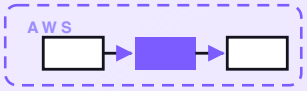
7-PART SERIES · FREE COMPANION



Price monitor

A serverless monitor that keeps an eye on competitor prices so you don't have to. It checks a list of competitor product pages on a schedule, notices when a price moves meaningfully, and sends the right owner a short alert — "Competitor X dropped product Y by 12%" — with the recent history. It only suggests; it never changes your prices. It stays polite to every site it checks. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Price monitor

PART 1 OF 7

MAY 22, 2026 PART 1 OF 7 · PRICE MONITOR SERIES ~5 MIN READ

A competitor price monitor on AWS for a few dollars a month

A small business loses track of what its competitors charge the moment it gets busy. The rival down the road that quietly cut its headline product by 15% last Tuesday. The online store that runs a flash sale every payday weekend. The supplier who crept their list price up 3% and nobody noticed for two months. Checking all of that by hand is the kind of chore that never makes it to the top of anyone's day. This post walks through the design of a small monitor that watches those pages for you, notices when a price actually moves, and tells the right person with enough context to decide — without ever touching your own prices.

KEY TAKEAWAYS

- Three sources for watched pages: a Drive watch list, a paste-a-link lane, and a catalog import lane.
- Every page ends in one of four moves on each check: steady, first alert, repeat move, or big swing.
- Per-page rules: a move threshold (default 5%), a check rate, and the owner who hears about it.
- It checks gently — honors robots rules, staggers requests, backs off on errors. It only suggests.
- 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.

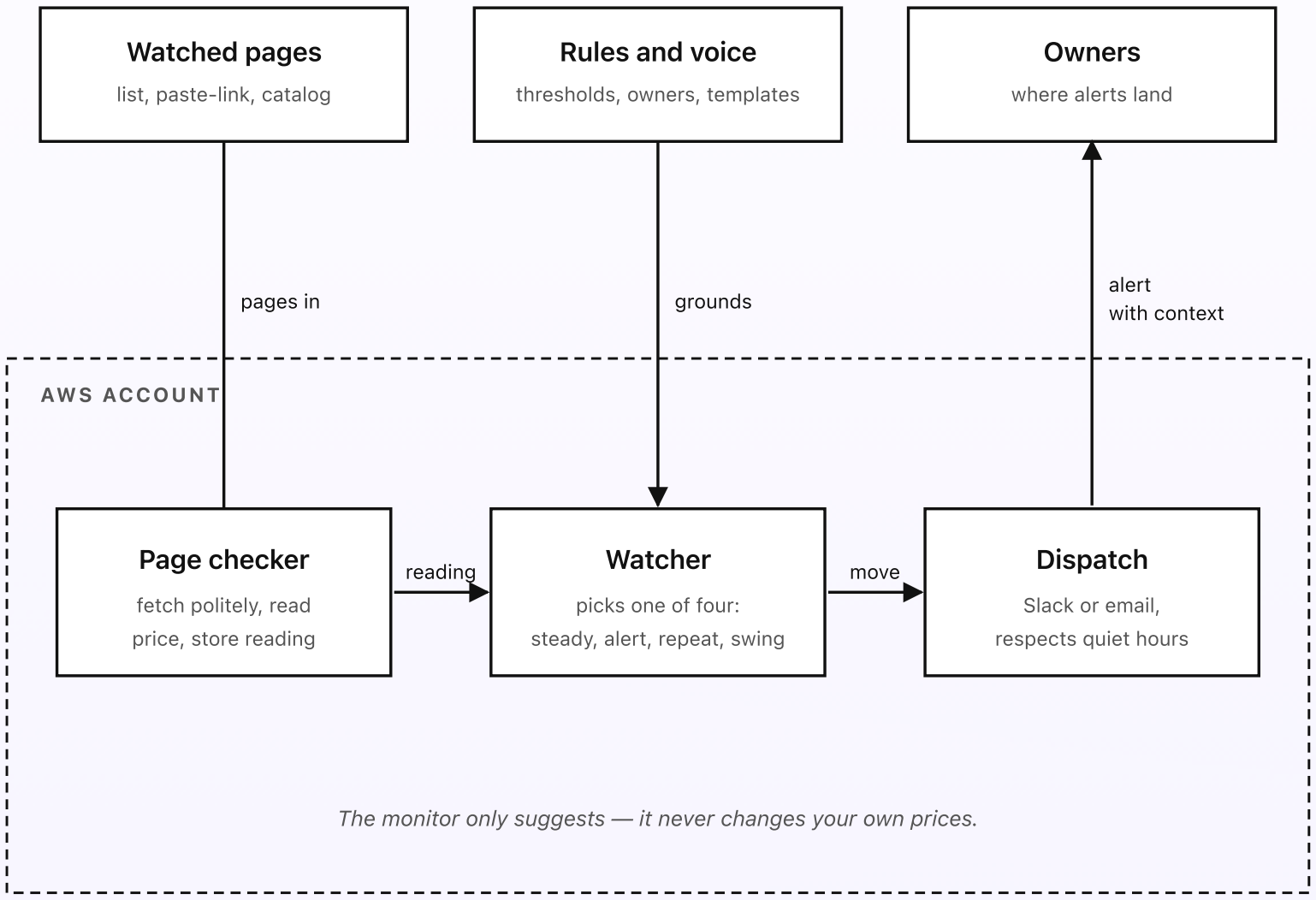

Watched pages
list, paste-link, catalog
Rules and voice
thresholds, owners, templates
Owners
where alerts land
pages in
grounds
alert
with context
AWS ACCOUNT
Page checker
fetch politely, read
price, store reading
reading
Watcher
picks one of four:
steady, alert, repeat, swing
move
Dispatch
Slack or email,
respects quiet hours
The monitor only suggests — it never changes your own prices.

Fig 1. Three sources outside, three pieces inside AWS. Pages flow in from a Drive watch list, a paste-a-link lane, and a catalog import lane. The Watcher runs on a schedule 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)

- **Watched pages.** A Google Sheet in a Drive folder, one row per page: product name, competitor, page URL, the field to read (the price, and sometimes a stock or sale flag), a check rate, your own listed price for reference, the owner email, and a move threshold (how big a change counts as worth knowing). You can fill it in once and forget it; new pages can also enter via two other lanes covered in Part 2 — a paste-a-link lane (drop a URL into a Slack box and the monitor reads it once and proposes a row) and a catalog import lane (products tagged in your own store catalog get matched to competitor pages automatically).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the move threshold per page or per product line — how big a price change has to be before the monitor bothers you (default 5%), how often to check, and the quiet hours and daily cap so a busy sale day doesn't bury you in alerts. The doc also lists the owner per product line (or per individual page, if it overrides) and the polite-fetch settings the monitor must obey. The *voice* doc holds one alert message template per kind of move — what the Slack DM or email actually says.
- **Owners.** The people responsible for each product line. 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. Alerts land with the product name, the competitor, the

old and new price, the size of the move, a small recent-history sparkline, your own listed price for reference, and a “Snooze” button that quiets that page for a while.

What runs on every check (the inside)

- **The page checker.** Reads the watch list and, on each page’s schedule, fetches the page politely — one quiet request, a clear identifying user agent, honoring the site’s robots rules and any crawl-delay. It reads the price using a saved rule for that page (a small recipe that says where on the page the number lives). The reading — price, currency, in-stock flag, and a timestamp — is written to the price-history store. If the saved rule can’t find a price (the page layout changed), that’s the one place a model gets involved, covered in Part 2.
- **The watcher.** Runs on a schedule, staggered across the day so requests never arrive in a burst. Reads the latest reading and the recent history for each page. Computes the change since the last reading and compares it against the per-page threshold. Picks one of four moves. *Steady*: the price didn’t move past the threshold — do nothing. *First alert*: the price crossed the threshold for the first time — tell the owner with full context. *Repeat move*: it moved again in the same direction since the last alert — re-alert, mention the previous reading. *Big swing*: a large jump (default twice the threshold, or a stock change) — flag it as urgent. The watcher itself doesn’t call a model — the move logic is plain Python.
- **Dispatch.** Reads the voice doc, formats the alert message for the chosen move, and sends it. Slack DMs go through the Slack API. Email goes through SES outbound. Both honor quiet hours (no alerts between 7pm and 8am local by default) and a daily cap so a sale day doesn’t bury an owner. Every dispatch writes a row so the next check can tell whether this move already went out. A

weekly digest summarizes the moves that week, plus any pages that failed to read. A monthly summary writes a short narrative: who moved, by how much, and where you sit against the market.

In plain words

You sell a popular kettle for $79. Your closest rival lists the same model. The monitor checks their page twice a day. On Tuesday at lunchtime the rival drops it to $69 — a 13% cut, well past your 5% threshold. The owner of that product line, Priya, gets a Slack DM: “Competitor Acme dropped the 1.7L kettle 13% — was $79, now $69 — you list it at $79. *[recent history] [link to their page]*" with a Snooze button. Priya looks at it, decides not to match (her margins are tighter), taps *Note* and types “holding price, our reviews are stronger.” The monitor logs the decision and keeps watching. Two weeks later the rival quietly puts it back to $79; the monitor sends a one-line “Acme kettle back to $79” so Priya knows the sale ended. Nothing in this story changed Priya’s price — the system only ever told her what happened.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the week you kept selling at full price while three rivals quietly undercut you, or the supplier increase you absorbed for two months because nobody was watching the list.

DESIGN RULES THAT SHAPED EVERY DECISION

- The monitor only suggests. There is no path that edits your store, your catalog, or any price.
- Every alert ships with full context — product, competitor, old and new price, the size of the move, recent history, your own price. The owner never has to dig.
- Four moves, always. Steady, first alert, repeat move, big swing. There is no fifth.
- Be polite. Honor robots rules, stagger requests, identify yourself, back off on errors. A page that keeps failing is paused, not hammered.
- Quiet hours, a daily cap, and a move threshold keep alerts rare and worth reading. Noise burns trust.
- The watch list lives in Drive. Adding a page, changing an owner, or shifting a threshold doesn't need a deploy.

Why this shape

Most owners track competitor prices in one of three places: a browser tab they keep meaning to refresh, a spreadsheet they update once a quarter, or a vague memory of what things cost last time they looked. The tab gets closed. The spreadsheet goes stale the day after it's built. And memory is the worst of the three: it tells you a competitor is "about the same" right up until you lose a deal you didn't know you were losing.

The setup above keeps the list of what to watch in a doc the team already edits, but adds a small system that *looks at* those pages on a gentle schedule and speaks up only when something actually moved. Alerts come with enough context to decide on the spot. They're rare by design — a threshold, quiet hours, and a daily cap mean an owner only hears about changes worth hearing about. And crucially, the system never acts on your behalf: it watches, it tells, and the pricing decision stays with a person who can weigh margin, reviews, and stock the way no rule can.

The next four posts walk through each piece in turn: how a competitor page gets watched, how a price move gets noticed, how an alert reaches the owner, and how an alert gets handled. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 22, 2026 PART 2 OF 7 · PRICE MONITOR SERIES ~4 MIN READ

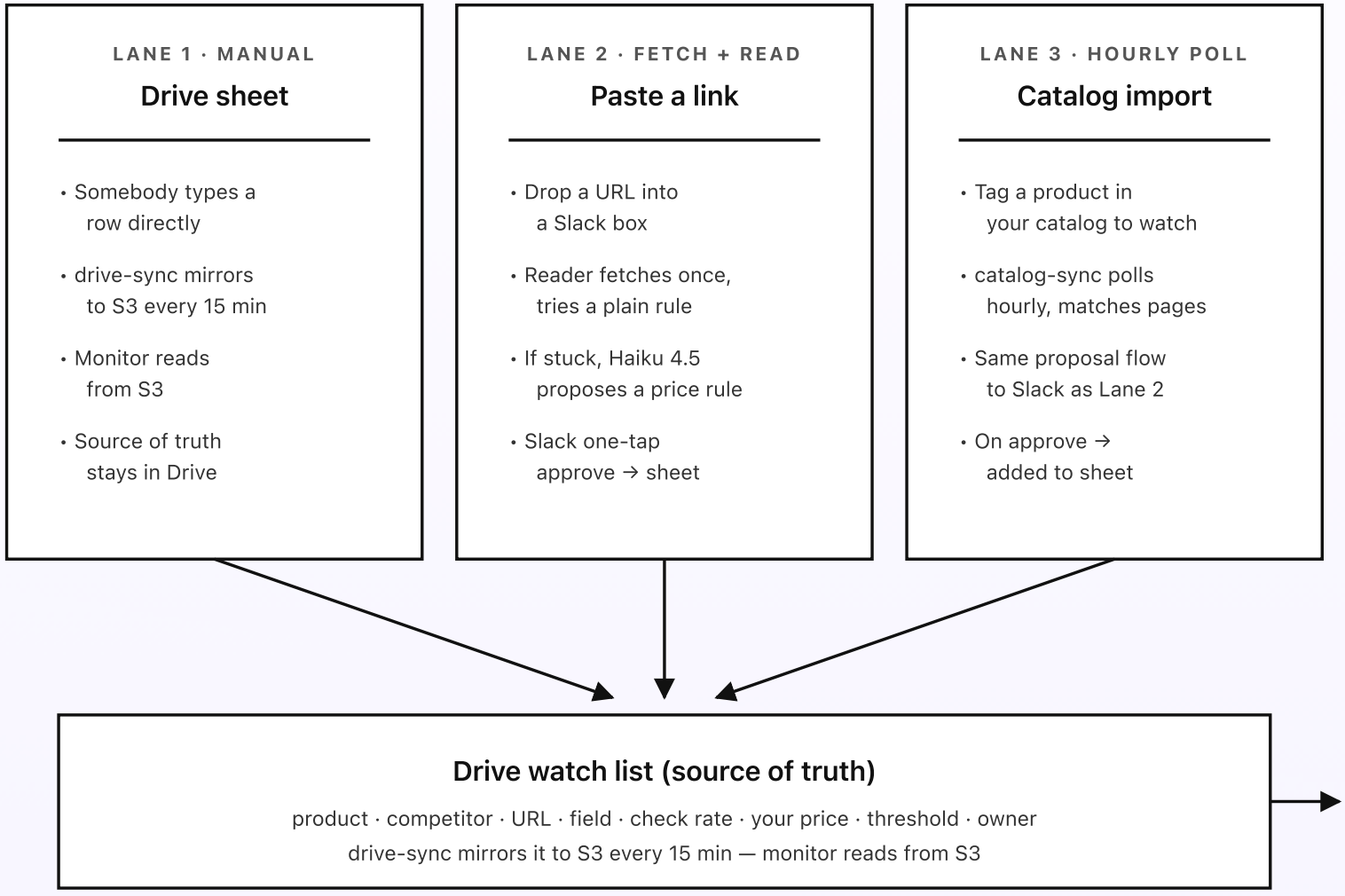
How a competitor page gets watched

The monitor only watches what's on the watch list. So the first job is making sure that list reflects the competitors you actually care about. There are three ways a page gets in: somebody types a row in the Drive sheet, somebody pastes a link into Slack, or a product in your own catalog gets matched to a competitor page. The first one is obvious. The other two exist because in real life nobody opens a spreadsheet to add the rival page they just spotted in a browser tab.

KEY TAKEAWAYS

- Three intake lanes feed one watch list: the Drive sheet, a paste-a-link lane, and a catalog import.
- Each new page is read once so the monitor can save a price rule — where on the page the number lives.
- If a plain rule can't find the price, Bedrock Haiku 4.5 proposes one from the page text.
- Every proposed row goes to a person for one-tap approval before it lands on the list.
- The watch list stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one watch list



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

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the paste-a-link lane and the catalog lane are conveniences that propose rows for human approval. The drive-sync Lambda mirrors the sheet to S3 so the monitor can read it without hitting Drive on every check.

Lane 1: the Drive sheet itself

The simplest lane. Open the watch list in Drive, add a row, save. The columns are short: product name, competitor, page URL, the field to read, a check rate, your own listed price, a move threshold, and the owner email. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://pm-watchlist-source/watchlist.csv` if the sheet has changed since the last sync. The monitor 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 know the page, you know what to read, and you can spend thirty seconds typing it in. Most existing competitors go in this way during the initial setup.

Lane 2: paste a link (the lane most teams actually use)

Set up a small Slack box — a slash command like `/watch` or a dedicated channel — where anyone on the team can drop a competitor product URL. A reader Lambda takes it from there. The Lambda fetches the page once, politely (one quiet request, a clear identifying user agent, honoring the site's robots rules), and tries a plain price rule first: look for common price markup — a `price` microdata field, a JSON-LD `offers.price`, or a labelled price element. Most well-built shop pages give the price up cleanly this way, and no model is needed.

When the plain rule comes up empty — the page renders the price oddly, or buries it in a script — the Lambda calls Bedrock Haiku 4.5. The prompt is short: "Here is the visible text of a product page. Return the current price, the currency, and a short rule describing where you found it. Return JSON only. Do not invent a price that isn't in the text." The output — product name, competitor, the price found, and a proposed *saved rule* for reading it next time — goes to a small Slack interactive message that pings whoever pasted the link: the proposed row, the price the monitor read, and three buttons — *approve*, *edit*, *discard*. On *approve*, a Lambda writes the row to the Drive sheet via the Sheets API. On *edit*, the person gets a fillable modal pre-populated with the proposal. On *discard*, the message is logged.

The reason every proposed row goes to a human first is simple: a price the monitor misread is worse than a page that never made it onto the list. The misread one will quietly report a wrong number every day until somebody notices the alerts don't match reality.

Lane 3: catalog import

Some teams already keep a list of which of their products map to which competitors — in their store admin, or a tagging column in their catalog export. Forcing those teams to also type rows in a separate sheet is a fight you don't need to have on day one.

Lane 3 picks up products in your own catalog that carry a "watch" tag and a competitor URL. A small `catalog-sync` Lambda runs hourly, reads the catalog export (from a store API or a nightly CSV drop, using a credential stored in Secrets Manager), and finds any newly tagged products. Each becomes a proposal in the

same Slack flow as Lane 2 — read once, propose a saved rule, one-tap approve to add to the watch list. Once approved, the catalog tag can stay where it is; the watch list now owns the page.

Catalog 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 keep elsewhere.

Why the watch list stays the source of truth

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

Next post: how the monitor actually checks a page, reads the price, compares it to history, and picks one of four moves.

PART 3 OF 7

MAY 22, 2026 PART 3 OF 7 · PRICE MONITOR SERIES ~5 MIN READ

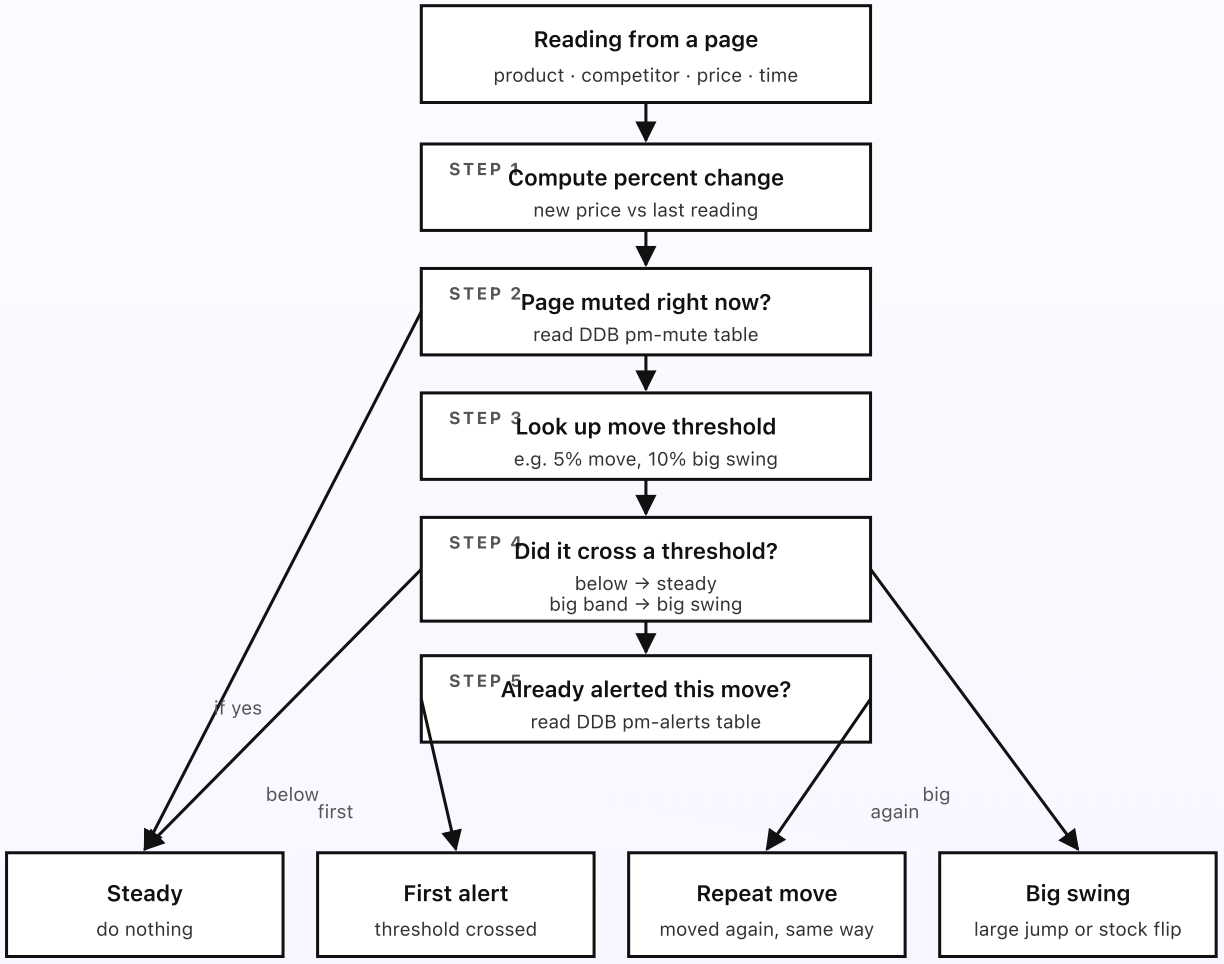
How a price move gets noticed

On each page’s schedule, an EventBridge Scheduler run fires the checker Lambda. The Lambda fetches the page politely, reads the price with that page’s saved rule, and writes the reading to history. Then the watcher looks at the new reading next to the last one, computes the change, and decides whether to stay quiet or to send an alert — and if so, which kind. The whole decision is plain Python. No model. No guessing. Every threshold lives in the rules doc, where a rep can edit it without a deploy.

KEY TAKEAWAYS

- The checks run on a staggered schedule via EventBridge Scheduler — gentle, never a burst.
- Each page has a move threshold in the rules doc — default 5%, with a bigger “big swing” band at 2× that.
- Four moves per page, every check: steady, first alert, repeat move, big swing.
- DynamoDB holds the last reading and last-alert state so the same move isn’t sent twice.
- The watcher itself never calls a model. The decision is entirely deterministic.

The decision flow, per page



The rules doc holds every threshold — change one and the next check uses the new value.

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

Thresholds: 5% isn't magic, it's in the doc

The rules doc has one short section per product line. Each section names the threshold in plain prose: "Kettles: alert on any move of 5% or more; treat 10% or a stock change as a big swing. Premium blenders: alert on 3%; they barely move. Clearance lines: alert on 8%; they bounce around." The first number is the move threshold — the smallest change worth an alert. The bigger number is the big-swing band — a jump large enough to flag as urgent the moment it happens.

The thresholds exist for a reason. A 5% move on a high-margin product is worth a glance. A 1% move is noise — rounding, a coupon, a regional tweak — and alerting on it just trains the owner to ignore the system. A big swing (a price halving, or a product going out of stock) usually means a real event — a clearance, a supply problem, a deliberate undercut — and deserves to jump the queue.

Per-page overrides exist too. The watch list sheet has an optional column called `threshold_override`. Type a percentage there and the watcher uses your number instead of the product-line default for that one row. This is the right escape hatch for the one product where even a 2% move matters because the margins are razor-thin.

Four moves, always

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

- **Steady.** The price didn't move past the threshold, or the page is muted. Do nothing. Most pages, most checks, are steady.
- **First alert.** The price crossed the move threshold and no alert has gone out for this move yet. Send a fresh alert with full context. Write a row to the `pm-alerts` DynamoDB table marking that this move has fired.
- **Repeat move.** The price moved again in the same direction since the last alert. Send a follow-up that names the previous reading so the owner sees the trend, not a fresh surprise. Write the new alert to `pm-alerts`.
- **Big swing.** A large jump — past the big-swing band — or a stock-state flip. Send it flagged as urgent, marked so dispatch can let it past the daily cap. A big swing usually means a real pricing event, so it's one of the few cases worth interrupting an owner for.

State that makes the decision deterministic

The watcher reads two DynamoDB tables every check. `pm-readings` holds the price history: `(page_id, ts)` with the price, currency, and stock flag. `pm-alerts` records every alert that's gone out: `(page_id, move, alert_date, dispatched_via)`. With those two tables, the move-decision logic is a few dozen lines of Python and zero magic. A given page with a given new reading, a given threshold, and a given history always produces the same move. Re-running a check produces no extra alerts (because the state in DDB shows what already fired).

A muted page is an explicit row in a third table, `pm-mute`, with a `mute_until` timestamp. The watcher reads it in the “page muted?” check and treats the page as steady until the mute ends. Part 5 covers muting in detail.

Why the check uses no model

The watcher could call a model on each check to decide whether a move is “interesting”. It doesn’t. Two reasons. First, the check should be the one part of the system that is utterly predictable — if the rules doc says alert at 5% and the price moved 7%, the alert fires. A model in that loop introduces variance the team can’t reason about. Second, model calls cost money, and most pages most checks are steady, so the call would be wasted nine times out of ten.

Bedrock fires elsewhere — when a page layout changes and the saved rule stops finding the price (the re-read lane from Part 2), and on the monthly summary mentioned in Part 6. Not on the routine check. The watcher itself is plain Python that reads a number and compares it to history.

Next post: how an alert finds the right person, how quiet hours and a daily cap are honored, and what context rides along with every alert.

PART 4 OF 7

MAY 22, 2026 PART 4 OF 7 · PRICE MONITOR SERIES ~5 MIN READ

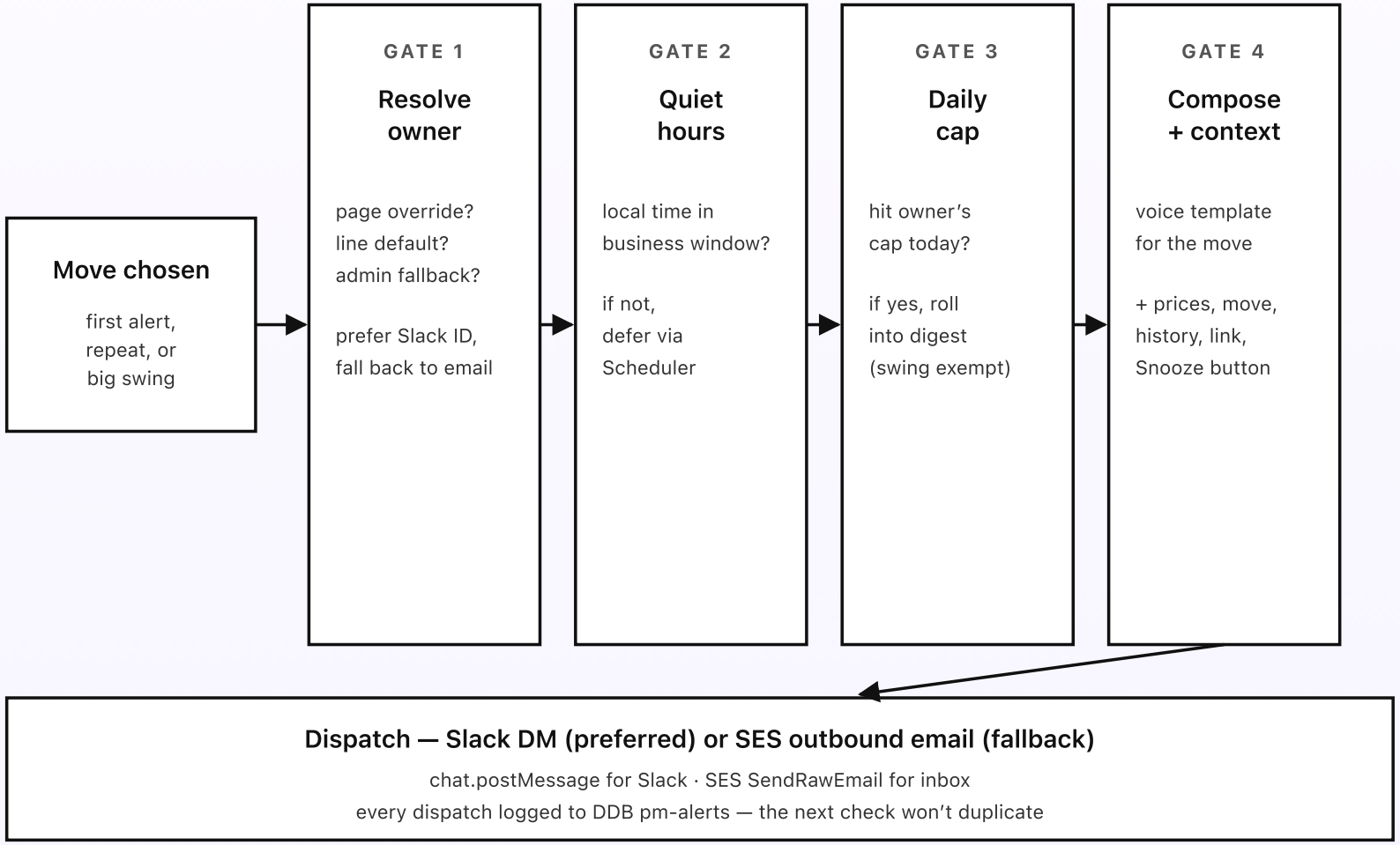
How a price alert reaches the owner

The watcher picked a move — first alert, repeat move, or big swing. 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 bare “a price changed,” a flood of twenty alerts during a competitor’s sale day. Four small guardrails sit between the move and the actual alert.

KEY TAKEAWAYS

- Owner resolution: per-page override beats per-product-line default beats fallback to the configured admin.
- Slack DMs are the default; email is the fallback if no Slack ID is configured.
- Quiet hours defer alerts to the next business hour; a daily cap stops a sale day from burying an owner.
- Every alert ships with the product, competitor, old and new price, the size of the move, recent history, and your own price.
- A big swing is allowed past the daily cap; the system never edits a price either way.

Four guardrails on every dispatch



Every gate is a deterministic check — no model calls, no surprise behavior on a busy sale day.

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

Gate 1: resolve the owner

Three places the dispatch Lambda looks for the owner of a page, in order. First, the watch list's per-page `owner_email` column — if a row has a specific person assigned, that person owns it regardless of the product-line default. Second, the per-product-line default in the rules doc ("all kettle pages default to Priya"). Third, the configured admin fallback — the person who set up the monitor and gets every unowned alert. The fallback should never fire in steady state; if it does, the weekly digest names every page that hit the fallback so the rules doc can be updated.

Once the dispatch knows which person to alert, 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 price moves feel like work-context messages, and a Slack DM with a Snooze button is more useful than an email link. Email is the fallback so nobody falls through the cracks.

Gate 2: quiet hours

The checks are staggered through the day, so an alert can land at any hour — including outside business hours. A price that drops at 11pm shouldn't buzz an owner's phone at 11pm; it can wait until morning, because nobody's changing a price overnight anyway.

Gate 2 reads the rules doc's quiet-hours setting (default 7pm 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. The one exception is a big swing flagged urgent, which the rules doc can allow to bypass quiet hours if the business wants to know the moment a competitor halves a price.

Gate 3: daily cap

A competitor running a storewide sale can move twenty of their products in an hour. Without a cap, that's twenty Slack DMs in an owner's morning — the fastest way to teach someone to mute the whole system. Gate 3 counts how many alerts have already gone to this owner today against a per-owner cap (default eight). Once the cap is hit, further first-alert and repeat-move alerts are rolled into that owner's daily digest instead of sent live: a single message listing everything that moved, sent at a set time.

A big swing is exempt. The whole point of the big-swing band is "this is unusual enough to interrupt for," so it always goes through, cap or not. The cap shapes the ordinary noise without hiding the rare events that matter.

Gate 4: compose with full context, then ship

The voice doc has one Slack message template per move: a short message with placeholders for the product name, competitor, old and new price, the percent move, a short recent history, your own listed price, and a link to the competitor

page. The dispatch Lambda fills the placeholders, attaches a “Snooze” button (covered in Part 5), and ships the message via the Slack `chat.postMessage` API. The Slack token 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 a snooze or a note — the email equivalent of the Slack buttons.

The recent history is what makes a price alert worth reading. “Competitor Acme dropped the kettle 13%” is useful; “...and this is the third cut in two weeks, the price has gone $79 → $74 → $69” tells the owner whether this is a one-off promo or a sustained campaign. The dispatch pulls the last several readings from `pm-readings` and renders them as a tiny text sparkline so the trend is visible at a glance.

Every dispatch — Slack or email, any move — writes a row to `pm-alerts` in DynamoDB. The next check reads that row and knows not to alert the same move again.

Why the guardrails exist

None of these gates are exotic. They’re the kind of small care a thoughtful person would take if they were sending the alerts themselves — check who actually owns this product, don’t ping at 11pm, don’t send twenty messages during one sale, include enough context that the recipient can decide without opening five tabs. Putting them in code as four small sequential gates makes them part of the design, not something you’re trusting any one alert to remember. And note what

no gate does: none of them touch a price. The system's job ends at telling the owner clearly.

Next post: how a price alert gets handled once it lands — the three actions an owner can take, and why none of them ever change your own prices.

PART 5 OF 7

MAY 22, 2026 PART 5 OF 7 · PRICE MONITOR SERIES ~5 MIN READ

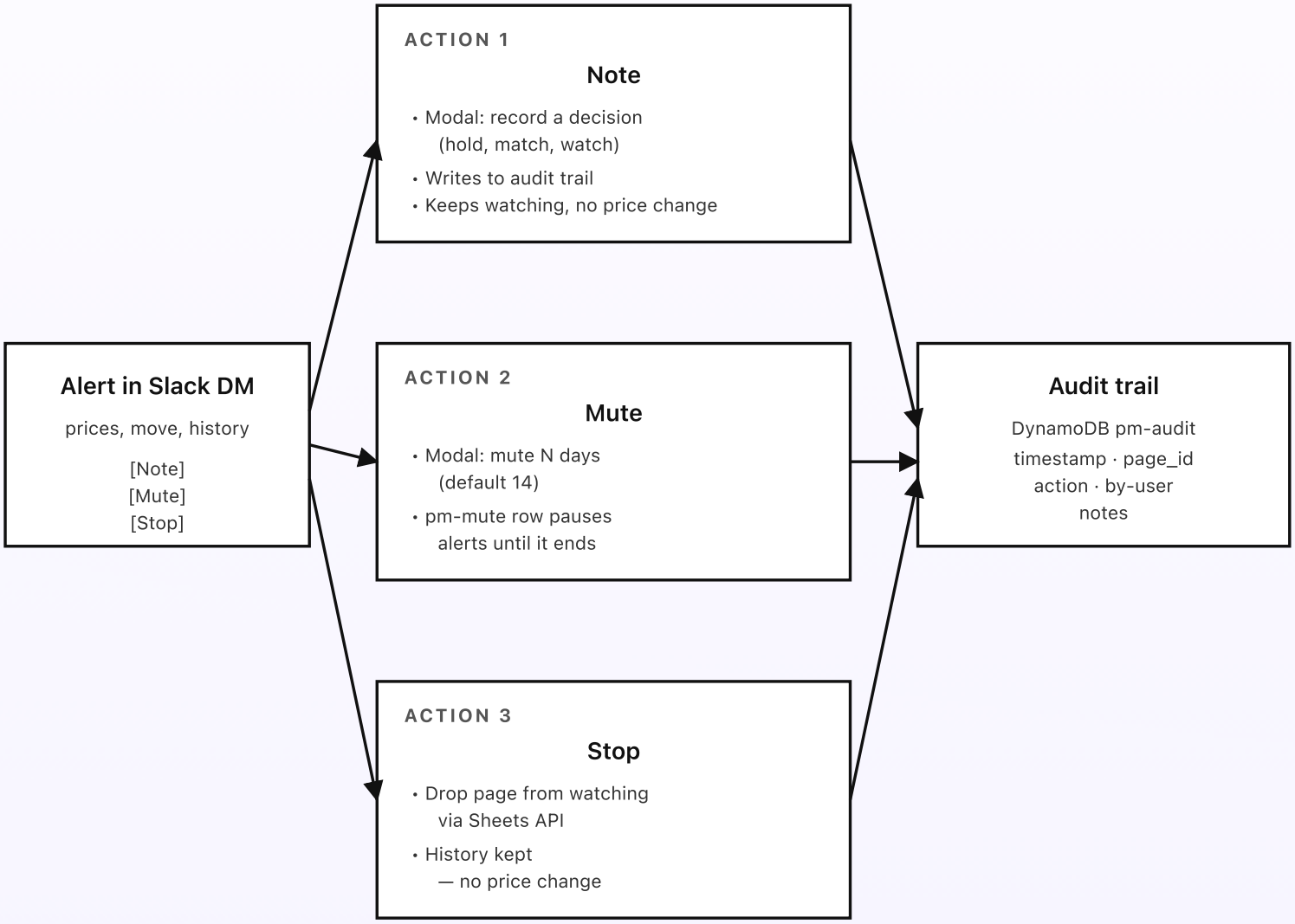
How a price alert gets handled

An alert lands in Priya's Slack DM at 9:12am. A competitor dropped the kettle 13%. There are three buttons. What happens when she taps one? Here's the most important thing about this whole post: none of the three buttons changes your own price. The monitor watches and tells; the pricing decision stays with Priya. This post walks through the three things she can do on an alert — note, mute, stop — and how the watch list, the page state, and the audit trail stay in sync.

KEY TAKEAWAYS

- Three actions per alert: *note* (log a decision, keep watching), *mute* (silence this page for a while), *stop* (drop the page).
- None of the three actions changes your own price — the monitor only ever suggests.
- Each action writes an audit row; mute and stop also update the page's state.
- Mute is bounded — it lasts a set number of days, then the page resumes on its own.
- The buttons are a Slack interactive message backed by a Function URL.

Three actions on an alert



No action here changes your own price — the monitor only suggests, and a person decides.

Fig 5. Three actions per alert, three different effects. Note logs a decision and keeps watching. Mute silences the page for a set time. Stop drops it from the list. Every action writes to the audit trail — and none of them touch your price.

Action 1: note (the most common)

Priya looks at the alert, decides how to respond, and wants to record it. She taps *Note*. A small Slack modal opens with a single free-text box and a couple of quick-pick chips — “holding price,” “will match,” “watching.” She types “holding — our reviews are stronger and margin is tight” and hits Save.

The Save button submits to a Function URL Lambda. Two things happen, in order. First, an `action: note` row is written to `pm-audit` with the user, timestamp, the alert that prompted it, and her text. Second, the live-alert flag for this move on this page is cleared, so the monitor keeps checking the page but won’t re-nag Priya about this same move — only a fresh move past the threshold will alert again.

What doesn’t happen is just as important. The note doesn’t change Priya’s price, doesn’t touch her store, and doesn’t feed back into any auto-pricing — there isn’t one. The note is a memory for the next time this competitor moves, and a record for whoever reviews pricing decisions later. If she decides to match, she goes and changes her price herself, the way she always would; the monitor just made sure she knew to.

Action 2: mute (the deferral)

Some competitors are noisy on purpose. A clearance specialist re-prices daily. A marketplace seller runs a sale every weekend. After the third alert in a week saying the same thing, the alerts stop being signal and start being noise. Priya isn't ignoring the page — she just wants the monitor to be quiet about it for a while.

Mute opens a small modal asking for the number of days, with a 14-day default and a max of 90. On save, a row is written to `pm-mute` with `(page_id, mute_until)`. The next check reads that row in the "page muted?" step from Part 3 and treats the page as steady until the mute ends. Crucially, the page is still *checked* and its history is still recorded the whole time — only the alerting is paused. When the mute ends, the monitor picks up where the history left off, so a price that moved during the mute is caught at the next check.

Mute is bounded by design. It always has an end date; there is no "mute forever" button, because a page muted forever is a page quietly lying to you. If a competitor really is no longer worth watching, the right action is *stop*, not an endless mute.

Action 3: stop (the "we're done with this one")

Sometimes a page just isn't worth watching anymore. The competitor discontinued the product. They went out of business. You stopped selling that line. Keeping a dead page on the list means a daily check that can only ever fail or mislead.

Stop drops the page from active watching. The Function URL Lambda flags the row in the watch list as `stopped` via the Sheets API, and the next check skips it.

The page's history in `pm-readings` is kept — you might want to look back at what a discontinued competitor used to charge — but no new readings are taken. An `action: stop` row goes to `pm-audit` with the user and reason, so a later "why did we stop watching Acme?" has an answer.

Stop is reversible: the row is still in the sheet, just flagged. Un-flag it (edit the sheet, Lane 1 from Part 2) and the next check resumes watching from a fresh reading. Nothing is deleted, so a stop made in haste costs nothing to undo.

Every action is logged, and nothing touches your price

The `pm-audit` table records every note, mute, and stop with the user who took the action, the timestamp, and a snapshot of the relevant state before and after. Three months from now, when someone asks "why aren't we getting alerts on that competitor?" or "did we ever decide whether to match their sale?", the answer is one query away.

And it's worth saying one more time, because it's the design rule that shaped this whole system: there is no button, no modal, no background job that changes your own prices. The monitor watches competitor pages, notices real moves, and tells the right person with enough context to decide. The decision — match, hold, undercut, ignore — is always made by a human who can weigh things a threshold never could. That's not a limitation; it's the point.

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.

PART 6 OF 7

MAY 22, 2026 PART 6 OF 7 · PRICE MONITOR SERIES ~3 MIN READ

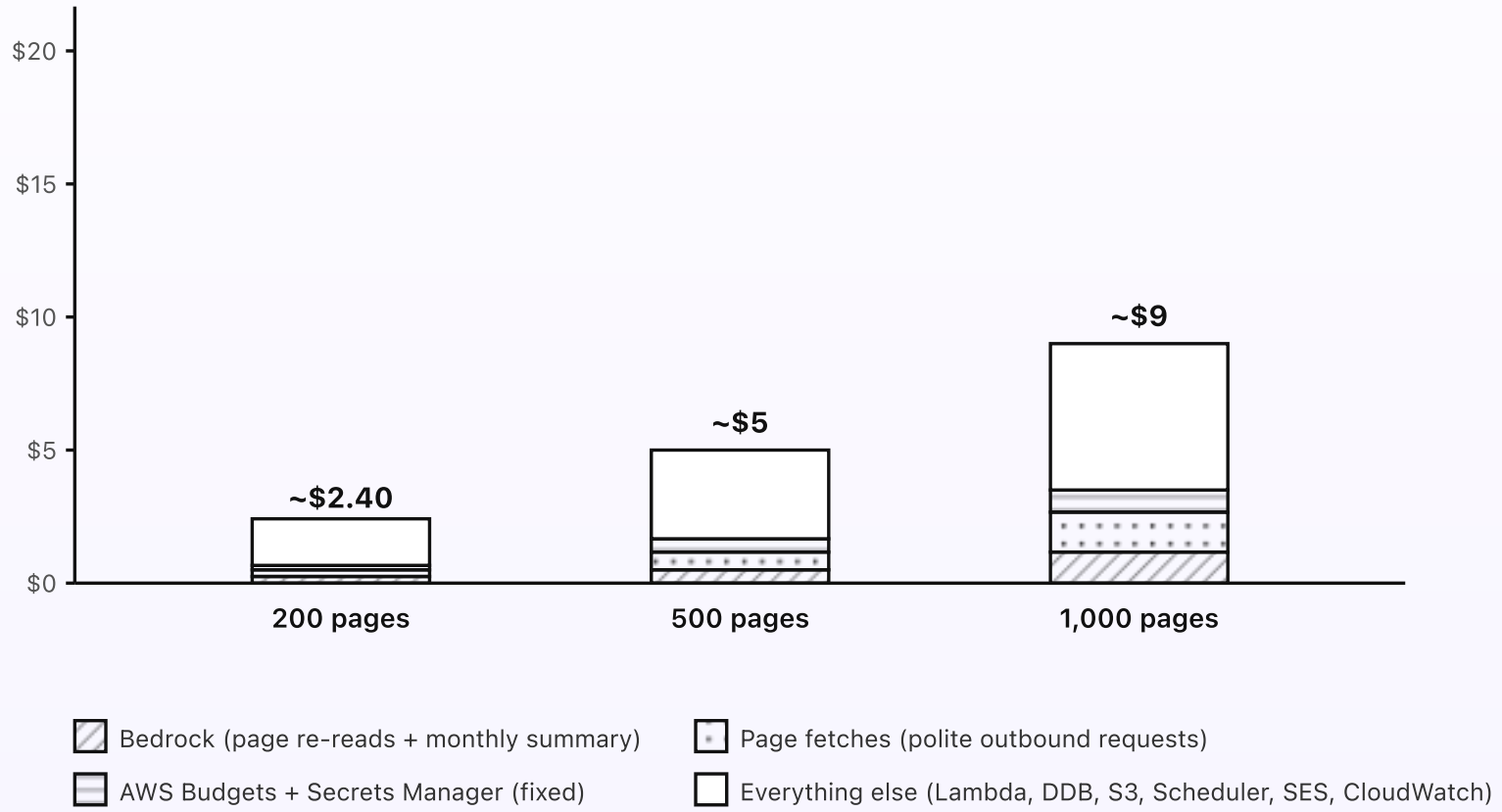
What the price monitor costs

The monitor is one of the cheapest systems in this whole series. Each check fetches a page, reads one number, writes a row to DynamoDB, and only sometimes posts a message to Slack. It calls no models on the routine check. Bedrock fires only when a page layout changes and the saved rule stops finding the price, and once a month for the summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (around 200 watched pages).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The routine check costs pennies — no model calls.
- Bedrock fires only when a page layout changes (a few times a month) and on the monthly summary.
- At 500 watched pages the bill is around $5. At 1,000 pages it's around $9.

Cost at three volumes



The scheduled checks are the dominant cost — and even that is fractions of a cent per page per check.

Fig 6. Monthly cost at three watched-page volumes. Bedrock and page re-reads are small slivers because Bedrock only fires when a layout changes and on the monthly summary. The dominant cost is the everything-else bucket: the scheduled checks reading every page.

Where the dollars actually go

Lambda runtime (the bulk). The checker runs on each page's schedule — once or twice a day for most pages. Each run fetches the page, reads one number with the saved rule, and writes a reading. At 200 pages checked twice a day, that's a few hundred short invocations a day, each a fraction of a second. Either way it's pennies a month. Add the watcher deciding moves, the dispatch Lambda firing for each alert (a handful a day at most), the Function URL Lambda for note/mute/stop, the catalog-sync Lambda running hourly, and the drive-sync Lambda every fifteen minutes — the Lambda total still lands a dollar or two at all three volumes.

DynamoDB on-demand. Four small tables: `pm-readings`, `pm-alerts`, `pm-mute`, `pm-audit`. Writes are dominant — one reading per page per check. Reads happen on the watcher's compare step and on the weekly digest. Pennies a month at any of these volumes.

S3 + Storage. The mirrored watch-list CSV plus saved page snapshots kept for the re-read lane. A few MB at SMB volume. Effectively free.

EventBridge Scheduler. The staggered check rules plus deferred dispatch rules from the quiet-hours gate. A few hundred invocations a day. Pennies.

Page fetches. The outbound requests themselves cost nothing in AWS fees beyond the tiny Lambda time and data transfer to fetch a page (a few KB to a couple hundred KB each). At a few hundred fetches a day, transfer is cents a month. The real constraint on fetches is politeness, not cost — the gentle rate is a courtesy to the sites, and it happens to keep the bill down too.

Bedrock (only when something fires it). The routine check uses no Bedrock. The re-read lane fires Haiku 4.5 only when a page layout changes and the saved rule stops finding the price: a few thousand input tokens (the page text) and a few hundred output tokens (the new rule), so a fraction of a cent per re-read. Across a watch list, layouts change a handful of times a month, so Bedrock costs cents. The monthly summary is one larger call: write a short narrative of the month's moves; a couple of cents.

SES. Outbound for the email-fallback alerts and the monthly summary: $0.10 per thousand sent. Negligible at this scale.

What doesn't cost money

- **API Gateway.** Replaced by a Lambda Function URL for the note/mute/stop endpoint.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The checker sleeps between scheduled runs.
- **A headless browser farm.** Most product pages give the price up in their markup, so a plain fetch and parse is enough. No fleet of full browsers rendering pages, which is where most price-scraping bills explode.
- **Models on the check.** The routine decision is plain Python. Bedrock fires only on page re-reads and the monthly summary.

How the cost scales

Lambda runtime grows roughly linearly with page count, because every page is fetched and checked on its schedule. DynamoDB grows linearly too. Bedrock is mostly uncorrelated with page count — it only fires when a layout breaks a saved rule or it's the first of the month. So the bill at 3,000 watched pages is around $25; at 5,000 it's around $40. Past those volumes you'd batch fetches more aggressively and lengthen the check interval on slow-moving pages, but those are tuning knobs, not redesigns.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The monitor'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, the polite-fetch policy, and EventBridge Scheduler config.

PART 7 OF 7

MAY 22, 2026 PART 7 OF 7 · PRICE MONITOR SERIES ~8 MIN READ

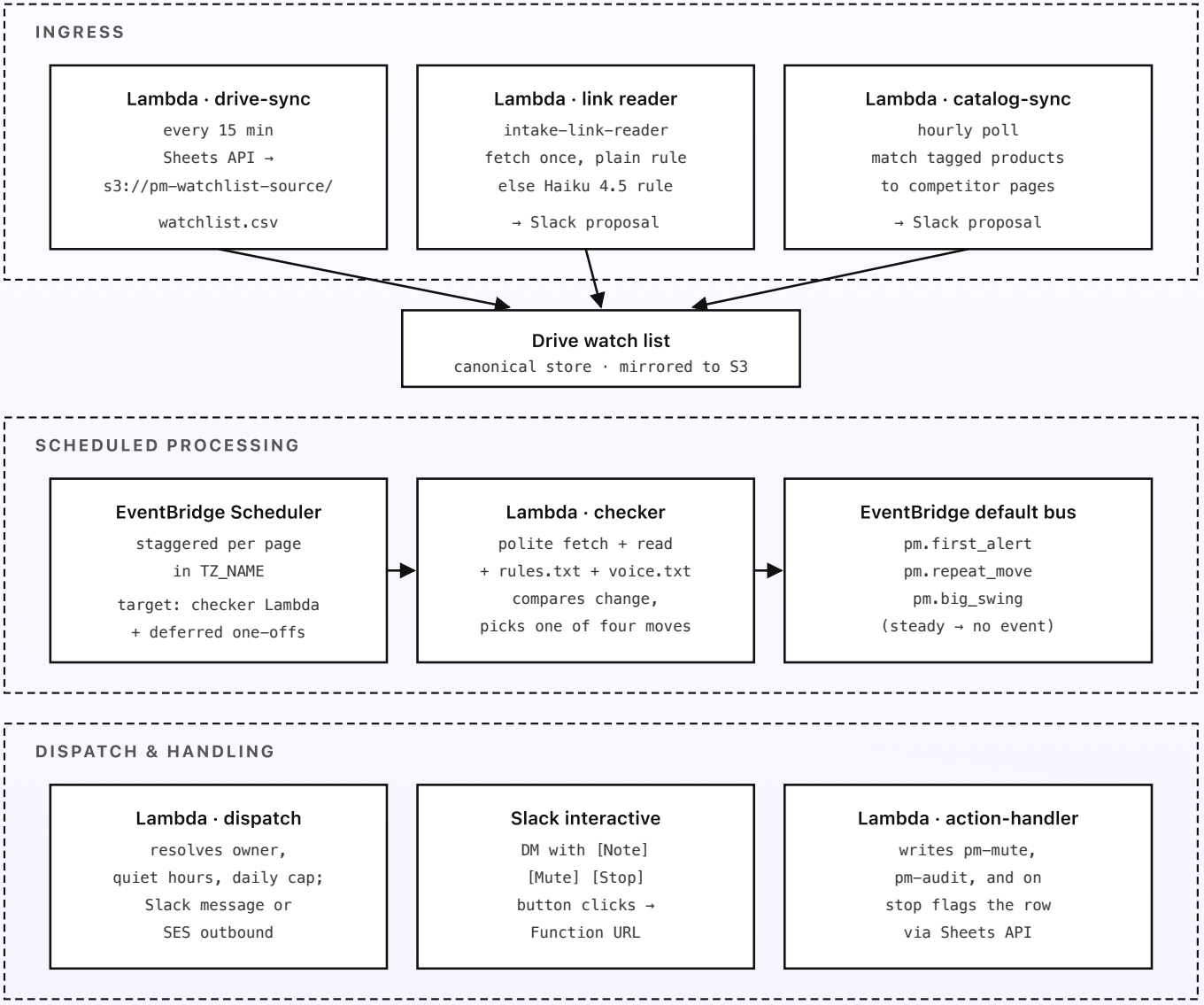
Engineering reference: the price monitor architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the polite-fetch policy, 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). Bedrock cross-Region inference, EventBridge Scheduler, and SES outbound are all in good shape there. A second region for resilience isn't worth the extra setup at SMB volume — the failure mode for an SMB is missing a price move for a day, not a regional outage. One AWS account dedicated to the monitor (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. Note the deliberate absence of any write path to a store or catalog — the monitor has no credentials that could change a price anywhere.

Topology



The monitor only suggests — every decision is logged to pm-audit and no price is ever changed.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the watch list), scheduled processing (the staggered checks emitting events), dispatch and handling (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; outbound fetches go straight to the internet over the Lambda's default egress.

- **`drive-sync`** — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `pm/drive/sa`) to export the watch-list sheet as CSV and write to `s3://pm-watchlist-source/watchlist.csv` only if the sheet has changed since the last sync. The same pattern syncs the rules and voice docs to `s3://pm-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`intake-link-reader`** — invoked by the Slack `/watch` slash command (via the `action-handler` Function URL) or by a dedicated channel webhook. Fetches the pasted URL once through the polite-fetch helper, tries a deterministic price rule (JSON-LD `offers.price`, schema.org microdata, OpenGraph `product:price:amount`, then a labelled-element heuristic). On miss, calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) with the cleaned

visible text to propose a price and a saved selector rule. Posts the proposal to Slack with Approve/Edit/Discard. Memory: 512 MB. Timeout: 60 s.

- **`catalog-sync`** — EventBridge Scheduler target, hourly. Reads the catalog export (store API or a nightly CSV drop in `s3://pm-catalog-source/`) for products tagged to watch, finds newly tagged rows with a competitor URL, and routes each through the same propose-and-approve flow as the link reader. Read-only against the catalog — it never writes back. Memory: 256 MB. Timeout: 30 s.
- **`checker`** — EventBridge Scheduler targets, staggered per page across the day (the schedule expressions run in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). For each page due, runs the polite-fetch helper, applies the saved price rule, and writes a reading to `pm-readings`. Then runs the watcher logic inline: computes the percent change against the last reading, reads `pm-alerts` and `pm-mute`, decides on a move. Emits one event per page that needs action: `pm.first_alert`, `pm.repeat_move`, or `pm.big_swing`, with the page context as the payload. Steady pages emit nothing. If the saved rule misses, enqueues a re-read job to `intake-link-reader` instead of alerting. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls on the read/compare path.*
- **`dispatch`** — EventBridge rule on the three move events. Resolves owner, checks quiet hours and the per-owner daily cap, formats the alert from the voice template (including a short text sparkline from the last N readings), and ships via Slack `chat.postMessage` (`pm/slack/bot-token` in Secrets Manager) or SES `SendRawEmail`. On a quiet-hours defer, creates a one-off EventBridge Scheduler rule that re-invokes `dispatch` at the next business minute. On a cap hit, appends to that owner's digest buffer instead of sending (big swings

bypass the cap). Writes a row to `pm-alerts` after a successful send. Memory: 256 MB. Timeout: 30 s.

- **`action-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive button clicks (Note/Mute/Stop), the `/watch` slash command, and email-link clicks. Writes to `pm-mute` and `pm-audit`; on stop, flags the watch-list row via the Sheets API. Holds no credential that can edit any price on any store. Memory: 256 MB. Timeout: 15 s.
- **`digest`** — EventBridge Scheduler target, weekly Sunday 6pm plus a daily flush of any capped alerts. Reads `pm-alerts` and `pm-readings`; sends a digest to a configured Slack channel summarizing moves, capped alerts, and any pages that failed to read. No Bedrock; a plain summary table. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `pm-readings`, `pm-alerts`, and `pm-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph narrative of how the market moved and where you sit; emails it via SES to the configured stakeholder list. Memory: 512 MB.

Polite-fetch policy

The single shared `fetch` helper used by `checker` and `intake-link-reader` enforces the etiquette so no individual function can forget it:

- **robots.txt.** Parsed and cached per host (TTL 24h in `pm-robots`); a disallowed path is never fetched and the page is flagged in the watch list as blocked. Any `Crawl-delay` is honored as a floor on that host's interval.

- **Identification.** A descriptive `User-Agent` naming the business and a contact URL. No browser-impersonation, no rotating agents.
- **Rate.** One request per page per scheduled run, staggered across the day; a per-host concurrency of 1 with a minimum gap so two pages on the same host never fire together.
- **Backoff.** On a 429 or 5xx, exponential backoff with jitter; after a configurable number of consecutive failures the page is paused (a `paused` flag in `pm-readings` state) and surfaced in the weekly digest rather than retried in a tight loop.
- **Footprint.** Plain HTTP fetch with a small response cap; no headless browser, no JavaScript execution unless a per-page `render: true` flag is set (rare, and rate-limited harder). Conditional requests (`ETag` / `If-Modified-Since`) are used where the host supports them.

Storage

- **DynamoDB · `pm-readings`** — one row per check. PK `page_id`; sort key `ts`; attributes: `price`, `currency`, `in_stock`, `http_status`, `paused`. On-demand. TTL on raw readings at 400 days (history beyond that is aggregated into the monthly summary).
- **DynamoDB · `pm-alerts`** — one row per dispatch. PK `(page_id, move)`; attributes: `alert_date`, `dispatched_via` (slack/email/digest), `recipient`, `old_price`, `new_price`, `pct`. On-demand.
- **DynamoDB · `pm-mute`** — one row per active mute. PK `page_id`; attributes: `mute_until`, `by_user`, `reason`. On-demand. TTL on `mute_until` so expired

mutes self-clean.

- **DynamoDB** · `pm-audit` — one row per write action of any kind. PK `(page_id, ts)`; attributes: `action` (note/mute/stop/approve), `by_user`, `before`, `after`, `note`. On-demand. No TTL — long-term decision trail.
- **S3** · `pm-watchlist-source` — mirrored CSV from the Drive watch-list sheet. Versioning enabled. Lifecycle to Glacier at 90 days.
- **S3** · `pm-rules-source` — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3** · `pm-snapshots` — the saved HTML snapshot from the last successful read of each page, used by the re-read lane when a layout breaks the saved rule. Lifecycle expiry at 30 days.

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-link-reader` for proposing a price rule when the deterministic rules miss, and `summary` for the monthly narrative. A heavier reasoning model (`anthropic.claude-sonnet-4-6`) is not used here — reading a price off a page is a Haiku-class task.
- **Embeddings.** Not used. The watch list is structured rows and price extraction is deterministic-first; no vector retrieval, no Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The routine check doesn't call Bedrock; the re-read lane fires only when a layout changes.

EventBridge Scheduler config

- `pm-check-{bucket}` — a small set of staggered `rate(...)` schedules (e.g. four buckets at `rate(6 hours)` offset by 90 minutes) so pages spread across the day. Target: `checker` Lambda with the bucket id as input.
- `pm-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `pm-catalog-sync` — `rate(1 hour)`. Target: `catalog-sync` Lambda.
- `pm-weekly-digest` — `cron(0 18 ? * SUN *)` in TZ, plus `pm-daily-flush` `cron(0 17 * * ? *)` for capped alerts. Target: `digest` Lambda.
- `pm-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` for quiet-hours defers. Use `at(YYYY-MM-DDTHH:MM:SS)` with `--action-after-completion DELETE` so the rule self-cleans.

SES outbound and Slack

- SES outbound for the email-fallback alerts and the monthly summary: verify a sender identity at `monitor@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request. No SES inbound is used — this system has no email intake lane.
- Alert messages are posted via the Slack `chat.postMessage` Web API with Block Kit blocks containing the Note/Mute/Stop buttons. Button clicks and the `/watch` slash command are sent by Slack to the `action-handler` Function URL, which verifies the signing secret, parses the `action_id` (`note`, `mute`,

`stop`, `watch`), opens a modal where needed (Note/Mute open modals; Stop is one-tap), and processes the submission.

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

IAM (least privilege per Lambda)

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

- **checker role:** `s3:GetObject` on the watch-list, rules, and snapshot keys; `s3:PutObject` on `pm-snapshots`; `dynamodb:Query` + `PutItem` on `pm-readings`, `dynamodb:Query` on `pm-alerts` and `pm-mute`; `events:PutEvents` on the default bus. *No* `bedrock:*`. Outbound internet for fetches.
- **dispatch role:** `scheduler:CreateSchedule` for the deferred-dispatch one-offs; `secretsmanager:GetSecretValue` on the Slack bot token; `ses:SendRawEmail` from the verified sender identity; `dynamodb:PutItem` on `pm-alerts`; outbound network to `slack.com`.
- **action-handler role:** `dynamodb:PutItem` on `pm-mute` and `pm-audit`; `secretsmanager:GetSecretValue` on the Sheets-API service-account secret; outbound network to `sheets.googleapis.com`; `lambda:InvokeFunction` on `intake-link-reader` for the `/watch` command. *No* store or catalog write scope.
- **intake-link-reader role:** `s3:GetObject`/`PutObject` on `pm-snapshots`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token; outbound internet for fetches.

- **drive-sync and catalog-sync roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the watch-list and rules buckets; `s3:GetObject` on `pm-catalog-source` (catalog-sync only, read-only); outbound network to `www.googleapis.com`.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"blocked"` to a CloudWatch metric for alerting.
- **Alarms:** checker failure rate > 5% in 24h (covers a widespread layout break or a host blocking us); dispatch failure rate > 1% in 24h; action-handler signature-verification failures > 5/hour (might mean the Slack secret rotated); a "stale page" metric for any page with no successful read in 48h.
- **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 `pm-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive, Sheets, and the catalog API live in Secrets Manager under `pm/drive/sa` and `pm/catalog/*`. Slack bot token and signing secret under `pm/slack/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, quiet-hours window, per-owner daily cap, default move threshold, and admin fallback owner all live in Parameter Store

under `/pm/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

Whichever IaC you prefer. The opinionated bits: turn on S3 versioning for both `pm-watchlist-source` and `pm-rules-source` so a bad Drive edit can be rolled back in one click; keep the polite-fetch helper in a shared layer so etiquette is enforced in one place; and version the EventBridge Scheduler timezone setting so you don't accidentally start checking on a UTC clock after a CI rotation. CDK with a Python stack file works well; SAM also fits. Total deployable surface: around eight Lambdas, four DDB tables, three S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SES sender identity, and one Budgets alarm. Note what isn't in the surface: any credential or path that could change a price.

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.