customer's email/identifier, and calls Bedrock

Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) with a constrained prompt that returns one of `sour` / `flat` / `happy`. Maps the label to a number, blends it with the customer's recent mood (exponential moving average over the trailing few tickets so one bad day doesn't dominate), and writes `support_mood` back to the sheet. Strictly read-only with respect to the customer — it never drafts or sends a reply. Memory: 256 MB. Timeout: 30 s.

- **`scorer`** — EventBridge Scheduler target, weekly Monday at 8am local time (the schedule expression runs in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). Reads `s3://cp-list-source/customers.csv` and the rules and voice docs. For each row, turns each signal into points using the weights, sums to a total out of 100, reads prior state from `cp-state`, and assigns a band. Emits one `cp.weekly_list` event per owner carrying that owner's at-risk and churning candidates with their scores and per-signal point breakdowns as the event payload. Steady and watch customers emit no list event. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`handoff`** — EventBridge rule on the `cp.weekly_list` event. Resolves owner, applies the cap (rank by score, churning first, keep top N from the rules doc), drops candidates inside the contact pause window read from `cp-state`, and for each surviving name calls Bedrock Haiku 4.5 to render the point breakdown into a one-line plain reason (grounded strictly in the supplied points). Ships via Slack `chat.postMessage` with Block Kit buttons (`cp/slack/bot-token` in Secrets Manager) or SES `SendRawEmail` for email fallback. Writes the surfaced names to `cp-state` after a successful send. Memory: 512 MB. Timeout: 60 s.
- **`outcome-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive

button clicks (Reached-out/Won-back/Lost) and by email-link clicks. Writes to `cp-state` and `cp-audit`; on won-back, resets the customer's score and clears the surfaced/contact fields; on lost, records the reason and marks the customer so the scorer stops surfacing them. Memory: 256 MB. Timeout: 15 s.

- **`digest`** — optional EventBridge Scheduler target, weekly Friday 4pm. Reads `cp-state` for the watch band and the week's outcomes; posts a short "watch list and outcomes so far" message to a configured Slack channel. No Bedrock; a plain summary table. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `cp-state` and `cp-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph owner narrative (flagged, reached, won back with recovered value, lost with top reasons); emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB · `cp-state`** — one row per customer. PK `customer_id`; attributes: `score`, `band`, `reason`, `surfaced_date`, `last_contact`, `status` (active/lost), `owner`. On-demand. No TTL — it's the live state the scorer reads each week.
- **DynamoDB · `cp-audit`** — one row per write action of any kind. PK `(customer_id, ts)`; attributes: `action` (reached_out/won_back/lost/undo), `by_user`, `before`, `after`, `notes` (e.g. lost-reason, recovered value). On-demand. No TTL — this is the long-term outcome trail the summary counts from.

- **S3** · `cp-list-source` — mirrored CSV from the Drive customer list. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3** · `cp-rules-source` — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3** · `cp-order-feed` — daily order exports from the store/billing tool. Lifecycle to Glacier at 30 days; expiry at 1 year (the derived columns live in the sheet, so the raw exports are short-lived).
- **S3** · `cp-raw-mime` — raw inbound MIME from forwarded support tickets. Lifecycle to Glacier at 30 days; expiry at 7 years.

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`. Three callsites: `mood-reader` for ticket sentiment, `handoff` for the per-name plain reason, and `summary` for the monthly narrative. If a heavier monthly analysis is ever wanted (cohort patterns across reasons), `summary` can be promoted to `anthropic.claude-sonnet-4-6-20250930-v1:0` via its Global profile — but Haiku is enough for the current paragraph.
- **Embeddings.** Not used. The list is structured rows and the score is plain arithmetic; deterministic math beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The scorer itself doesn't call Bedrock; the mood and reason calls are small and bursty around ticket arrival and the Monday run.

EventBridge Scheduler config

- **`cp-weekly-run`** — `cron(0 8 ? * 2 *)` (Mondays at 8am) in the SMB's timezone. Target: `scorer` Lambda.
- **`cp-drive-sync`** — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- **`cp-weekly-digest`** — `cron(0 16 ? * 6 *)` (Fridays 4pm) in TZ. Target: `digest` Lambda.
- **`cp-monthly-summary`** — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **Order import** — the `order-import` Lambda is S3-PUT-driven on `cp-order-feed`, not Scheduler-driven, so it runs whenever the export lands. If the store can't push on a schedule, a `rate(1 day)` Scheduler rule can pull instead.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `support-signals.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `cp-inbound-rules`: one rule with recipient `support-signals@your-company.com` → spam scan → S3 PUT to `s3://cp-raw-mime/<message-id>` → stop. The S3 PUT triggers `mood-reader`.
- SES outbound for the email-fallback lists and the monthly summary: verify a sender identity at `churn@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **scorer role:** `s3:GetObject` on the list, rules, and voice keys; `dynamodb:Query` + `GetItem` on `cp-state`; `events:PutEvents` on the default bus. *No* `bedrock:*`.
- **handoff role:** `s3:GetObject` on the voice doc; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token; `ses:SendRawEmail` from the verified sender; `dynamodb:PutItem` + `Query` on `cp-state`; outbound network access to `slack.com`.
- **outcome-handler role:** `dynamodb:PutItem` + `UpdateItem` on `cp-state` and `cp-audit`; `secretsmanager:GetSecretValue` on the Slack signing secret; `dynamodb:Query` for snapshot reads on undo.
- **mood-reader role:** `s3:GetObject` on `cp-raw-mime`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Sheets-API service-account secret; outbound network to `sheets.googleapis.com`.
- **order-import and drive-sync roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:GetObject` / `PutObject` on the relevant buckets; outbound network to `www.googleapis.com`. *No* `bedrock:*`.

Slack interactive flow

The weekly list is posted via the Slack `chat.postMessage` Web API with Block Kit blocks containing one row per customer and three action buttons each. Button clicks are sent by Slack to the configured Interactivity request URL, which is the `outcome-handler` Function URL. `outcome-handler` verifies the Slack signing secret on the inbound request, parses the `action_id` (`reached_out`, `won_back`,

`lost`), opens a modal if needed (Lost opens a small reason picker; Reached-out and Won-back are one-tap, Won-back optionally confirming the recovered value), 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 `cp/slack/bot-token`. The signing secret is `cp/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:** scorer Lambda failures > 0 in a week (the weekly run is the one piece that has to fire); handoff failure rate > 1% in 24h; outcome-handler signature-verification failures > 5/hour (might mean the Slack secret rotated).
- **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 `cp-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 `cp/drive/sa` (one service account with scopes for both APIs). Slack bot token and signing secret under `cp/slack/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, the signal weights and band

cut-offs (mirrored from the rules doc for fast reads), the weekly cap, the contact pause window, and the admin fallback owner all live in Parameter Store under `/cp/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

Whichever IaC you prefer. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `cp-list-source` and `cp-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 weekly run in UTC after a CI rotation. Deploy with GitHub Actions using OIDC (no long-lived keys) and AWS SAM; a CDK Python stack also fits. Total deployable surface: around eight Lambdas, two 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.