7-PART SERIES · FREE COMPANION

Document expiry watcher

A serverless watcher that tracks every contract, certificate, insurance policy, license, lease, and software subscription your business renews; pings the right owner with full context before each one lapses; escalates if nobody acts. The owner can renew, snooze, or ack-only right from the alert. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Document expiry watcher

PART 1 OF 7

MAY 5, 2026 PART 1 OF 7 · DOCUMENT EXPIRY WATCHER SERIES ~5 MIN READ

A document expiry watcher on AWS for a few dollars a month

A small business has more renewals than anyone keeps in their head. The vendor contract that auto-renews on May 31 unless someone gives 60 days' notice. The cyber-insurance policy that lapses on the 14th and is no fun to explain to the next prospect who asks about it. The food-handler certificate, the lease, the domain, the SOC 2 audit window, the workers' comp policy, the three software subscriptions that quietly switched to annual when nobody was watching. This post walks through the design of a small watcher that tracks all of it, pings the right owner with enough context to act, and escalates if nobody does.

KEY TAKEAWAYS

- Three sources for tracked items: a Drive registry, an inbox forwarding lane, and a calendar import lane.
- Every item ends in one of four moves on each tick: healthy, first alert, reminder, or escalate.
- Per-category rules: legal contracts get a 90/60/30/14/3-day chain, insurance gets 60/30/7, software gets 30/14/3.
- Pings respect quiet hours and your holiday calendar. Acknowledged items stop pinging.
- Designed on AWS for about $2/month at typical small-business volume.

The whole system on one page

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

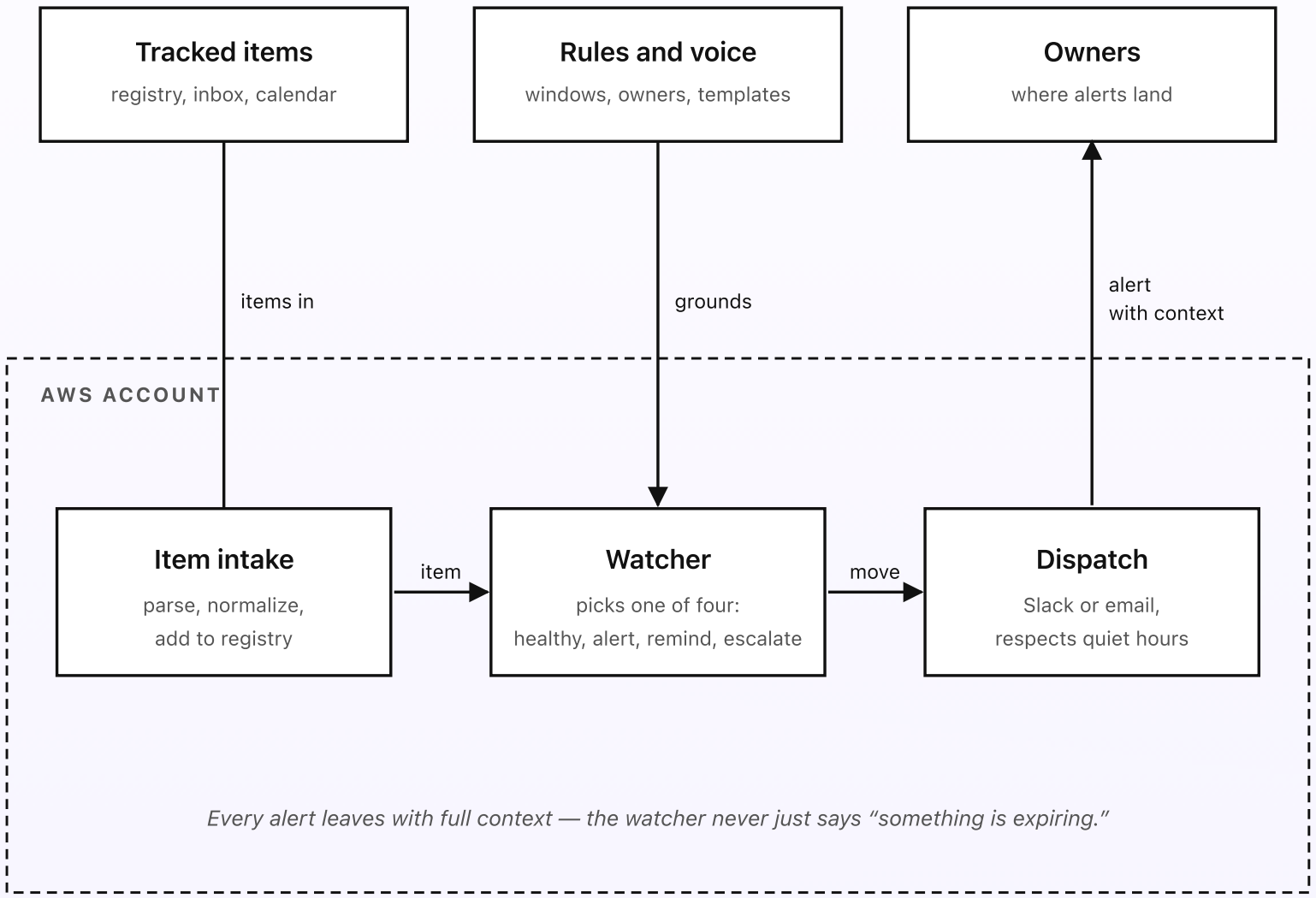

Tracked items
registry, inbox, calendar
Rules and voice
windows, owners, templates
Owners
where alerts land
items in
grounds
alert
with context
AWS ACCOUNT
Item intake
parse, normalize,
add to registry
item
Watcher
picks one of four:
healthy, alert, remind, escalate
move
Dispatch
Slack or email,
respects quiet hours
Every alert leaves with full context — the watcher never just says “something is expiring.”

Fig 1. Three sources outside, three pieces inside AWS. Items flow in from a Drive registry, an inbox forwarding lane, and a calendar import lane. The Watcher runs daily and picks one of four moves. Dispatch sends the right alert to the right person at the right time.

What you set up once (the outside)

- **Tracked items.** A Google Sheet in a Drive folder, one row per item: name, category (contract, certificate, insurance, license, lease, software, registration), owner email, vendor, expiry date, renewal cost, contract value, and a link to the source document. You can fill it in once and forget it; new items can also enter via two other lanes covered in Part 2 — an inbox-forwarding lane (forward a contract PDF to a dedicated address and the watcher proposes a row for one-tap approval) and a calendar import lane (events tagged in Google Calendar with `#expires` get pulled in automatically).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the alert windows for each category — how many days ahead the watcher should ping, and how many times. Legal contracts typically get a 90/60/30/14/3-day chain; insurance gets 60/30/7; software gets 30/14/3. The doc also lists the owner per category (or per individual item, if it overrides), the escalation target if the owner doesn't acknowledge, the quiet hours, and any holiday calendars to skip. The *voice* doc holds one alert message template per category — what the Slack DM or email actually says.
- **Owners.** The people responsible for each category. Each owner has a Slack member ID (so the alert is a DM, not a public ping) or, if Slack isn't set up for them, an email address. Pings land with the item name, days remaining, renewal

cost, contract value, a link to the source document, and an “Acknowledge” button that stops further pings on that item.

What runs on every tick (the inside)

- **The item intake.** Three sources feed the registry. The Drive sheet itself is the canonical store. New items can also be added via the inbox forwarding lane (forward a PDF to `expires@your-company.com`, the watcher uses Textract to read the PDF and Bedrock Haiku 4.5 to extract name, vendor, expiry, contract value, then drops a one-tap approval card in the rep’s Slack to confirm before the row is added) and the calendar import lane (events tagged `#expires` get pulled hourly by a small sync Lambda).
- **The watcher.** Runs once a day at 8am local. Reads the registry. For each item, computes days-to-expiry. Compares against the per-category window chain in the rules doc. Picks one of four moves. *Healthy*: more than the first window away — do nothing. *First alert*: just crossed the first window threshold — ping the owner with full context. *Reminder*: crossed a subsequent window with no acknowledgment — re-ping, mention when the previous ping went out. *Escalate*: hit the final window with no acknowledgment — ping the escalation target named in the rules doc; log it. The watcher itself doesn’t call a model on the daily tick — the move logic is plain Python.
- **Dispatch.** Reads the voice doc, formats the alert message for the chosen move and category, and sends it. Slack DMs go through an incoming-webhook URL. Email goes through SES outbound. Both honor quiet hours (no pings between 6pm and 8am local by default) and the holiday calendar (no pings on configured days). Every dispatch writes a row in DynamoDB so the next day’s tick can tell whether the owner acknowledged. A weekly digest summarizes

everything that pinged that week, plus what's coming up. A monthly summary writes a board-ready paragraph: count by category, total contract value at risk, longest-overdue items.

In plain words

Your cyber-insurance policy expires June 14. The renewal premium is $4,200 and the broker needs two weeks to bind cover. The owner is your office manager Maria. The watcher pings her in Slack on April 15 (60 days out): "Cyber-insurance renewal — $4,200 at last quote, broker needs ~2 weeks — expires June 14. *[link to policy PDF]*" with an Acknowledge button. Maria's busy that week, doesn't open it. May 15 (30 days out) she gets a reminder: "Cyber-insurance — 30 days left, last ping April 15." She acknowledges this time and books a call with the broker. The watcher stops pinging her. On May 28 the broker comes back with new terms; Maria updates the policy in the Drive registry with a new expiry of June 14, 2027 and a new premium. The watcher rolls forward and the cycle starts again next year.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the one missed cyber-insurance lapse that takes the wind out of a sales conversation, or the SOC 2 audit window everyone forgot about, or the auto-renew clause nobody read.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every alert ships with full context — item, days remaining, renewal cost, link to the source. The owner never has to dig.
- Four moves, always. Healthy, first alert, reminder, escalate. There is no fifth.
- Quiet hours and holidays are respected. Pings are a finite resource; bad timing burns them.
- Acknowledge stops further pings on that item until the next chain. A renewal updates the registry and resets the chain.
- The registry lives in Drive. Adding an item, changing an owner, or shifting an expiry doesn't need a deploy.
- Every dispatch is logged. Audit a renewal next year and you can see every ping that went out.

Why this shape

Most teams track renewals in one of three places: a spreadsheet that nobody opens, a calendar invite that gets dismissed, or somebody's head. The spreadsheet works until it doesn't — one missed update and the whole thing goes stale. The calendar invite is the worst kind of false comfort: it pings on the day, with no context, when there's no longer time to negotiate. And the head, of course, fails the moment the person who held it goes on holiday or leaves the company.

The setup above moves the source of truth into a doc the team already edits, but adds a small system that *looks at* that doc every day and acts only when something needs acting on. Pings come early enough to do something. They include enough context that the owner doesn't have to go find the PDF. They escalate cleanly when the owner is out. And they stop the moment somebody says "got it." The watcher is invisible most days; visible only on the days it actually matters.

The next four posts walk through each piece in turn: how an item gets tracked, how the watcher knows when to alert, how an alert finds the right person, and how an item gets renewed and the cycle restarts. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 5, 2026 PART 2 OF 7 · DOCUMENT EXPIRY WATCHER SERIES ~4 MIN READ

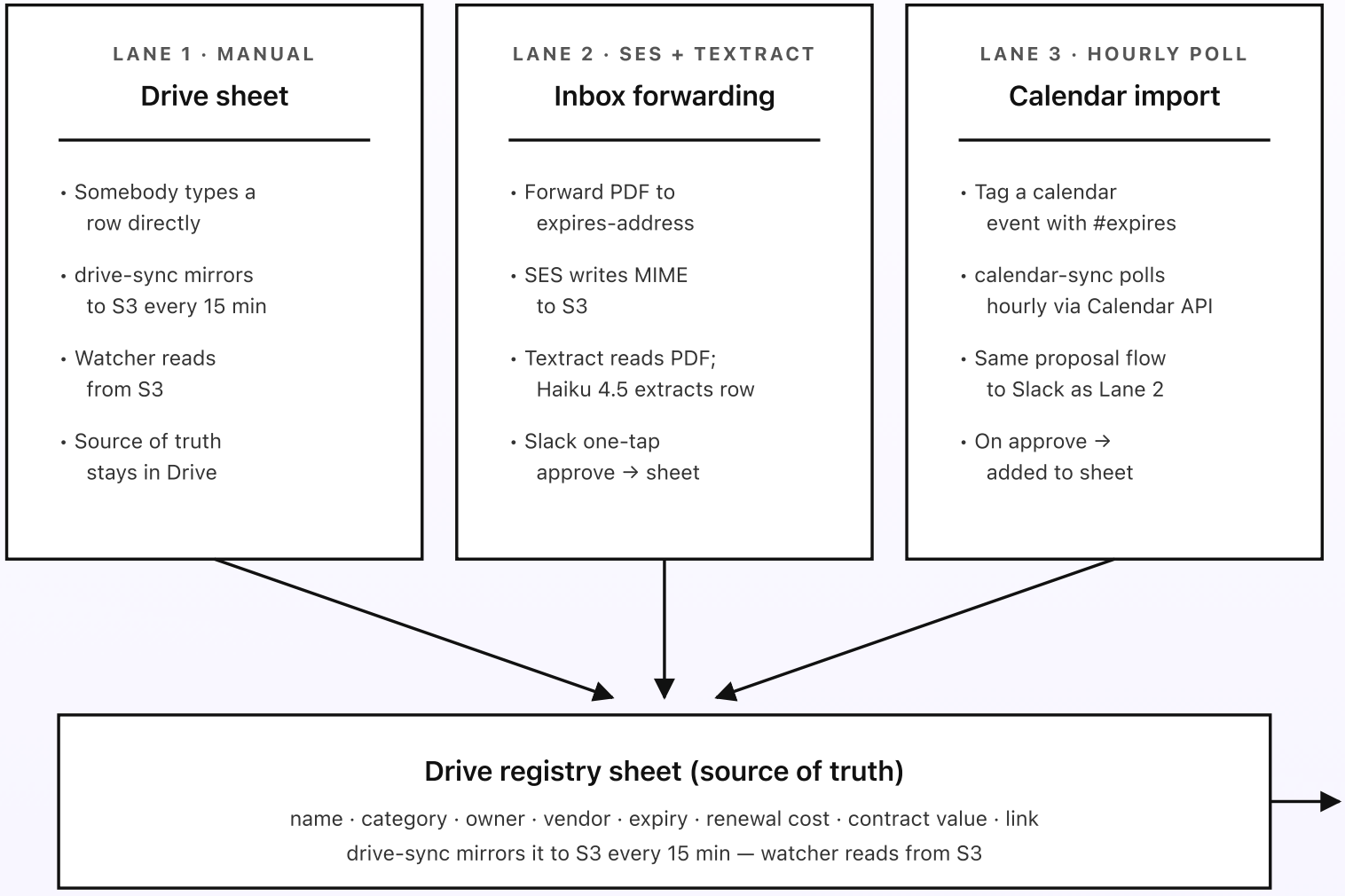
How an item gets tracked

The watcher only watches what's in the registry. So the first job is making sure the registry actually reflects what your business has. There are three ways an item gets in: somebody types a row in the Drive sheet, somebody forwards a PDF contract to a dedicated address, or somebody puts an event on their Google Calendar with a small tag. The first one is obvious. The other two exist because in real life nobody types a row in a sheet for the contract they signed three minutes ago.

KEY TAKEAWAYS

- Three intake lanes feed one registry: the Drive sheet, an inbox-forwarding lane, and a calendar import.
- Inbound PDFs are parsed by Textract; Bedrock Haiku 4.5 reads the text and 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 `#expires` 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 watcher can read it without hitting Drive on every tick.

Lane 1: the Drive sheet itself

The simplest lane. Open the registry sheet in Drive, add a row, save. The columns are short: name, category, owner email, vendor, expiry date, renewal cost, contract value, and a link to the source document. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://ew-registry-source/registry.csv` if the sheet has changed since the last sync. The watcher 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 cases where you already have a contract, you know when it expires, and you can spend thirty seconds typing it in. Most existing items go in this way during the initial setup.

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

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

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

Then a Bedrock Haiku 4.5 call reads the text and emits a structured row: name, vendor, category, expiry date, renewal cost (if present in the document), contract value (if present), and an owner-suggestion based on the "To" line of the original forward. The model prompt is short: "Extract a row for the registry. Return JSON only. Mark each field with a confidence score. Do not invent a date that isn't in the text." 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 PDF moved to a discarded prefix in S3 for audit.

The reason every parsed row goes to a human first is simple: a contract expiry the model misread is worse than a contract that never made it into the registry at all. The misread one will quietly tell you everything is fine until the morning the policy lapses.

Lane 3: calendar import

Some teams already track renewals on a calendar. The lease is on Maria's personal calendar with a yellow flag. The SOC 2 audit window is on the engineering calendar. The cyber-insurance renewal is on the office manager'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 `#expires` 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 itself 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 things they already typed once.

Why the registry stays the source of truth

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

Next post: how the watcher actually reads the registry, computes days-to-expiry, and picks one of four moves.

PART 3 OF 7

MAY 5, 2026 PART 3 OF 7 · DOCUMENT EXPIRY WATCHER SERIES ~5 MIN READ

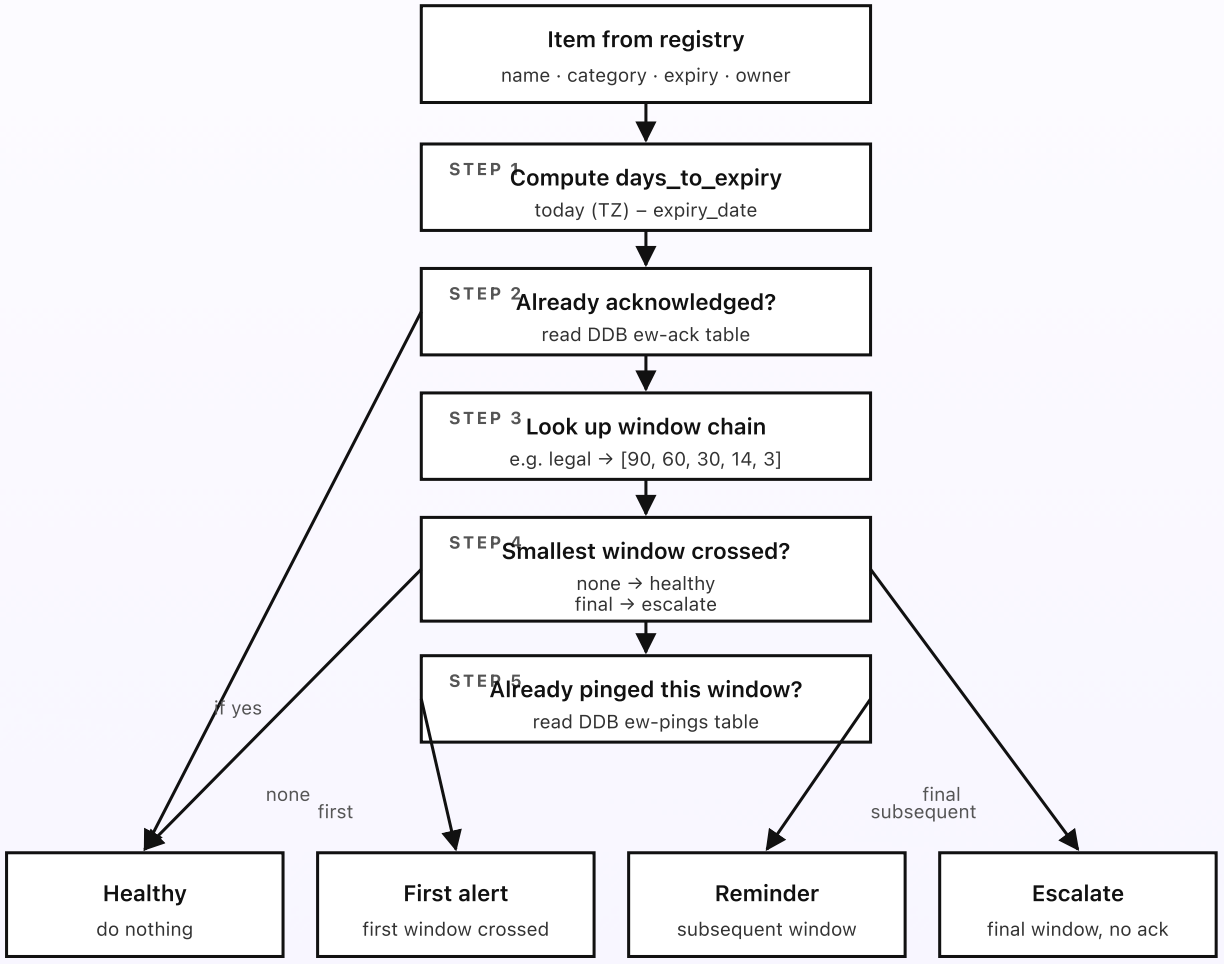
How the watcher knows when to alert

Once a day, at 8am local time, an EventBridge Scheduler rule fires the watcher Lambda. The Lambda reads the registry, looks at one row at a time, computes the days remaining, and decides whether to do nothing or to fire an alert — and if so, which kind. The whole decision is plain Python. No model. No vector retrieval. Every threshold lives in the rules doc, where a rep can edit it without a deploy.

KEY TAKEAWAYS

- The watcher runs once a day via EventBridge Scheduler at 8am local time.
- Per-category window chains live in the rules doc — legal contracts get 90/60/30/14/3, insurance gets 60/30/7, software gets 30/14/3.
- Four moves per item, every tick: healthy, first alert, reminder, escalate.
- DynamoDB tracks last-ping and acknowledgment per item so reminders aren't duplicate spam.
- The watcher itself never calls a model. The decision is entirely deterministic.

The decision flow, per item



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

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

Window chains: 90/60/30/14/3 isn't magic, it's in the doc

The rules doc has one short section per category. Each section names the chain in plain prose: "Legal contracts: ping at 90, 60, 30, 14, and 3 days. Insurance: 60, 30, 7. Software subscriptions: 30, 14, 3. Domain renewals: 60, 30, 14, 7, 3." The numbers are days remaining when the alert fires. The first number is the first alert. The last number is the escalation point — if the owner hasn't acknowledged by then, the escalation target gets pinged too.

The chains exist for a reason. A 90-day legal-contract ping gives time to negotiate with the vendor or solicit a competing quote. A 30-day insurance ping is when the broker actually has time to bind cover. A 3-day software ping is the last-chance "you forgot, please go pay this" reminder before the SaaS app locks the team out. Different categories have different mechanics; the chains reflect that.

Per-item overrides exist too. The registry sheet has an optional column called `chain_override`. Type a comma-separated list of days there and the watcher uses your numbers instead of the category default for that one row. This is the right escape hatch for the contract you signed knowing it has a 120-day notice period.

Four moves, always

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

- **Healthy.** The expiry is more than the first window away, or the item has been acknowledged for the current chain. Do nothing. Most items, most days, are healthy.
- **First alert.** The expiry just crossed the first window threshold and there's no acknowledgment yet. Send a fresh alert with full context. Write a row to the `ew-pings` DynamoDB table marking that the first window has fired.
- **Reminder.** A subsequent window crossed without acknowledgment. Send a follow-up that names the previous ping's date so the owner doesn't feel like they're seeing the alert for the first time. Write the new ping to `ew-pings`.
- **Escalate.** The final window in the chain crossed without acknowledgment. Send to the escalation target named in the rules doc — usually the owner's manager — in addition to the owner. Mark the item as escalated in DynamoDB; the next tick will keep escalating daily until somebody acknowledges or renews. Bad timing burns pings; an escalated item is one of the few cases where daily noise is the right answer.

State that makes the decision deterministic

The watcher reads two DynamoDB tables every tick. `ew-pings` records every alert that's gone out: `(item_id, chain_index, ping_date, dispatched_via)`. `ew-ack` records every acknowledgment: `(item_id, ack_date, by_user)`. With those two tables, the move-decision logic is a few dozen lines of Python and zero magic. A given item with a given expiry, a given window chain, and a given

ack/ping history always produces the same move. Re-running the tick produces no extra pings (because the state in DDB shows what already fired).

Renewing an item is an explicit reset of both tables for that item: rows for the old chain are kept for audit, but a new chain starts fresh against the new expiry date. Part 5 covers the renewal flow in detail.

Why the daily tick uses no model

The watcher could call a model on the tick to write a smarter alert message, or to decide whether to ping at all. It doesn't. Two reasons. First, the daily tick should be the one part of the system that is utterly predictable — if the rules doc says ping at 60 days and there's no ack, the ping fires. A model in that loop introduces variance the team can't reason about. Second, model calls cost money, and most days most items are healthy, so the call would be wasted nine days out of ten.

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

Next post: how an alert finds the right person, how quiet hours and holidays are honored, and what an acknowledgment actually does to the chain.

PART 4 OF 7

MAY 5, 2026 PART 4 OF 7 · DOCUMENT EXPIRY WATCHER SERIES ~5 MIN READ

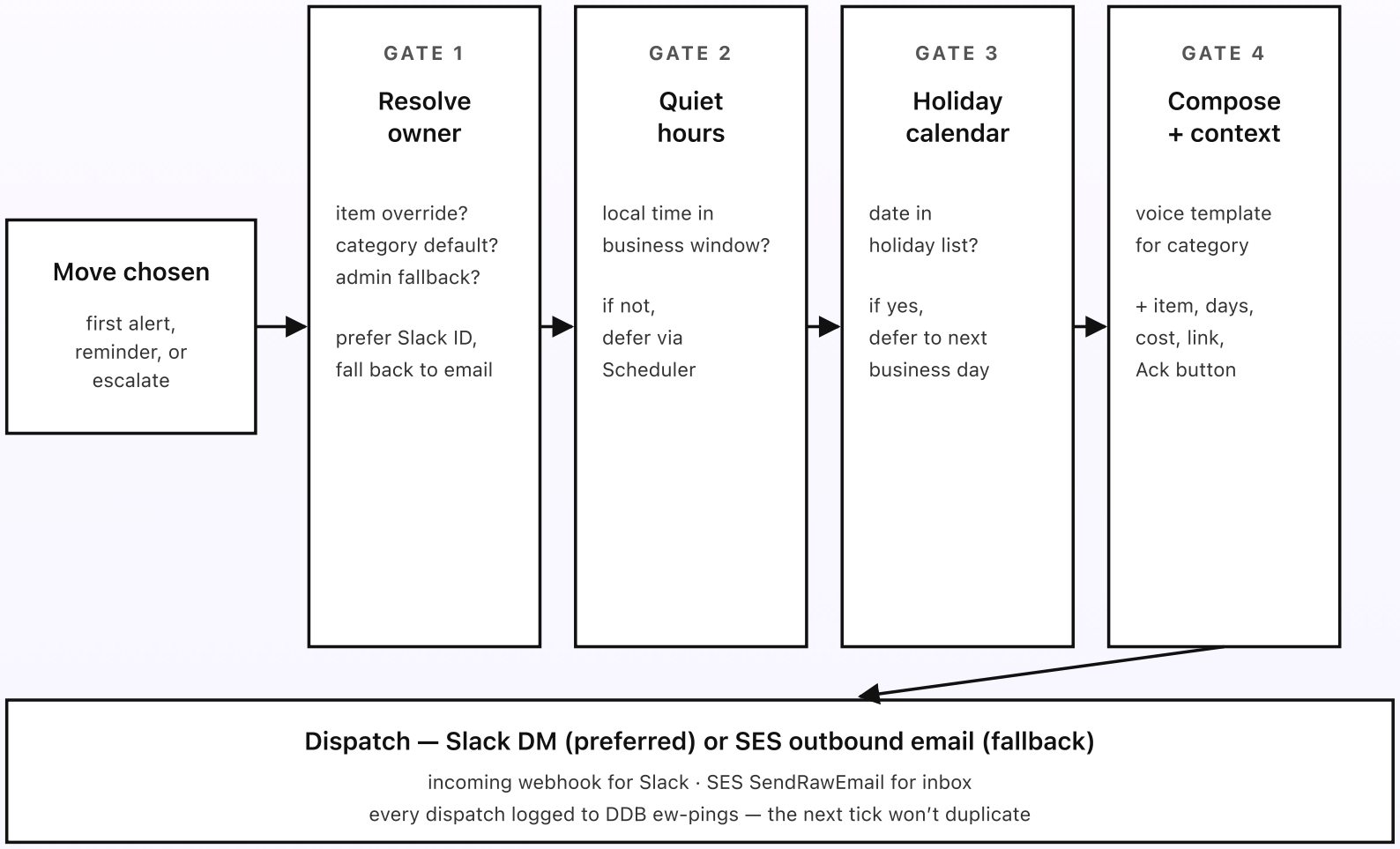
How an alert finds the right person

The watcher picked a move — first alert, reminder, or escalate. Now the dispatch Lambda has to figure out who to send it to, on what channel, at what time of day, and with what context attached. Get any of those wrong and the alert is worse than no alert: a 2am Slack ping, a generic “something is expiring,” a notification to somebody who left the company three months ago. Four small guardrails sit between the move and the actual ping.

KEY TAKEAWAYS

- Owner resolution: per-item override beats per-category default beats fallback to the configured admin.
- Slack DMs are the default; email is the fallback if no Slack ID is configured.
- Quiet hours and holiday calendars defer pings to the next available business hour.
- Every alert ships with the item, days remaining, renewal cost, link to the source, and an Acknowledge button.
- Escalation pings the named target instead of (or alongside) the owner; the owner stays in the loop.

Four guardrails on every dispatch



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

Fig 4. Four guardrails between the move and the dispatched alert. Resolve the owner. Honor quiet hours. Skip holidays. Compose with full context. Then ship via Slack or email and log the dispatch so the next tick doesn't duplicate.

Gate 1: resolve the owner

Three places the dispatch Lambda looks for the owner of an item, in order. First, the registry sheet's per-item `owner_email` column — if a row has a specific person assigned, that person owns it regardless of the category default. Second, the per-category default in the rules doc ("all insurance items default to the office manager"). Third, the configured admin fallback — the person who set up the watcher and gets every unowned ping. The fallback should never fire in steady state; if it does, the weekly digest names every item that hit the fallback so the rules doc can be updated.

Once the dispatch knows which person to ping, 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 alerts feel like work-context messages, and a Slack DM with action buttons is more useful than an email link. Email is the fallback so nobody falls through the cracks.

Gate 2: quiet hours

The watcher itself runs at 8am local time, so the first time a move fires it's already in business hours. But escalations and reminders that result from a tick can fire later in the day. And one-off computed dispatches (the second-window ping that takes effect at the same time as the first) can land outside the configured window.

Gate 2 reads the rules doc's quiet-hours setting (default 6pm to 8am, configurable per business). If the current local time is in the quiet window, the dispatch creates a one-off EventBridge Scheduler rule that fires at the next business-hour minute and exits without sending. The Scheduler invokes the same dispatch Lambda with the same payload at the deferred time, where Gate 2 will let it through.

Gate 3: holiday calendar

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

The list is on purpose — the watcher won't auto-detect a country's public holidays for you. The failure modes are very different. A holiday you forgot to add fires a ping that lands on a closed laptop. A holiday in the list that's no longer observed just delays a ping by one business day, which is fine. The trade-off favors keeping the list explicit.

Gate 4: compose with full context, then ship

The voice doc has one Slack message template per category: a short message with placeholders for the item name, days remaining, renewal cost, contract value, and link to the source document. The dispatch Lambda fills the placeholders, attaches an "Acknowledge" button, and ships the message via the Slack incoming webhook. The webhook URL itself lives in Secrets Manager.

For email fallback, the same template is wrapped in a small HTML email with the same fields and a link that, when clicked, hits a Function URL that records the acknowledgment — the email equivalent of the Slack button.

An escalate move adds a second recipient: the escalation target named in the rules doc for that category. The owner is still pinged (the escalation isn't a substitute for the original owner's ping — both go out), but the manager now sees it too. The escalate template is slightly different: it includes the previous ping dates and the cumulative days the item has been overdue, so the manager has the audit trail at hand.

Every dispatch — Slack or email, owner or escalate — writes a row to `ew-pings` in DynamoDB. The next day's tick reads that row and knows not to ping the same window again.

Why the guardrails exist

None of these gates are exotic. They're the kind of small care a thoughtful human would take if they were sending the alerts themselves — check who actually owns this, don't ping at 11pm, skip the day everyone's off, include enough context that the recipient doesn't have to ask a follow-up question. Putting them in code as four small sequential gates makes them part of the design, not a feature you're trusting the writer of any one alert to remember.

Next post: how an item gets renewed once an owner has acted on the alert — how the watcher captures the new expiry, archives the old chain, and starts a fresh one for next year.

PART 5 OF 7

MAY 5, 2026 PART 5 OF 7 · DOCUMENT EXPIRY WATCHER SERIES ~5 MIN READ

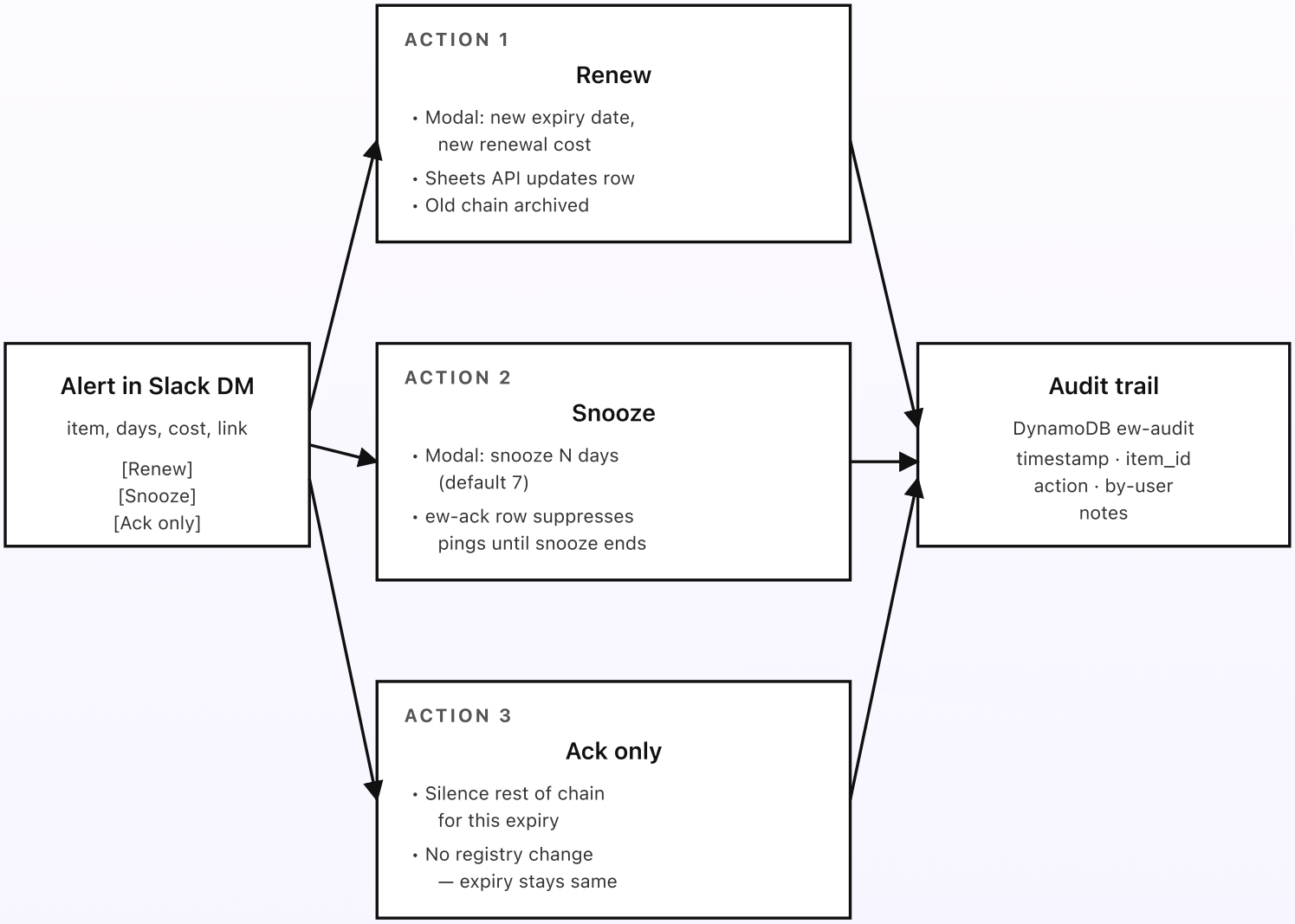
How an item gets renewed

An alert lands in Maria's Slack DM at 8:03am. The cyber-insurance policy expires in 60 days. There's an Acknowledge button. What happens when she taps it? The honest answer is "it depends on what she actually did." This post walks through the three things the watcher can do on an acknowledgment — renew, snooze, ack-only — and how the registry, the chain state, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Three actions per acknowledgment: *ack-and-renew* (new expiry, fresh chain), *ack-and-snooze* (delay), *ack-only* (silence the chain).
- Each action updates the registry sheet via the Sheets API and writes an audit row.
- A renewal archives the old chain to a separate sheet for history.
- Snooze is bounded — you can only snooze a few times before the watcher escalates anyway.
- The Acknowledge button is a Slack interactive message backed by a Function URL.

Three actions on Acknowledge



Ack only doesn't change the expiry — it stops the noise. The next renewal cycle starts on schedule.

Fig 5. Three actions per acknowledgment, three different effects. Renew updates the expiry and starts a fresh chain. Snooze delays without dismissing. Ack-only silences the chain without changing the expiry. Every action writes to the audit trail.

Action 1: ack-and-renew (the most common)

Maria worked with the broker, signed the new policy, and now needs to update the registry. She taps *Renew*. A small Slack modal opens with two fields: *New expiry date* (defaulting to one year from the previous expiry, which is right most of the time) and *New renewal cost* (defaulting to the previous cost). She edits whichever fields actually changed and hits Save.

The Save button submits to a Function URL Lambda. Three things happen, in order. First, the Sheets API updates the row in the registry sheet: new `expiry_date`, new `renewal_cost`, and a small note in the `last_renewed` column with today's date and the user who acted. Second, the existing chain's rows in `ew-pings` are copied to `ew-pings-archive` with a chain id, and the live chain is cleared. Third, an `action: renewed` row is written to `ew-audit` with the user, timestamp, old expiry, new expiry, and any cost change.

Tomorrow's tick reads the registry, sees the new expiry date is more than 90 days out, and lands at *healthy*. The owner won't hear from the watcher about this item again until the new chain's first window crosses.

Action 2: ack-and-snooze (the deferral)

Some renewals take time the alert chain doesn't plan for. The vendor is slow to respond. The legal review is stuck behind a holiday. The CEO is the only person who can sign and they're traveling. Maria isn't ready to renew, but she's also handling it — she just needs the watcher to be quiet for a week.

Snooze opens a small modal asking for the number of days, with a 7-day default and a max of 14. On save, a row is written to `ew-ack` with `(item_id, snooze_until)`. The next day's tick reads that row in the "already acknowledged?" check from Part 3 and treats the item as healthy until the snooze ends. When the snooze ends, the watcher re-evaluates the chain from where it was — if the item is now in escalation territory, the next ping is an escalation.

Snooze is bounded. The rules doc has a configurable `max_snoozes_per_chain` setting (default three). After that many snoozes on the same expiry, further snooze attempts are rejected with a "You've hit the snooze cap on this item; please renew or escalate" reply, and the next tick pings normally regardless. This is a soft constraint that exists because the most dangerous failure mode is repeatedly snoozing an item to nowhere.

Action 3: ack-only (the "I've got it")

Sometimes the owner doesn't want to renew right now and doesn't want to be reminded again either. Maybe the policy is genuinely not getting renewed (changing vendors). Maybe the item is being deprecated. Maybe the owner is going to handle it manually and doesn't need the system in the loop.

Ack only writes a row to `ew-ack` with `(item_id, ack_until: expiry_date)` — effectively, "don't ping me about this item again until its expiry passes." The expiry

itself isn't changed. The chain is silenced for the rest of this cycle. If the item ends up actually getting renewed via Lane 1 (somebody just edits the registry sheet directly), the new expiry resets the chain naturally on the next tick.

If the expiry passes without a renewal, the watcher does one more thing: it writes the item to a separate *lapsed* sheet in the same Drive folder, with the date it lapsed and the last-known owner. The lapsed sheet is what the monthly summary in Part 6 reports on. A lapsed item that was ack-only'd gets a small note — "ack-only by Maria on 2026-04-15" — so the audit trail isn't a mystery later.

Every action is logged, every action is reversible

The `ew-audit` table records every renew, snooze, and ack-only with the user who took the action, the timestamp, and a snapshot of the row before and after. If a wrong renewal date gets entered (off by a year, off by a day), a rep can run an "undo last action" through a small admin command that reads the previous-state snapshot and restores the row. The undo is itself an audit row, so the trail of edits stays clean.

This kind of reversibility matters most for the renewals you'll only think about once a year. The next time the cyber-insurance comes up, it'll be Maria again or it'll be the person who took her job after she got promoted. Either way, the audit trail is the only memory the next person has.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why it's less than the lead intake bot's bill.

PART 6 OF 7

MAY 5, 2026 PART 6 OF 7 · DOCUMENT EXPIRY WATCHER SERIES ~3 MIN READ

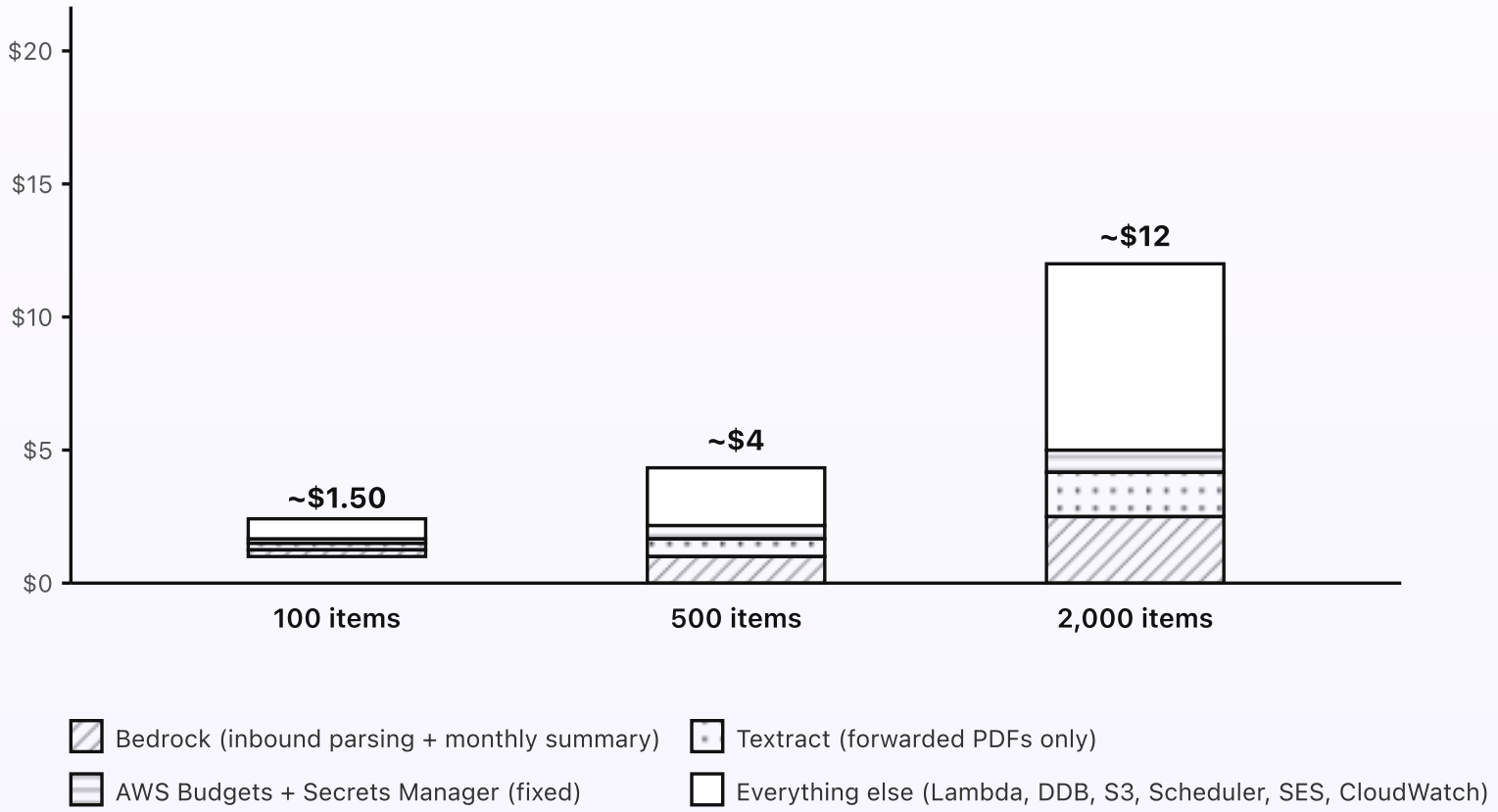
What the expiry watcher costs

The watcher is one of the cheapest systems in this whole series. The daily tick reads a CSV from S3, does some date arithmetic, writes a few rows to DynamoDB, and posts a handful of messages to Slack. It calls no models on the tick. Bedrock fires only when somebody forwards a contract PDF and once a month for the board summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $1.50/month at typical SMB volume (around 100 tracked items).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The daily tick costs pennies — no model calls.
- Bedrock fires only on inbound PDF parsing (a few times a month) and the monthly summary.
- At 500 tracked items the bill is around $4. At 2,000 items it's around $12.

Cost at three volumes



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

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

Where the dollars actually go

Lambda runtime (the bulk). The watcher runs once a day. Each tick reads the registry CSV from S3, iterates the rows, computes `days_to_expiry` for each, and decides on a move. At 100 items, that's a few hundred milliseconds. At 2,000 items it's a couple of seconds. Either way it's pennies a month. Add the dispatch Lambda firing for each ping (around two to ten pings a month at 100 items, twenty to fifty at 2,000), the Function URL Lambda for acknowledgments, the calendar-sync Lambda running hourly, and the drive-sync Lambda every fifteen minutes — the Lambda total still lands under a dollar at all three volumes.

DynamoDB on-demand. Three small tables: `ew-pings`, `ew-ack`, `ew-audit`. Reads are dominant during the daily tick (one read per item per tick, plus chain history). Writes are dispatch events 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 tick rule plus deferred dispatch rules from quiet-hours and holiday gates. A few invocations a day. Pennies.

SES. Inbound for the forwarding lane: $0.10 per thousand received messages (so a couple of cents a year for an SMB). Outbound for email-fallback alerts: $0.10 per thousand sent. Both are negligible at this scale.

Bedrock (only when something fires it). The daily tick uses no Bedrock. The inbound parsing lane fires Haiku 4.5 once per forwarded PDF: a few thousand input tokens (the Textract output) and a few hundred output tokens (the proposed

row JSON), so a fraction of a cent per parse. At a few forwarded contracts a month, Bedrock costs cents. The monthly summary is one larger call: write a paragraph that summarizes the month's pings, renewals, and lapses; a couple of cents.

Textract (only on forwarded PDFs). Per-page pricing; a typical contract is two to ten pages. A few cents per parse. At a few PDFs a month, Textract is a few cents to a dollar. At 2,000 tracked items with twenty PDFs forwarded per month, it lands around a couple of dollars.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the acknowledgment endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The watcher sleeps 23.99 hours a 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 tick.** The daily decision is plain Python. Bedrock fires only on the inbound parsing lane and the monthly summary.

How the cost scales

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

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The watcher'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

MAY 5, 2026 PART 7 OF 7 · DOCUMENT EXPIRY WATCHER SERIES ~8 MIN READ

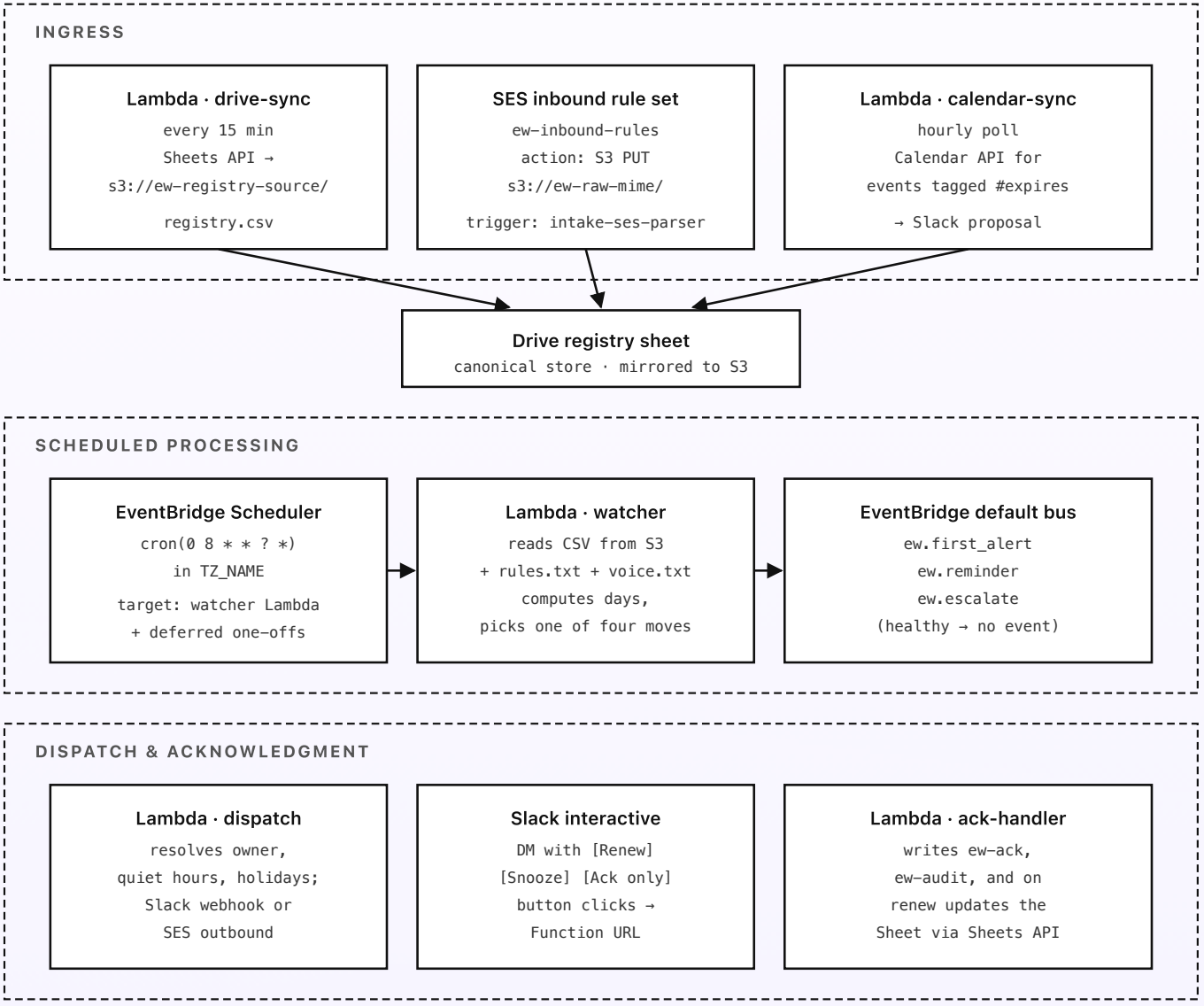
Engineering reference: the expiry watcher architecture

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: **us-east-1**. 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 somebody missing a renewal alert, not a regional outage. One AWS account dedicated to the watcher (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every alert leaves with full context — and every interaction is logged to ew-audit.

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

Lambda functions

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

- **`drive-sync`** — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `ew/drive/sa`) to export the registry sheet as CSV and write to `s3://ew-registry-source/registry.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://ew-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 `#expires` 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://ew-raw-mime/`. Parses MIME, extracts the PDF attachment, 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 a registry row. Posts the proposal to Slack via the incoming webhook with Approve/Edit/Discard buttons. For DOCX attachments (Textract doesn't accept them), falls back to `python-docx`; XLSX uses `openpyxl`. Both packages are stable and widely used in 2026, though their maintenance velocity is light — for a contract-parsing path that only runs a few times a month, that's acceptable. If extraction precision becomes a concern, the active community fork `python-docx-oss` is a drop-in alternative. Memory: 512 MB. Timeout: 60 s.
- **`watcher`** — EventBridge Scheduler target, daily at 8am local time (the schedule expression runs in `TZ_NAME` set to the SMB's timezone, e.g. `America/New_York`). Reads `s3://ew-registry-source/registry.csv` and the rules and voice docs. For each row, computes `days_to_expiry`, reads chain state from `ew-pings` and `ew-ack`, decides on a move. Emits one event per row that needs action: `ew.first_alert`, `ew.reminder`, or `ew.escalate`, with the item context as the event payload. Healthy items emit nothing. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`dispatch`** — EventBridge rule on the three move events. Resolves owner, checks quiet hours and holiday calendar, formats the alert from the voice template, and ships via Slack incoming webhook (`ew/slack/webhook` in Secrets Manager) or SES `SendRawEmail`. On quiet-hours or holiday defer, creates a one-off EventBridge Scheduler rule that re-invokes `dispatch` at the

next available business minute. Writes a row to `ew-pings` after a successful send. Memory: 256 MB. Timeout: 30 s.

- **`ack-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive button clicks (Renew/Snooze/Ack-only) and by email-link clicks. Writes to `ew-ack` and `ew-audit`; on renew, updates the Drive sheet via the Sheets API and archives the old chain in `ew-pings-archive`. Memory: 256 MB. Timeout: 15 s.
- **`digest`** — EventBridge Scheduler target, weekly Sunday 6pm. Reads `ew-pings` for the past week and the registry; sends a digest message to a configured Slack channel summarizing pings sent and items 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 `ew-pings`, `ew-ack`, and `ew-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph board narrative; emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB · `ew-pings`** — one row per dispatch. PK `(item_id, chain_index)`; attributes: `ping_date`, `dispatched_via` (slack/email), `recipient`, `move` (first_alert/reminder/escalate). On-demand. No TTL.
- **DynamoDB · `ew-ack`** — one row per acknowledgment. PK `item_id`; sort key `ack_date`; attributes: `action` (renew/snooze/ack-only), `by_user`, `snooze_until` (if action = snooze), `old_expiry`, `new_expiry` (if action = renew). On-demand.

- **DynamoDB** · `ew-audit` — one row per write action of any kind. PK `(item_id, ts)`; attributes: `action`, `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB** · `ew-pings-archive` — archived chains after a renewal. Same shape as `ew-pings`; PK `(item_id, chain_id, chain_index)`. On-demand.
- **S3** · `ew-registry-source` — mirrored CSV from the Drive registry sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3** · `ew-rules-source` — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3** · `ew-raw-mime` — raw inbound MIME from forwarded contracts. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3** · `ew-source-pdfs` — the parsed source contracts after the inbound parser handles them, kept for reference if the registry row links to one.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `intake-ses-parser` for the inbound contract parsing, and `summary` for the monthly board narrative.
- **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 watcher itself doesn't call Bedrock; the parsing lane fires a few times a month at most.

EventBridge Scheduler config

- `ew-daily-tick` — `cron(0 8 * * ? *)` in the SMB's timezone. Target: `watcher` Lambda.
- `ew-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `ew-calendar-sync` — `rate(1 hour)`. Target: `calendar-sync` Lambda.
- `ew-weekly-digest` — `cron(0 18 ? * SUN *)` in TZ. Target: `digest` Lambda.
- `ew-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `dispatch` when a quiet-hours or holiday defer is needed. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `expires.your-company.com`) to `inbound-smtp.us-east-1.amazonaws.com`.
- SES inbound rule set `ew-inbound-rules`: one rule with recipient `expires@your-company.com` → spam scan → S3 PUT to `s3://ew-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the email-fallback alerts: verify a sender identity at `watcher@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:

- **watcher role:** `s3:GetObject` on the registry, rules, and voice keys; `dynamodb:Query` + `GetItem` on `ew-pings`, `ew-ack`; `events:PutEvents` on the default bus. *No* `bedrock:*`.
- **dispatch role:** `events:ListSchedules` + `CreateSchedule` for the deferred-dispatch one-offs; `secretsmanager:GetSecretValue` on the Slack webhook secret; `ses:SendRawEmail` from the verified sender identity; `dynamodb:PutItem` on `ew-pings`; outbound network access to `hooks.slack.com`.
- **ack-handler role:** `dynamodb:PutItem` on `ew-ack` and `ew-audit`; `secretsmanager:GetSecretValue` on the Sheets-API service-account secret; outbound network access to `sheets.googleapis.com`; `dynamodb:Query` for chain state lookup; on renew, `dynamodb:BatchWriteItem` for archiving the old chain to `ew-pings-archive`.
- **intake-ses-parser role:** `s3:GetObject` on `ew-raw-mime`; `textract:StartDocumentTextDetection` + `StartDocumentAnalysis`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack webhook.
- **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 alert messages are posted via the `chat.postMessage` Web API instead, with Block Kit blocks containing the action buttons. Button clicks are sent by Slack to the configured Interactivity request URL, which is the `ack-handler` Function URL. `ack-handler` verifies the Slack signing secret on the inbound request, parses the action_id (`renew`, `snooze`, `ack_only`), opens a modal if needed (Renew/Snooze open modals; Ack-only is one-tap), and processes the response when the modal is submitted.

The Slack app needs `chat:write`, `im:write`, and the Interactivity URL configured. The bot token lives in Secrets Manager under `ew/slack/bot-token`. The signing secret is `ew/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:** watcher Lambda failures > 0 in a day (the daily tick is the one piece that has to run); dispatch failure rate > 1% in 24h; ack-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 `ew-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 `ew/drive/sa` (one service account with scopes for all three APIs). Slack bot token, signing secret, and webhook URL all under `ew/slack/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, holiday list reference, quiet-hours window, and admin fallback owner all live in Parameter Store under `/ew/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 `ew-registry-source` and `ew-rules-source` so a bad Drive edit can be rolled back in one click, and version the EventBridge Scheduler timezone setting so you don't accidentally start running the daily tick in UTC after a CI rotation. CDK with a Python stack file works well; SAM also fits. 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.