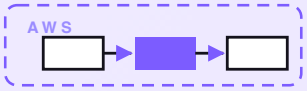
7-PART SERIES · FREE COMPANION



Renewal negotiator

A serverless helper that keeps customers at renewal time. Ahead of each contract or subscription renewal it gathers what it knows — current terms, usage, and history — drafts a tailored renewal offer in your voice with any discount inside your rules, and hands it to you to approve, edit, or skip before it ever reaches the customer. It only prepares the offer; a human always sends and can always override. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Renewal negotiator

PART 1 OF 7

JUNE 11, 2026 PART 1 OF 7 · RENEWAL NEGOTIATOR SERIES ~5 MIN READ

A renewal negotiator on AWS for a few dollars a month

A small business loses more customers at renewal time than it ever loses to a competitor. The annual contract that quietly lapses because nobody reached out two weeks before. The heavy user who would happily have moved up a tier if someone had asked. The loyal account who churned over a price bump that a small loyalty discount would have softened. Renewals are won by getting in front of the customer early with a fair, tailored offer — and most teams just don't have the hours. This post walks through the design of a small helper that, ahead of each renewal, gathers what it knows, drafts an offer in your voice, and hands it to you to approve, edit, or skip. It never sends anything itself.

KEY TAKEAWAYS

- Three sources for accounts: a Drive registry, an inbox forwarding lane, and a calendar import lane.
- Ahead of each renewal the system drafts one tailored offer and puts it in your approval queue.
- The plan and the discount come from your rules; the model only writes the words around them.
- Nothing reaches a customer without a person pressing send. Discounts stay inside your caps.
- Designed on AWS for about $3/month at typical small-business volume.

The whole system on one page

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

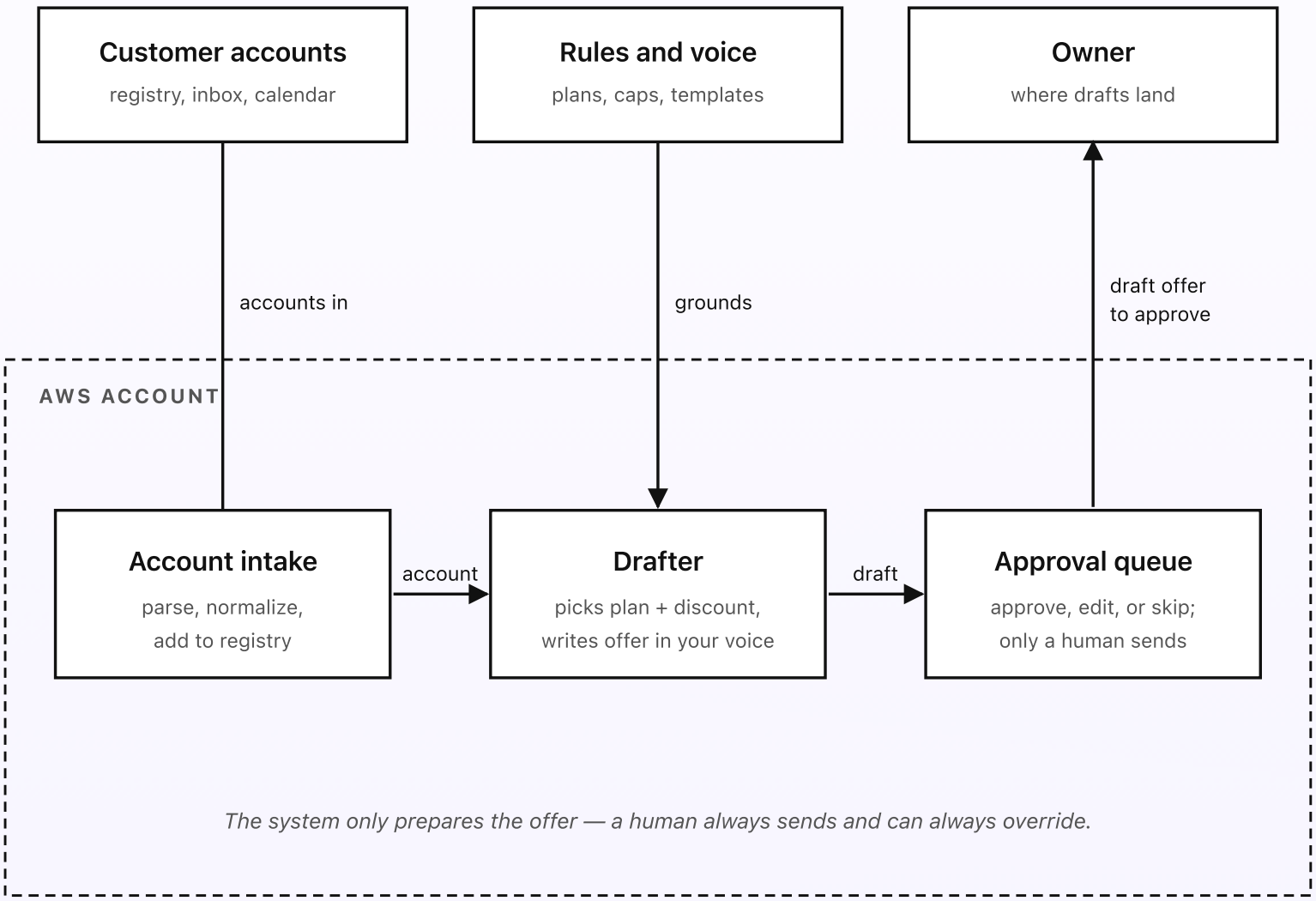

Customer accounts
registry, inbox, calendar
Rules and voice
plans, caps, templates
Owner
where drafts land
accounts in
grounds
draft offer
to approve
AWS ACCOUNT
Account intake
parse, normalize,
add to registry
account
Drafter
picks plan + discount,
writes offer in your voice
draft
Approval queue
approve, edit, or skip;
only a human sends
The system only prepares the offer — a human always sends and can always override.

Fig 1. Three sources outside, three pieces inside AWS. Accounts flow in from a Drive registry, an inbox forwarding lane, and a calendar import lane. The Drafter runs daily and prepares one offer per upcoming renewal. The Approval queue holds it for the owner; only a human sends.

What you set up once (the outside)

- **Customer accounts.** A Google Sheet in a Drive folder, one row per account: name, plan, current price, renewal date, a usage signal (seats used, hours billed, orders placed — whatever fits your business), owner email, and a short history (last renewal, last discount given, any notes). You can fill it in once from your billing export and forget it; new accounts can also enter via two other lanes covered in Part 2 — an inbox-forwarding lane (forward a signed contract or a usage export and the system proposes a row for one-tap approval) and a calendar import lane (renewal dates tagged in Google Calendar with `#renews` get pulled in automatically).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc holds your plan menu (what each tier costs and includes), your discount caps per tier ("small accounts: up to 10% loyalty discount; mid: up to 15%; never below the floor price"), how far ahead of the renewal date to prepare an offer, and which accounts to always skip (the ones you handle by hand). The *voice* doc holds one offer template per situation — the tone and shape of what the email to the customer should say. The model fills these in; it never makes up the numbers.
- **Owner.** The person responsible for renewals. Each drafted offer lands in their approval queue — an email or a Slack message with the account, the proposed plan, the discount (always inside the cap), the short reasoning, the full draft

email, and three buttons: *approve & send*, *edit*, and *skip*. The customer never hears from the system until the owner presses send.

What runs on every check (the inside)

- **The account intake.** Three sources feed the registry. The Drive sheet itself is the canonical store. New accounts can also be added via the inbox forwarding lane (forward a contract PDF or a CSV usage export to `renewals@your-company.com`; the system reads it and drops a one-tap approval card so a rep can confirm before the row is added) and the calendar import lane (renewal dates tagged `#renews` get pulled hourly by a small sync Lambda).
- **The drafter.** Runs once a day. Reads the registry. For each account whose renewal date falls inside the prepare-ahead window, it gathers the context — current plan and price, the usage signal, the history — and picks a move with plain Python: which plan to propose and which discount band to use, both straight from the rules doc. A heavy user near a plan ceiling gets an upgrade suggestion; a steady small account gets a modest loyalty discount inside the cap; a shrinking account might get a right-size-down offer to keep them. Only then does it call Bedrock to write the offer in your voice. The model gets the chosen plan and discount as fixed inputs; it writes the words, not the numbers.
- **The approval queue.** Each finished draft is written to a queue and a message goes to the owner. Nothing is sent to the customer. The owner can approve and send as-is, open the draft to edit wording or nudge the discount (still inside the cap), or skip the account this cycle. Every choice is logged. A weekly digest lists what's waiting and what was sent; a monthly summary writes a short narrative: renewals coming up, offers sent, accepted, and lost.

In plain words

Your customer Bayside Studio is on the Pro plan at $180/month and renews August 1. They've been with you two years, their seat usage is near the Pro ceiling, and they've never been given a discount. On July 2 (30 days out) the drafter prepares an offer: the rules say a heavy user near a ceiling gets an upgrade nudge plus a 10% loyalty discount, so it proposes the Business plan at $240/month with 10% off the first year. Bedrock writes a warm two-paragraph email in your voice that thanks them for two years, notes they're bumping the Pro limits, and frames the upgrade as the cheaper-per-seat option. It lands in your queue. You read it, change one sentence, and press send. Bayside replies a week later and upgrades. The account row updates with the new plan, price, and renewal date, and the cycle starts again next year.

The cost of running this is about $3 a month at SMB volume. The cost of *not* running it is the renewal that lapsed in silence, the heavy user you never upsold, and the loyal account who left over a price bump you would gladly have softened.

DESIGN RULES THAT SHAPED EVERY DECISION

- The system only prepares an offer. A human approves and sends every single one — there is no auto-send.
- The rules doc owns the numbers. Plans and discount caps come from it; the model never invents a price.
- Every discount is checked against the cap before the draft is queued. A draft can't exceed your limit.
- Each draft ships with the account, the proposed plan, the discount, and the reasoning. The owner never has to dig.
- The registry lives in Drive. Changing a plan, a cap, or an owner doesn't need a deploy.
- Every action is logged. Audit a renewal next year and you can see the offer, who sent it, and what changed.

Why this shape

Most teams handle renewals in one of three ways: a reminder that fires on the day with no offer attached, a generic price-increase email blasted to everyone, or nothing at all until the customer asks to cancel. The day-of reminder is too late to do anything useful. The generic blast treats a two-year heavy user the same as a trial that never converted — and annoys both. And "nothing" is how good customers slip away quietly.

The setup above moves the work to where it pays off: early, and tailored. The system gets in front of each renewal with enough lead time to actually have a conversation. It tailors the offer to the account — the plan that fits their usage, a discount sized to their loyalty and your margins. And critically, it stops short of sending. A renewal offer is a commercial promise; it should never leave the building without a person reading it. The drafter does the gathering and the first draft — the slow part — and leaves the judgment to you.

The next four posts walk through each piece in turn: how a renewal gets prepared, how an offer gets drafted, how the offer reaches the owner, and how a renewal gets sent or skipped. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 11, 2026 PART 2 OF 7 · RENEWAL NEGOTIATOR SERIES ~4 MIN READ

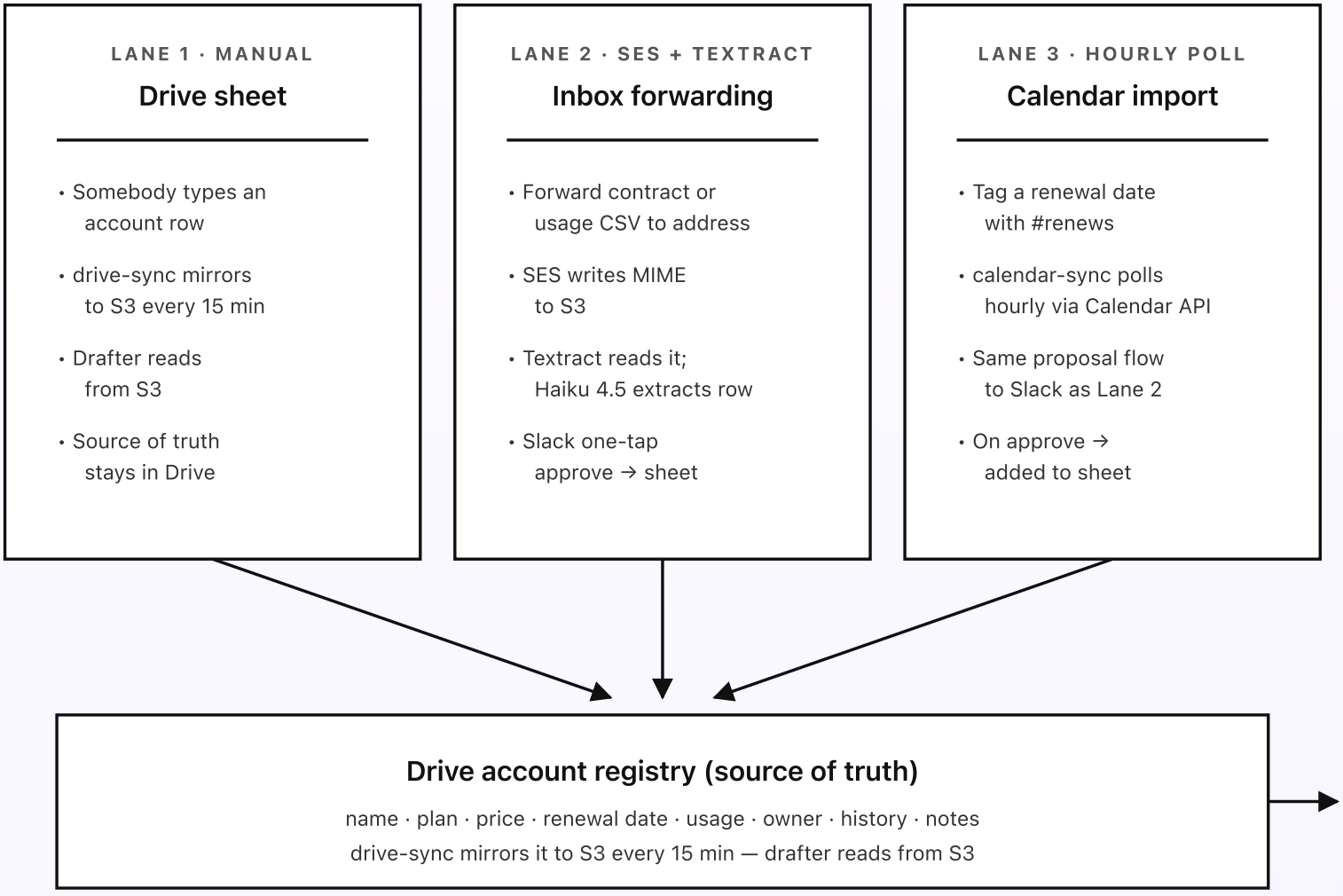
How a renewal gets prepared

The drafter only works on accounts that are in the registry, with the facts the offer needs — plan, price, renewal date, a usage signal, and a little history. So the first job is making sure the registry actually reflects your customers. There are three ways an account gets in: somebody types a row in the Drive sheet, somebody forwards a contract or a usage export to a dedicated address, or somebody puts a renewal date on a Google Calendar with a small tag. The first one is obvious. The other two exist because in real life nobody types a row for the deal they just closed.

KEY TAKEAWAYS

- Three intake lanes feed one registry: the Drive sheet, an inbox-forwarding lane, and a calendar import.
- Forwarded contracts are parsed by Textract; usage CSVs are read directly; Bedrock Haiku 4.5 proposes a row.
- Every parsed row goes to a rep's Slack for one-tap approval before it lands in the registry.
- Calendar events tagged `#renews` get pulled hourly via the Google Calendar API.
- The registry stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one registry



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 calendar lane are conveniences that propose rows for human approval. The drive-sync Lambda mirrors the sheet to S3 so the drafter can read it without hitting Drive on every check.

Lane 1: the Drive sheet itself

The simplest lane. Open the registry sheet in Drive, add a row, save. The columns are short: name, plan, current price, renewal date, usage signal, owner email, and a short history. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://rn-registry-source/registry.csv` if the sheet has changed since the last sync. The drafter 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 bulk of your setup: export your billing system once, paste in the accounts, and you're done. Most existing customers go in this way on day one.

Lane 2: inbox forwarding (the lane most teams actually use)

Set up a dedicated inbound address — something like `renewals@your-company.com` — via Amazon SES. Anyone on the team forwards a signed contract PDF or a usage-export CSV to that address and the system takes it from there. SES writes the raw MIME to `s3://rn-raw-mime/`. The S3 PUT triggers a parser Lambda. The Lambda walks the MIME tree to the attachment. For a PDF it runs Amazon Textract (Textract reads PDF, PNG, JPEG, and TIFF natively); for a CSV or spreadsheet it reads the rows directly with `openpyxl` or the CSV module. Either way it gets back the extracted text and tables.

Then a Bedrock Haiku 4.5 call reads the content and emits a structured row: name, plan, price, renewal date, usage signal (if present), and an owner-suggestion based on the "To" line of the original forward. The model prompt is short: "Extract an account row for the registry. Return JSON only. Mark each field with a confidence score. Do not invent a renewal date or a price that isn't in the document." The output goes to a small Slack interactive message that pings the rep who forwarded the email: 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 rep gets a fillable modal pre-populated with the proposal. On *discard*, the message is logged and the file moved to a discarded prefix in S3 for audit.

The reason every parsed row goes to a human first is simple: a renewal date or a price the model misread is worse than an account that never made it into the registry at all. The misread one will quietly drive a wrong offer to a real customer.

Lane 3: calendar import

Some teams already track renewal dates on a calendar. The big enterprise deal is on the account manager's calendar with a reminder. The annual retainer is on the founder's calendar. Forcing those teams to also type rows in a sheet is a fight you don't need to have on day one.

Lane 3 picks up calendar events tagged with `#renews` in the description. A small `calendar-sync` Lambda runs hourly, iterates through the configured Google Calendars (using a service-account credential stored in Secrets Manager), and pulls any events with the tag whose start time is in the future. Each pulled event becomes a proposal in the same Slack flow as Lane 2 — one-tap approve to add

to the registry. Once approved, the calendar event can stay where it is or be deleted; the registry now owns the renewal.

Calendar import is the most opt-in of the three lanes. A team that doesn't use it loses nothing; a team that does avoids retyping dates they already typed once.

Why the registry stays the source of truth

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

Next post: how the drafter reads an account, picks the plan and discount from your rules, and calls Bedrock to write the offer in your voice.

PART 3 OF 7

JUNE 11, 2026 PART 3 OF 7 · RENEWAL NEGOTIATOR SERIES ~5 MIN READ

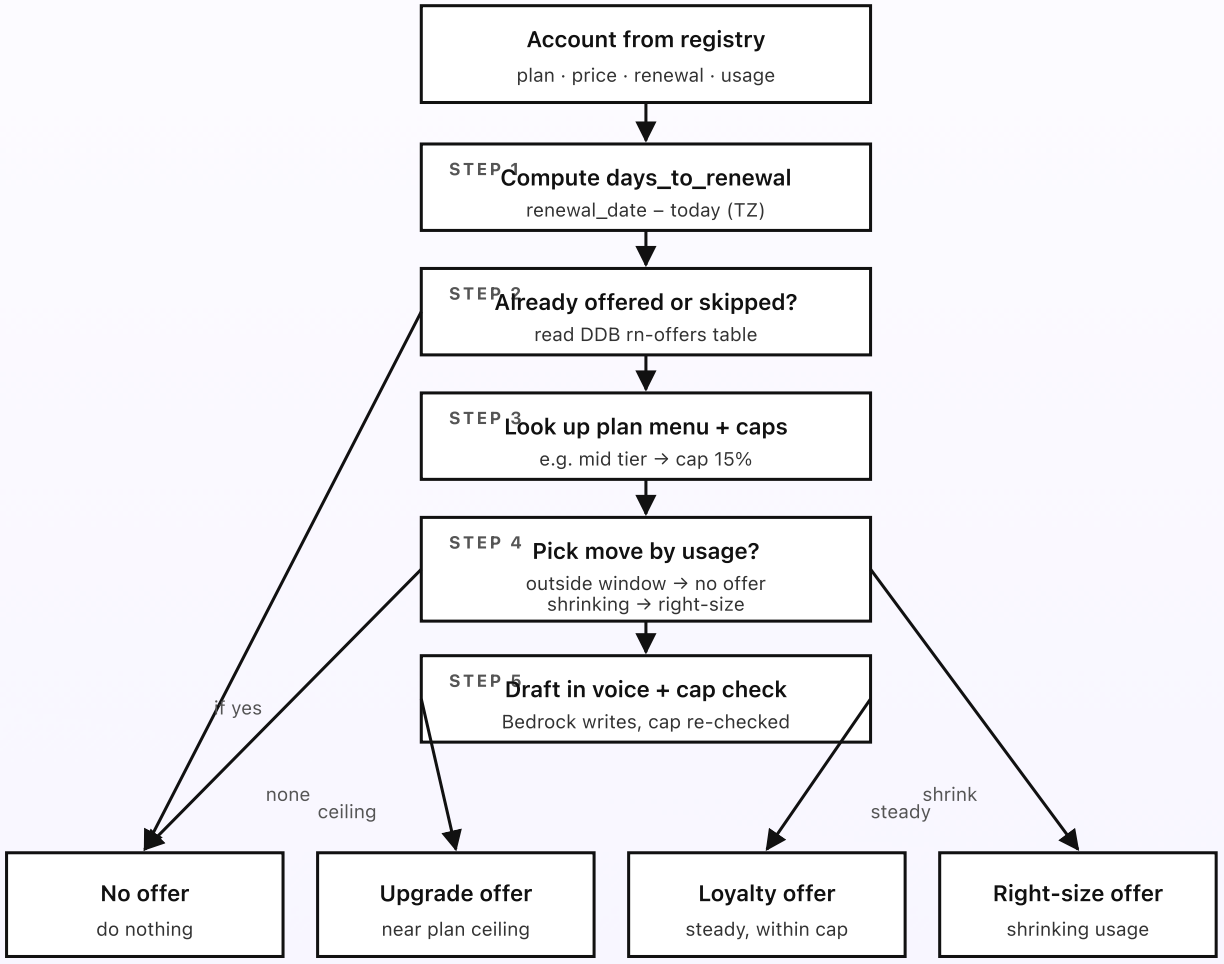
How a renewal offer gets drafted

Once a day, an EventBridge Scheduler rule fires the drafter Lambda. The Lambda reads the registry, looks at one account at a time, computes how many days until the renewal, and decides whether to do nothing or to prepare an offer — and if so, which kind. The choice of plan and discount is plain Python reading your rules. Only after the numbers are fixed does the model get involved, and only to write the words. Every threshold and cap lives in the rules doc, where you can edit it without a deploy.

KEY TAKEAWAYS

- The drafter runs once a day via EventBridge Scheduler and only acts on renewals inside the prepare-ahead window.
- The plan and discount band come from your rules doc — tiers, caps, and floor prices live there, not in code.
- Four moves per account: no offer, upgrade offer, loyalty offer, or right-size offer.
- The model writes the email in your voice from the chosen plan and discount — it never sets a number.
- Every drafted discount is checked against the cap before the draft is queued.

The decision flow, per account



The rules doc holds every plan, cap, and floor — the model only writes the words around them.

Fig 3. The drafter's decision tree, per account, per daily check. Five steps decide which of four moves applies. The rules doc holds every plan, cap, and floor; the drafter only enforces them and the model only writes the words.

Plans and caps live in the doc, not the code

The rules doc has one short section per tier. Each names the plans and the limits in plain prose: "Small accounts (under $100/month): loyalty discount up to 10%, never below the floor of $69. Mid accounts ($100–$400): up to 15%. Enterprise: route to a human, no auto-draft." It also names the upgrade path for each plan ("Pro upgrades to Business at $240") and the prepare-ahead window ("start drafting 30 days before the renewal date"). The numbers are yours; the drafter reads them and never exceeds them.

The caps exist for a reason. A loyalty discount is a tool, not a giveaway — a 10% cap on a small account protects your margin while still feeling generous. A floor price stops a stack of discounts from ever taking a plan below what it costs you to serve. And routing enterprise to a human keeps the highest-stakes negotiations where a person belongs. Per-account overrides exist too: the registry has an optional `cap_override` column for the one customer you've agreed special terms with.

Four moves, always

Every account inside the prepare-ahead window lands in exactly one of four buckets. The names are simple on purpose.

- **No offer.** The renewal is still outside the window, or the account has already been offered or skipped this cycle, or the tier is one you route to a human. Do nothing. Most accounts, most days, are no-offer.
- **Upgrade offer.** The usage signal is near the plan ceiling — seats nearly full, hours nearly used up. The rules pick the next plan and a discount inside the cap. The pitch frames the upgrade as the better-value option, not a price hike.
- **Loyalty offer.** Steady usage, a loyal account, no upgrade needed. The rules pick a modest renewal discount inside the cap to thank them and keep them. The most common shape for healthy small accounts.
- **Right-size offer.** The usage signal is shrinking — the customer is using less than they pay for and is a churn risk. The rules pick a smaller plan that fits their real usage, keeping the relationship instead of losing it to a cancel button. Counterintuitive, but a smaller paying customer beats a churned one.

Where the model comes in (and where it doesn't)

By the time Step 5 runs, the plan and the discount are fixed numbers from the rules. The drafter passes them to Bedrock with the account context and the matching voice template, and asks for one thing: an email in your voice that presents this exact offer. The prompt is explicit — "Use only the plan and discount given. Do not invent prices, percentages, or terms. Write warmly, in the supplied voice, two short paragraphs." Most accounts use Claude Haiku 4.5; a flagged-tricky account (a long history of back-and-forth, a sensitive note in the row) uses Claude Sonnet 4.6 for the heavier reasoning. The model returns the draft text only.

After the draft comes back, the drafter does one last deterministic thing: it re-reads the discount it asked for and confirms it's still inside the cap and above the floor. If a number somehow drifted, the draft is rejected and the account is flagged for a human rather than queued. The check is cheap and it's the backstop that makes "the model never sets the price" true in practice, not just in intent.

Why split the numbers from the words

The drafter could ask the model to pick the discount too. It doesn't, for the same reason a good sales manager sets discount authority in advance: pricing is a commercial decision with margin consequences, and you want it to be the same every time for the same situation. A model deciding the discount introduces variance you can't reason about and a customer could later dispute. Letting plain Python own the numbers and the model own the prose gives you both — consistent pricing and a warm, tailored email — with a clean line between the two.

Next post: how the finished draft reaches the owner — owner resolution, the discount-cap gate, quiet hours, and how the draft lands in the approval queue without ever touching the customer.

PART 4 OF 7

JUNE 11, 2026 PART 4 OF 7 · RENEWAL NEGOTIATOR SERIES ~5 MIN READ

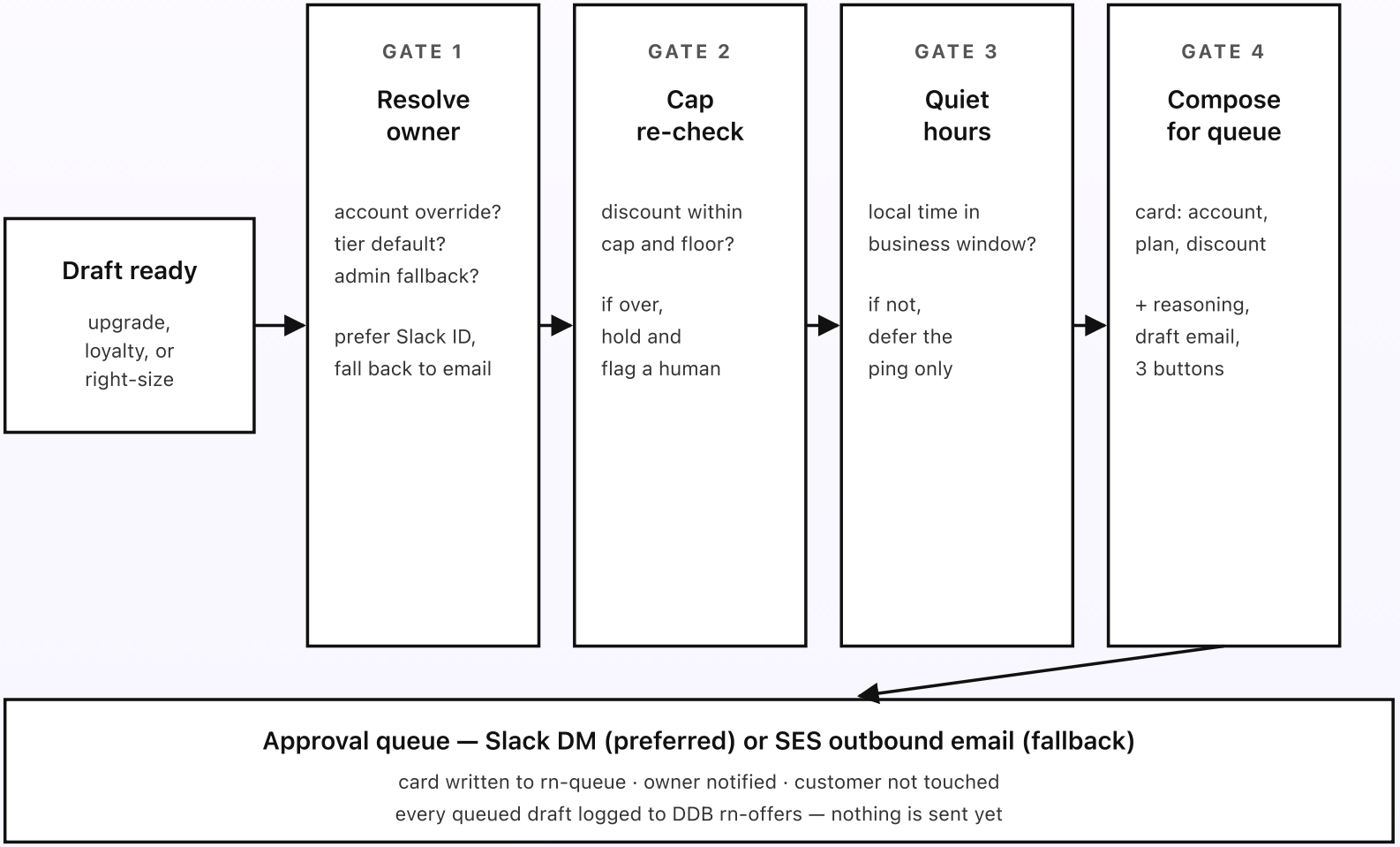
How a renewal offer reaches the owner

The drafter wrote an offer — an upgrade, a loyalty discount, or a right-size. Now it has to reach the right owner, with the discount double-checked, at a sensible time, in a form they can act on in ten seconds. Get any of those wrong and you waste a renewal: a draft routed to the wrong rep, a discount that slipped past the cap, a 2am ping nobody reads. Four small guardrails sit between the finished draft and the owner's approval queue. The customer is never touched in this path.

KEY TAKEAWAYS

- Owner resolution: per-account override beats per-tier default beats fallback to the configured admin.
- A discount-cap gate re-checks every offer; anything over the cap is held, never queued silently.
- Quiet hours defer the owner ping to the next business hour — the draft still waits in the queue.
- Every draft reaches the owner with the account, plan, discount, reasoning, and the full email text.
- The offer goes to the owner's queue, not the customer. Only a human send in Part 5 reaches the customer.

Four guardrails before the queue



Nothing in this path reaches the customer — the offer waits in the owner's queue until a human sends it.

Fig 4. Four guardrails between the drafted offer and the approval queue. Resolve the owner. Re-check the discount cap. Honor quiet hours for the ping. Compose the queue card. Then write it to the queue and notify the owner — the customer is never touched here.

Gate 1: resolve the owner

Three places the gate looks for the owner of an account, in order. First, the registry sheet's per-account `owner_email` column — if a row has a specific rep assigned, that rep owns the renewal regardless of the tier default. Second, the per-tier default in the rules doc ("all mid-tier accounts default to the account manager"). Third, the configured admin fallback — the person who set up the system and gets every unowned draft. The fallback should rarely fire; if it does, the weekly digest names every account that hit it so the rules doc can be updated.

Once the gate knows which person to notify, 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 a DM with buttons is faster to act on than an email link. Email is the fallback so nobody falls through the cracks. Either way, the message is to the *owner* — never to the customer.

Gate 2: the discount-cap re-check

Part 3 already checked the discount against the cap inside the drafter. Gate 2 checks it again, on purpose. Defense in depth matters most for the one thing that can't be taken back — a discount that went out too rich. This gate re-reads the discount and the resulting price straight off the draft and compares them to the

per-tier cap and the floor price in the rules doc. If the discount is at or under the cap and the price is at or above the floor, it passes. If either fails, the draft is not queued for one-tap approval; it's held and flagged for a human to look at, with a note explaining which limit it tripped.

This is the gate that makes the system's core promise real: a draft can never quietly offer more than your rules allow. Even a bug in the drafter, or a strange model output that slipped through the first check, gets caught here before it can reach a customer-facing queue card.

Gate 3: quiet hours

The drafter runs at a fixed hour, so most drafts are ready during business hours. But deferred drafts and re-runs can finish later. Gate 3 reads the rules doc's quiet-hours setting (default 6pm to 8am, configurable). If the current local time is in the quiet window, the gate writes the draft to the queue anyway — it just defers the *notification* to the owner. It creates a one-off EventBridge Scheduler rule that pings the owner at the next business-hour minute. The draft is never lost; only the buzz waits. An owner who opens the queue early still sees everything waiting for them.

Gate 4: compose the queue card, then write it

The queue card is what the owner actually sees. Gate 4 assembles it: the account name and current plan, the proposed plan and price, the discount and which cap it sits under, the one-line reasoning ("heavy user near the Pro ceiling; loyalty discount 10%"), and the full draft email exactly as it will go to the customer if

approved. Below that sit three buttons — *approve & send*, *edit*, *skip*. For Slack, this is a Block Kit message; for email fallback, it's an HTML card with links that hit a Function URL. The card is written to the `rn-queue` DynamoDB table and a row is logged to `rn-offers` marking the account as drafted-and-waiting.

Crucially, composing the card sends nothing to the customer. The draft email is shown to the owner for review; it leaves the building only when the owner presses send in Part 5. The whole of Part 4 is about getting a good draft in front of the right person, safely — not about delivery.

Why the guardrails exist

None of these gates are exotic. They're the care a thoughtful sales manager would take before handing a rep a renewal to send — check who owns it, confirm the discount is within authority, don't buzz anyone at midnight, and lay out everything they need to decide in one glance. Putting them in code as four small sequential gates makes them part of the design, not something you're trusting any one draft to remember. And the cap re-check, sitting in the middle, is the guardrail that keeps an automated drafter from ever becoming an automated giveaway.

Next post: what the owner can do with a draft — approve and send, edit first, or skip — and how each choice updates the account, the queue, and the audit trail.

PART 5 OF 7

JUNE 11, 2026 PART 5 OF 7 · RENEWAL NEGOTIATOR SERIES ~5 MIN READ

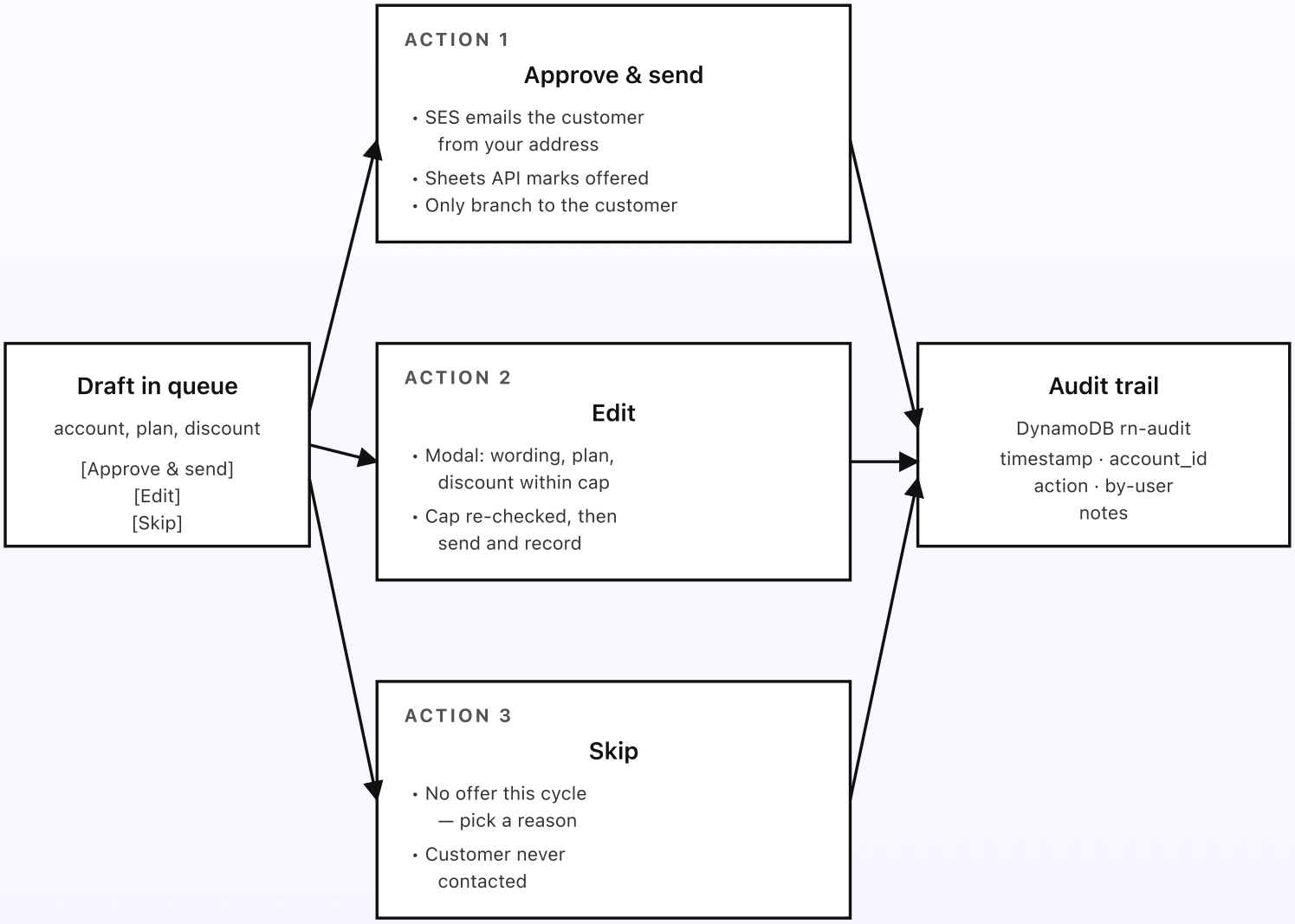
How a renewal gets sent or skipped

A draft for Bayside Studio lands in Maria's queue at 8:03am: upgrade to Business, 10% off, a warm two-paragraph email ready to go. There are three buttons. What happens when she taps one? This is the part of the system where the human makes the call. This post walks through the three things Maria can do — approve and send, edit, or skip — and how the account, the queue, and the audit trail all stay in sync. Nothing reaches the customer until she presses send.

KEY TAKEAWAYS

- Three actions per draft: *approve & send* (to the customer), *edit* (change then send), *skip* (no offer).
- Approve & send is the only path that emails the customer — via SES, from your address.
- Edit opens a modal; any discount change is re-checked against the cap before it can send.
- Skip closes the cycle cleanly — the account gets no offer and the reason is logged.
- Every action writes to the `rn-audit` table and is reversible from a snapshot.

Three actions on a draft



Only Approve & send (or a saved Edit) reaches the customer — Skip and an unopened draft never do.

Fig 5. Three actions per draft, three different effects. Approve & send emails the customer and marks the account offered. Edit changes the offer (re-checked against the cap) then sends. Skip closes the cycle with no offer. Every action writes to the audit trail.

Action 1: approve & send (the common path)

Maria reads the Bayside draft, agrees with it, and taps *Approve & send*. The button submits to a Function URL Lambda. Three things happen, in order. First, the offer email — the exact text she reviewed — is sent to the customer via SES outbound, from the owner's own sending address, so the reply lands in Maria's inbox like any normal email. Second, the Sheets API updates the account row: the offer is marked sent, with the plan and discount that went out and today's date in `last_offered`. Third, an `action: sent` row is written to `rn-audit` with the user, timestamp, and the full offer that was sent.

This is the only one of the three actions that contacts the customer. Everything earlier in the series — intake, drafting, the four guardrails — exists to make this one tap safe and fast. When Bayside replies and upgrades, Maria edits the registry row with the new plan, price, and renewal date (or forwards the signed paperwork to the inbox lane), and the cycle rolls forward to next year.

Action 2: edit (the tweak)

Often the draft is 90% right. Maria wants to add a personal line about the project Bayside shipped last quarter, or she knows this customer responds better to a round number than a percentage. She taps *Edit*. A modal opens pre-filled with the

draft email, the proposed plan, and the discount. She can change the wording freely. She can swap the plan from the menu. She can adjust the discount — but here's the one hard rule: any discount change is re-checked against the cap and floor from the rules doc before the modal will let her send. If she tries to go past the cap, the modal blocks it with a note ("that's above the 15% cap for this tier — lower it or ask for an exception").

On save, the edited offer is sent to the customer exactly as Action 1 would, and the audit row records both the original draft and the edited version, so next year's reviewer can see what the system proposed and what the human actually sent. The edit path is where the human's judgment and the system's consistency meet: the system did the heavy lifting, the person added the touch only they could.

Action 3: skip (the "not this one")

Sometimes the right move is no offer. The account is one Maria is negotiating by hand. The customer already gave notice and is leaving. The renewal is far enough out that she'd rather wait. She taps *Skip* and picks a reason from a short list — *handling by hand*, *already churned*, *not ready yet*, *other*. The registry row is marked skipped for this cycle with the reason, and an `action: skipped` row is written to `rn-audit`. The customer is never contacted.

Skip is bounded and honest. A skipped account doesn't silently disappear — it shows up in the weekly digest and the monthly summary as "skipped: handling by hand" so nothing falls through a crack. And *not ready yet* is special: it re-queues the account for a fresh draft a configurable number of days later (default 7), so "skip for now" doesn't become "skip forever." The most dangerous failure mode

for a renewal system is an account that quietly gets no attention; skip is built so that can't happen by accident.

Every action is logged, every action is reversible

The `rn-audit` table records every send, edit, and skip with the user who took the action, the timestamp, and a snapshot of the offer before and after. If an offer went out with a wrong figure, or an account was skipped that shouldn't have been, a rep can run an "undo last action" through a small admin command that reads the previous-state snapshot and restores the row (a sent email can't be unsent, of course, but the registry state and the cycle can be corrected, and a follow-up drafted). The undo is itself an audit row, so the trail of edits stays clean.

This reversibility matters most for the renewals you'll only think about once a year. The next time Bayside comes up, it might be Maria again or whoever took her seat. Either way, the audit trail is the only memory the next person has — what was offered, who sent it, and what changed.

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 one model call per renewal keeps the bill small.

PART 6 OF 7

JUNE 11, 2026 PART 6 OF 7 · RENEWAL NEGOTIATOR SERIES ~3 MIN READ

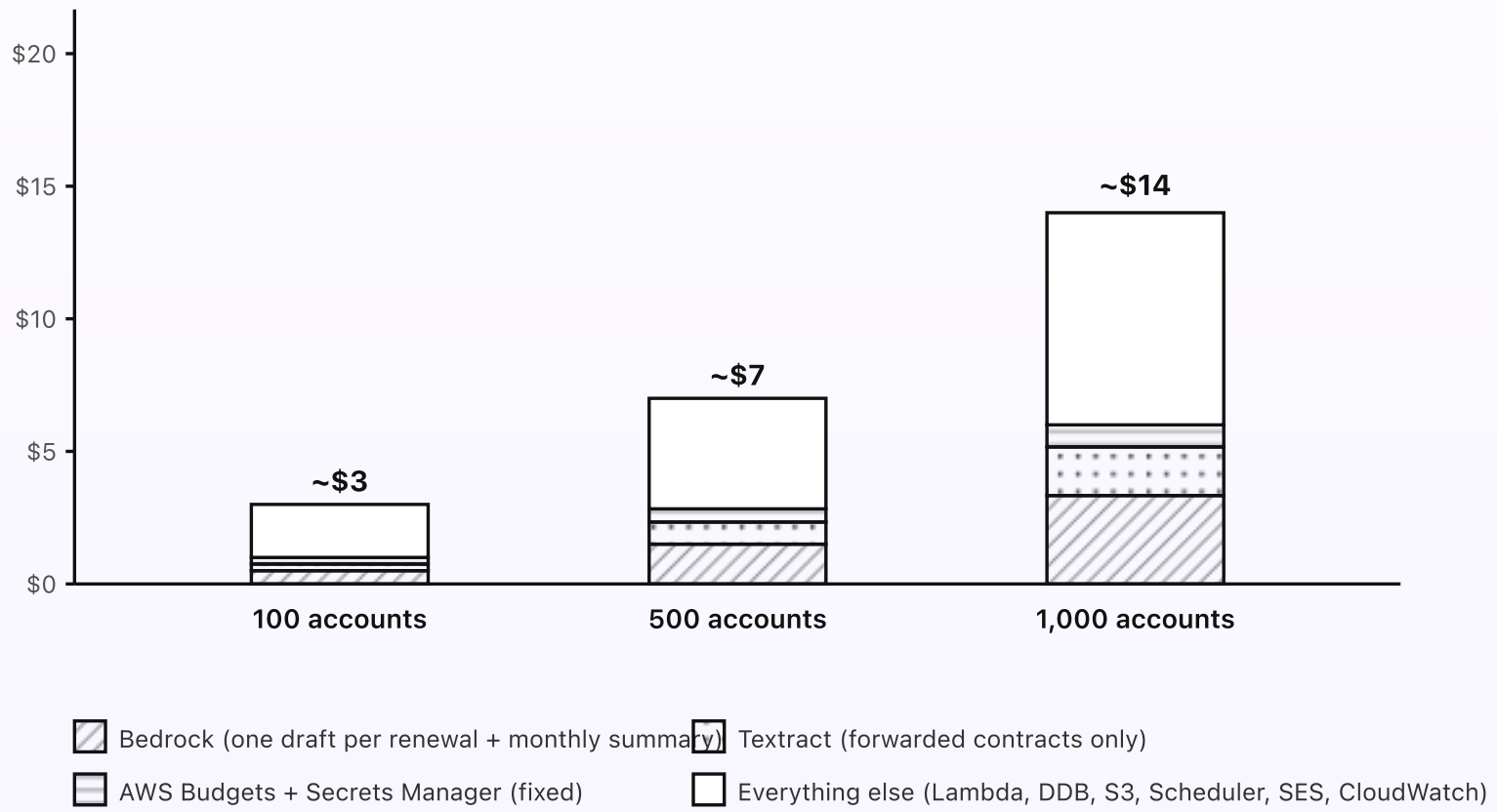
What the renewal negotiator costs

The negotiator is one of the cheaper systems in this series. The daily check reads a CSV from S3, does a little date arithmetic, and decides which accounts are coming due. It calls no model on that check. Bedrock fires once per upcoming renewal to write the draft, and once a month for the summary. Because each account renews about once a year, the model runs a small, steady number of times a month — not on every account every day. At typical SMB volume, the bill is a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $3/month at typical SMB volume (around 100 accounts renewing across the year).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The daily check costs pennies — no model calls; the model fires only on a draft per renewal.
- Bedrock is a larger slice here than in pure-alert systems, but still small: one draft per renewal.
- At 500 accounts the bill is around $7. At 1,000 accounts it's around $14.

Cost at three volumes



The daily check is the dominant cost — and the model runs only a small steady number of times a month.

Fig 6. Monthly cost at three account volumes. Bedrock is a visible but minority slice because it fires once per renewal, not per account per day. Textract is tiny — only forwarded contracts hit it. The dominant cost is the everything-else bucket: the daily check reading every account.

Where the dollars actually go

Lambda runtime (the bulk). The drafter runs once a day. Each check reads the registry CSV from S3, iterates the rows, computes `days_to_renewal` for each, and finds the few coming due. At 100 accounts that's a few hundred milliseconds; at 1,000 it's a second or two. Add the drafting Lambda firing once per upcoming renewal, the Function URL Lambda handling approve/edit/skip, the calendar-sync Lambda running hourly, and the drive-sync Lambda every fifteen minutes — the Lambda total still lands around a dollar at all three volumes.

Bedrock (the model). The daily check uses no Bedrock. The drafting step fires Claude Haiku 4.5 once per upcoming renewal: a few thousand input tokens (the account context and voice template) and a few hundred output tokens (the email), so a fraction of a cent per draft. Because each account renews about once a year, only a small slice of your accounts come due in any given month — roughly one-twelfth. So at 1,000 accounts you draft on the order of 80 renewals a month, not 1,000 a day. A handful of tricky accounts use Claude Sonnet 4.6, which costs more per call but stays a minor share. The monthly summary is one larger call. Bedrock is a bigger slice here than in a pure alert system, but it's still measured in cents to a couple of dollars.

DynamoDB on-demand. Four small tables: `rn-offers`, `rn-queue`, `rn-audit`, and a state table. Reads are dominant during the daily check; writes are queue cards and audit rows. Pennies a month at any of these volumes.

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

EventBridge Scheduler. The daily check rule plus deferred owner-ping rules from the quiet-hours gate. A few invocations a day. Pennies.

SES. Inbound for the forwarding lane and outbound for the offer emails: $0.10 per thousand messages each way. At a few dozen offers a month, a few cents.

Textract (only on forwarded contracts). Per-page pricing; a typical contract is two to ten pages. A few cents per parse. At a handful of forwarded contracts a month, Textract is cents.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the approve, edit, and skip endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The drafter sleeps almost the whole day.
- **A Knowledge Base.** The registry is structured rows, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the check.** The daily find-what's-due is plain Python. Bedrock fires only on a draft per renewal and the monthly summary.

How the cost scales

Lambda runtime grows roughly linearly with account count, because every account is read on every check. DynamoDB grows linearly too. Bedrock grows

with the *number of renewals per month* — roughly one-twelfth of your accounts — not with the daily read, so it stays a minority slice. So the bill at 5,000 accounts is around $45; at 10,000 it's around $90. Past those volumes the daily full-read probably stops being right (you'd switch to a partial check that only scans accounts inside the prepare-ahead window), but that's an optimization for large books — not a redesign.

Set an AWS Budgets alarm at $20/month so anything unusual pages you before the bill matters. The negotiator's normal-volume bill stays well under that ceiling.

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

PART 7 OF 7

JUNE 11, 2026 PART 7 OF 7 · RENEWAL NEGOTIATOR SERIES ~8 MIN READ

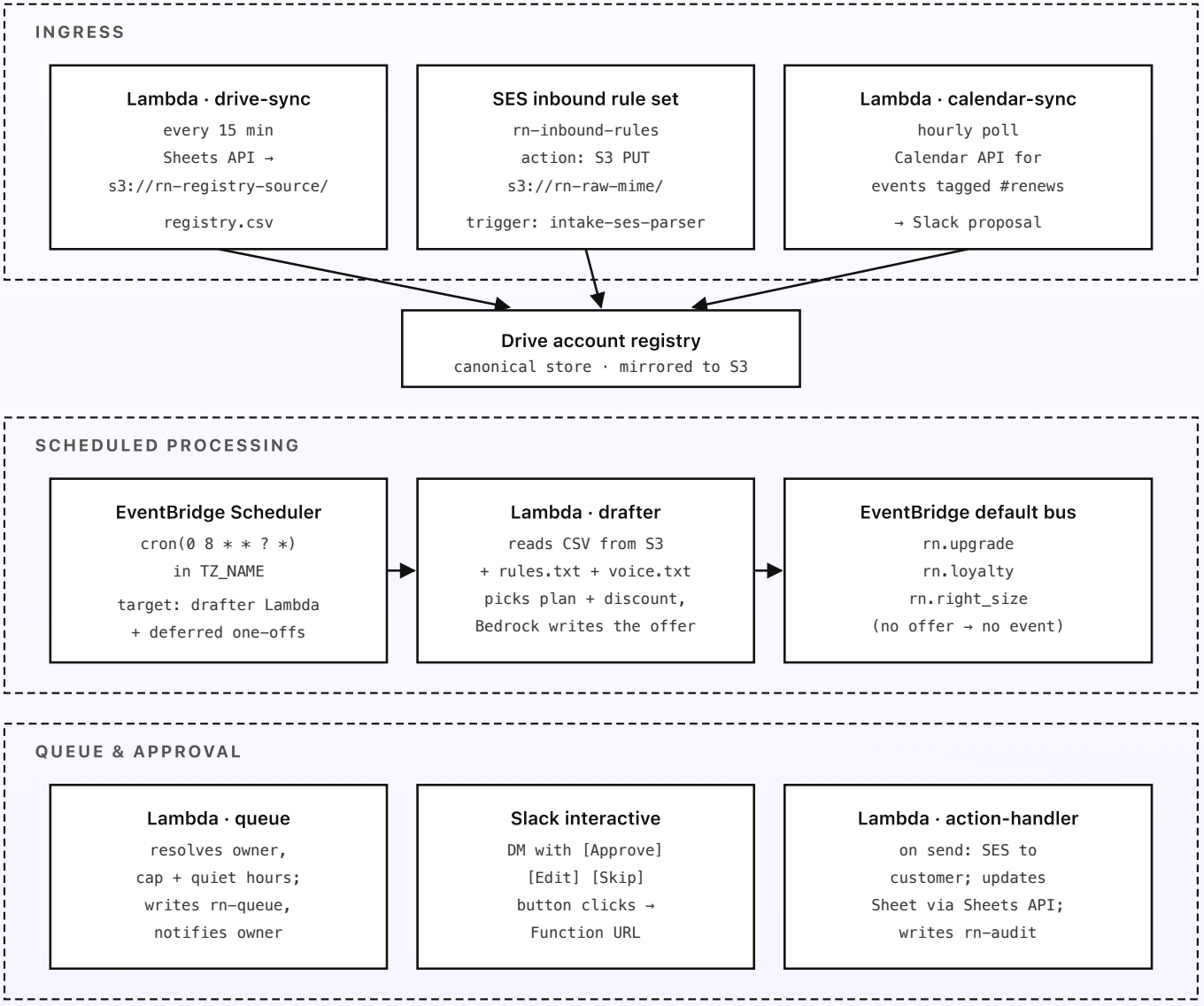
Engineering reference: the renewal negotiator architecture

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 a renewal offer not getting drafted on time, not a regional outage. One AWS account dedicated to the negotiator (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Only a human send reaches the customer — and every interaction is logged to rn-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the registry), scheduled processing (the daily drafter emitting draft events), queue and approval (the offer waits, then a human send reaches the customer). 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 `rn/drive/sa`) to export the registry sheet as CSV and write to `s3://rn-registry-source/registry.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://rn-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`calendar-sync`** — EventBridge Scheduler target, hourly. Uses the Google Calendar API `events.list` to scan configured calendars for events with `#renews` in the description; for any new events, creates a Slack interactive proposal message. For lower-latency setups you can switch to `events.watch` and have Calendar push notifications to a Function URL instead of polling, at the cost of renewing the channel before it expires (Calendar push channels have a finite TTL and need a small refresh job). Memory: 256 MB. Timeout: 30 s.

- **`intake-ses-parser`** — S3 PUT trigger on `s3://rn-raw-mime/`. Parses MIME, extracts the attachment. For PDFs, runs Textract via `StartDocumentTextDetection` + `StartDocumentAnalysis` (asynchronously to handle multi-page contracts); 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 account row. For CSV or XLSX usage exports (Textract isn't needed), reads rows directly with the CSV module or `openpyxl`. Posts the proposal to Slack via the incoming webhook with Approve/Edit/Discard buttons. Both packages are stable and widely used in 2026, though their maintenance velocity is light — acceptable for a path that runs a few times a month. Memory: 512 MB. Timeout: 60 s.
- **`drafter`** — EventBridge Scheduler target, daily at 8am local time (the schedule expression runs in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). Reads `s3://rn-registry-source/registry.csv` and the rules and voice docs. For each row, computes `days_to_renewal`; for accounts inside the prepare-ahead window, reads offer state from `rn-offers`, picks the plan and discount band with plain Python from the rules, calls Bedrock to write the offer, then re-checks the discount against the cap and floor. Emits one event per account needing a draft: `rn.upgrade`, `rn.loyalty`, or `rn.right_size`, with the account context and the fixed plan/discount as the event payload. No-offer accounts emit nothing. Memory: 512 MB. Timeout: 120 s. *Bedrock callsite.*
- **`queue`** — EventBridge rule on the three draft events. Resolves owner, re-checks the cap and floor, checks quiet hours, formats the queue card from the voice template, writes it to `rn-queue`, and notifies the owner via Slack `chat.postMessage` (`rn/slack/bot-token` in Secrets Manager) or SES

`SendRawEmail`. On a quiet-hours defer, creates a one-off EventBridge Scheduler rule that re-invokes the owner notification at the next available business minute; the card itself is queued immediately. Writes a row to `rn-offers` marking the account drafted-and-waiting. Memory: 256 MB. Timeout: 30 s. *No Bedrock; no customer contact.*

- **`action-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive button clicks (Approve/Edit/Skip) and by email-link clicks. On *approve* or a saved *edit*, sends the offer to the customer via SES `SendRawEmail` from the owner's verified identity, updates the Drive sheet via the Sheets API (mark offered, plan, discount, `last_offered`), and writes `rn-audit`. An *edit* re-checks any discount change against the cap before sending. On *skip*, records the reason in `rn-offers` and `rn-audit`; *not-ready* schedules a re-draft via a one-off rule. Memory: 256 MB. Timeout: 15 s.
- **`digest`** — EventBridge Scheduler target, weekly Sunday 6pm. Reads `rn-offers` and `rn-queue` for the past week and the registry; sends a digest message to a configured Slack channel summarizing offers sent, accepted, skipped, and renewals coming up. 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 `rn-offers` and `rn-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph board narrative on renewals offered, accepted, and lost; emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB · `rn-offers`** — one row per account per cycle. PK `(account_id, cycle_id)`; attributes: `state` (drafted/queued/sent/skipped), `plan`, `discount`, `drafted_at`, `skip_reason` (if skipped). On-demand. No TTL.
- **DynamoDB · `rn-queue`** — one row per queued draft awaiting the owner. PK `account_id`; attributes: `card_json`, `owner`, `email_body`, `plan`, `discount`, `queued_at`. On-demand. Items deleted on send or skip.
- **DynamoDB · `rn-audit`** — one row per write action of any kind. PK `(account_id, ts)`; attributes: `action` (sent/edited/skipped/undo), `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB · `rn-state`** — small per-account scheduling state (last sync hash, next re-draft time for not-ready skips). PK `account_id`. On-demand.
- **S3 · `rn-registry-source`** — mirrored CSV from the Drive registry sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3 · `rn-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3 · `rn-raw-mime`** — raw inbound MIME from forwarded contracts and usage exports. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3 · `rn-source-files`** — parsed source contracts and exports after the inbound parser handles them, kept for reference if the registry row links to one.

Bedrock

- **Foundation models.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0` for the inbound parsing, the per-renewal draft, and the monthly

summary. For accounts flagged tricky (long negotiation history, sensitive note), the `drafter` escalates to `anthropic.claude-sonnet-4-6-20250930-v1:0` via `global.anthropic.claude-sonnet-4-6-20250930-v1:0` for the heavier reasoning. The model receives the plan and discount as fixed inputs and writes prose only; it never sets a number.

- **Embeddings.** Not used. The registry 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 daily check doesn't call Bedrock; drafts fire roughly one-twelfth of accounts per month.

EventBridge Scheduler config

- **`rn-daily-check`** — `cron(0 8 * * ? *)` in the SMB's timezone. Target: `drafter` Lambda.
- **`rn-drive-sync`** — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- **`rn-calendar-sync`** — `rate(1 hour)`. Target: `calendar-sync` Lambda.
- **`rn-weekly-digest`** — `cron(0 18 ? * SUN *)` in TZ. Target: `digest` Lambda.
- **`rn-monthly-summary`** — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `queue` (deferred owner ping) and `action-handler` (re-draft after a not-ready skip). 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. `renewals.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `rn-inbound-rules`: one rule with recipient `renewals@your-company.com` → spam scan → S3 PUT to `s3://rn-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the offer emails and the email-fallback queue notifications: verify each owner's sender identity (e.g. `maria@your-company.com`) with DKIM and SPF on the parent domain, so customer replies land in the owner's real inbox. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **drafter role:** `s3:GetObject` on the registry, rules, and voice keys; `dynamodb:Query` + `GetItem` on `rn-offers`, `rn-state`; `bedrock:InvokeModel` on the Haiku and Sonnet ARNs; `events:PutEvents` on the default bus.
- **queue role:** `events:ListSchedules` + `CreateSchedule` for deferred owner pings; `secretsmanager:GetSecretValue` on the Slack bot token; `ses:SendRawEmail` for email-fallback notifications; `dynamodb:PutItem` on `rn-queue`, `rn-offers`; outbound network access to `slack.com`. *No* customer-facing send rights.
- **action-handler role:** `dynamodb:PutItem` on `rn-audit`, `rn-offers`; `dynamodb:DeleteItem` on `rn-queue`; `secretsmanager:GetSecretValue` on

the Sheets-API and Slack secrets; `ses:SendRawEmail` from the verified owner identities; outbound network access to `sheets.googleapis.com`; `events:CreateSchedule` for not-ready re-drafts.

- **intake-ses-parser role:** `s3:GetObject` on `rn-raw-mime`; `textract:StartDocumentTextDetection` + `StartDocumentAnalysis`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token.
- **drive-sync and calendar-sync roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the registry 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 queue cards 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 (`approve_send`, `edit`, `skip`), opens a modal if needed (Edit and Skip open modals; Approve is one-tap), and processes the response when the modal is submitted. The Edit modal enforces the discount cap before allowing submit.

The Slack app needs `chat:write`, `im:write`, and the Interactivity URL configured. The bot token lives in Secrets Manager under `rn/slack/bot-token`. The signing secret is `rn/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:** drafter Lambda failures > 0 in a day (the daily check is the one piece that has to run); `action-handler` signature-verification failures > 5/hour (might mean the Slack secret rotated); any cap-breach hold logged by the `queue` Lambda (should be rare; a spike means a drafter bug).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $20/month threshold, alarm at 80% and 100%, posts to SNS topic `rn-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive, Sheets, and Calendar APIs all live in Secrets Manager under `rn/drive/sa` (one service account with scopes for all three APIs). Slack bot token and signing secret under `rn/slack/*`. SES sender identities live in IAM and the verified-domain config. The configured timezone, quiet-hours window, prepare-ahead window, per-tier discount caps and floor prices, and admin fallback owner all live in Parameter Store under `/rn/config/` (the rules doc is the human-editable source; Parameter Store holds the deploy-time defaults). 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) running AWS SAM. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `rn-registry-source` and `rn-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 check in UTC after a CI rotation. 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.