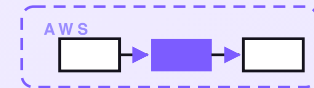
7-PART SERIES · FREE COMPANION



Backup sentinel

A serverless watcher that makes sure your backups actually worked — because a backup you never checked isn't a backup. It watches each backup job you point it at — databases, file stores, cloud drives — confirms each one finished, is recent, and is the right size, and warns the right owner the moment one is missing, stale, or shrank suspiciously. It only watches and warns; it never deletes or restores anything. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

Workflow guide $19 · Deployable AWS CDK starter $79 · Bundle $89

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/backup-sentinel

CONTENTS

Backup sentinel

01 A backup sentinel on AWS for a few dollars a month

02 How a backup job gets registered

03 How a backup gets checked

04 How a backup alert reaches its owner

05 How a backup status gets cleared

06 What the backup sentinel costs

07 Engineering reference: the backup sentinel architecture

PART 1 OF 7

JUNE 8, 2026 PART 1 OF 7 · BACKUP SENTINEL SERIES ~5 MIN READ

A backup sentinel on AWS for a few dollars a month

Most small businesses set up backups once and never look at them again. The nightly database dump that started failing in March because a disk filled up. The file-server sync that has been copying an empty folder since somebody renamed a share. The cloud-drive export that quietly stopped when an access token expired. Each one looks fine from a distance — there's a job, it's scheduled, nobody touched it — right up until the morning you need to restore and find the last good copy is six months old. A backup you never checked isn't a backup. This post walks through the design of a small watcher that checks all of them, every day, and warns the right person the moment one stops working.

KEY TAKEAWAYS

- Three sources for registered jobs: a Drive job list, an inbox forwarding lane, and a heartbeat lane.
- Every job ends in one of four states on each check: all green, warn, alert, or escalate.
- Three plain tests per job: did it finish, is it recent enough, is it the right size.
- Pings respect quiet hours and your holiday calendar. A cleared job stops pinging.
- It only watches and warns — it never deletes or restores. Designed on AWS for about $2/month.

The whole system on one page

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

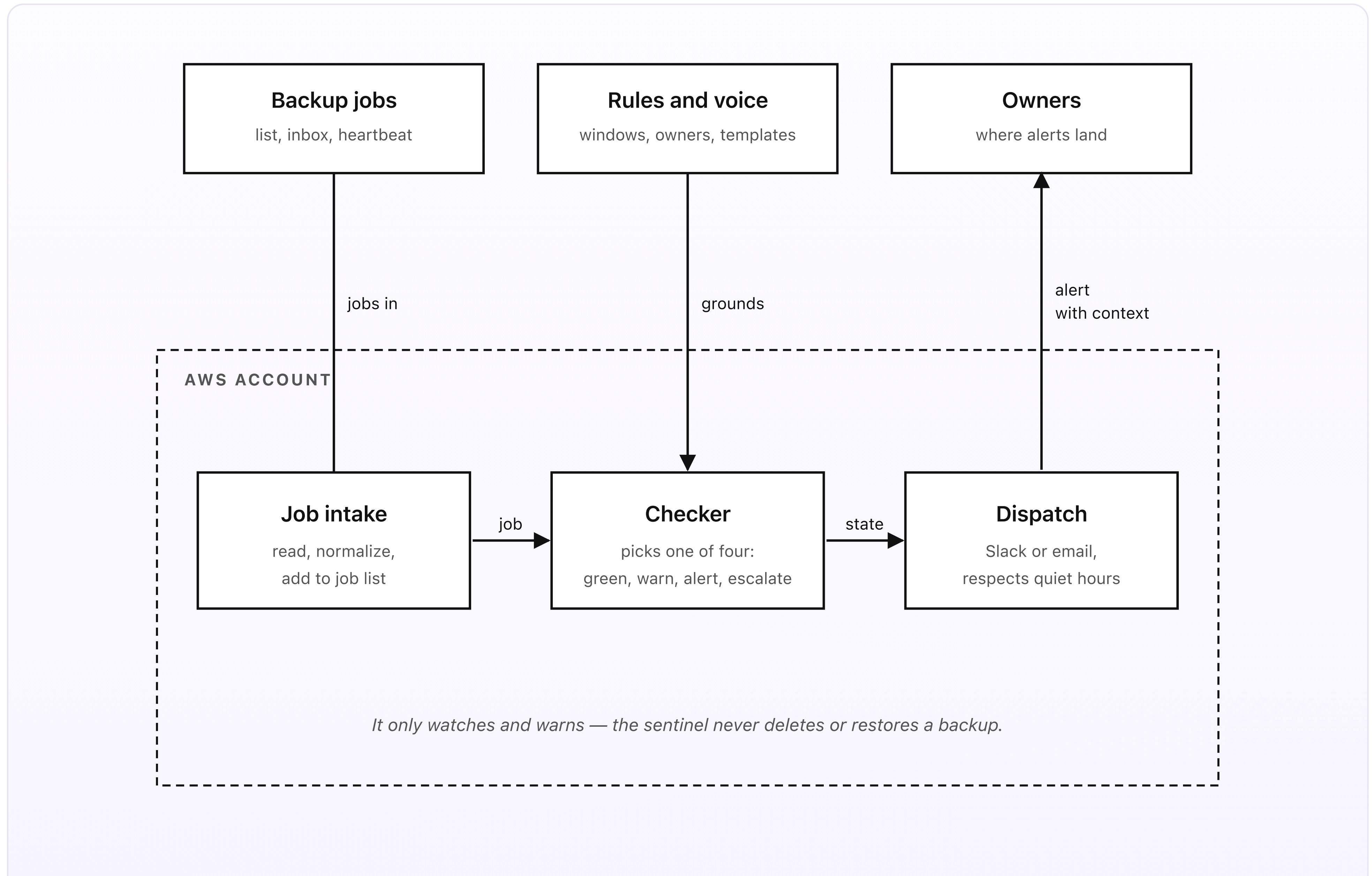

Backup jobs
list, inbox, heartbeat
Rules and voice
windows, owners, templates
Owners
where alerts land
jobs in
grounds
alert
with context
AWS ACCOUNT
Job intake
read, normalize,
add to job list
job
Checker
picks one of four:
green, warn, alert, escalate
state
Dispatch
Slack or email,
respects quiet hours
It only watches and warns — the sentinel never deletes or restores a backup.

Fig 1. Three sources outside, three pieces inside AWS. Jobs flow in from a Drive job list, an inbox forwarding lane, and a heartbeat lane. The Checker runs on a schedule and picks one of four states. Dispatch sends the right alert to the right person at the right time.

What you set up once (the outside)

- **Backup jobs.** A Google Sheet in a Drive folder, one row per job: name, what it backs up (e.g. `orders database`, `shared drive`, `QuickBooks export`), owner email, where the backup lands (an S3 path, a folder, a bucket), how often it should run, the smallest size you'd expect, and how late is too late. You can fill it in once and forget it; new jobs can also enter via two other lanes covered in Part 2 — an inbox-forwarding lane (forward the report your backup tool emails out and the sentinel proposes a row for one-tap approval) and a heartbeat lane (a job calls a private web address when it finishes, and the sentinel registers it the first time it hears from it).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the freshness window for each job — how recent the last good run has to be before the sentinel worries — plus the smallest size that counts as a real backup, the owner per job (or per group), the escalation target if the owner doesn't act, the quiet hours, and any holiday calendars to skip. A nightly database dump might allow 26 hours; a weekly file export might allow 8 days. The *voice* doc holds one alert message template per kind of problem — what the Slack DM or email actually says when a job is missing, stale, or too small.
- **Owners.** The people responsible for each job. 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 job name, exactly what failed, how late or

how small it is, a link to the evidence, and buttons to mark it fixed, snooze it, or mute it.

What runs on every check (the inside)

- **The job intake.** Three sources feed the job list. The Drive sheet itself is the canonical store. New jobs can also be added via the inbox forwarding lane (forward the success-or-failure email your backup tool sends to `backups@your-company.com`; the sentinel reads it and drops a one-tap approval card in the team's Slack to confirm before the row is added) and the heartbeat lane (a job sends a quick ping to a private web address each time it finishes; the first ping proposes a row).
- **The checker.** Runs on a schedule — hourly for jobs that run often, or a few times a day for the rest. Reads each job's latest evidence: the report it left behind, the marker file in the backup folder, or the most recent heartbeat. For each job it runs three plain tests. *Did it finish?* — was there a successful run at all. *Is it recent?* — is the last good run inside the freshness window. *Is it the right size?* — is the latest backup at least the smallest expected size, and didn't it shrink sharply from the run before. Then it picks one of four states. *All green*: all three tests pass. *Warn*: a soft miss — slightly late, or a little smaller than usual. *Alert*: a hard miss — missing, well past the window, or far too small. *Escalate*: still broken after the owner had a fair chance to act. The checker doesn't call a model on the check — the test logic is plain Python.
- **Dispatch.** Reads the voice doc, formats the alert for the chosen state and problem, and sends it. Slack DMs go through the Slack API. Email goes through SES outbound. Both honor quiet hours (no pings between 6pm and 8am local by default) and the holiday calendar. Every dispatch writes a row in DynamoDB

so the next check can tell whether the owner has acted. A daily summary writes one plain-English line per job — "all green" or "here's what's wrong" — so the owner gets a calm morning read even on quiet days.

In plain words

Your orders database is dumped to S3 every night at 2am by a small script. The sentinel knows this job should produce a file under `s3://acme-backups/orders/` every 24 hours, at least 400 MB, owned by your ops lead Sam. One Tuesday the dump script dies halfway because the disk it writes to first is full. No file lands. At the 8am check the sentinel reads the folder, sees the newest file is from 2am *yesterday* — 30 hours old, past the 26-hour window — and the state flips from all green to alert. Sam gets a Slack DM: "Orders database backup — no new file since Mon 2:04am, 30h late (window 26h). Last good file 412 MB. *[link to folder]*" with Mark fixed / Snooze / Mute buttons. Sam clears the disk, re-runs the dump, sees a fresh 415 MB file land, and taps *Mark fixed*. The next check sees the fresh file, confirms it, and the job is green again.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the one morning you reach for a backup that has silently been failing since spring — and the newest copy you have is from before the thing you need to undo ever happened.

DESIGN RULES THAT SHAPED EVERY DECISION

- It only reads. The sentinel never deletes, moves, or restores a backup — the worst it can do is send an alert.
- Four states, always. All green, warn, alert, escalate. There is no fifth.
- Three plain tests per job: did it finish, is it recent, is it the right size. No magic.
- Only a state change pings you. A still-broken job stays quiet until it crosses into escalation.
- The job list lives in Drive. Adding a job, changing an owner, or shifting a window doesn't need a deploy.
- Every check and every action is logged. Audit a recovery next year and you can see every alert that went out.

Why this shape

Most teams "monitor" backups in one of three ways: they trust that the backup tool would email them on failure, they glance at a folder once in a while, or they assume green-once-means-green-forever. All three fail the same way. The failure email never arrives because the job died before it could send one. The glance happens the week everything's fine and never the week it isn't. And the assumption is just hope wearing a hard hat — a job that worked in January tells you nothing about June.

The setup above flips the default. Instead of waiting for a backup to announce its own failure, a small system *looks at* each backup every day and treats silence as a problem, not as good news. This is the “dead man’s switch” idea — a check that fires precisely when it *stops* hearing the all-clear. Alerts come with enough context that the owner knows what to fix without digging. They escalate cleanly when the owner is out. And they stop the moment somebody says “fixed.” The sentinel is invisible most days; visible only on the days a backup actually broke.

The next four posts walk through each piece in turn: how a backup job gets registered, how a backup gets checked, how a backup alert reaches its owner, and how a backup status gets cleared once it’s fixed. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 8, 2026 PART 2 OF 7 · BACKUP SENTINEL SERIES ~4 MIN READ

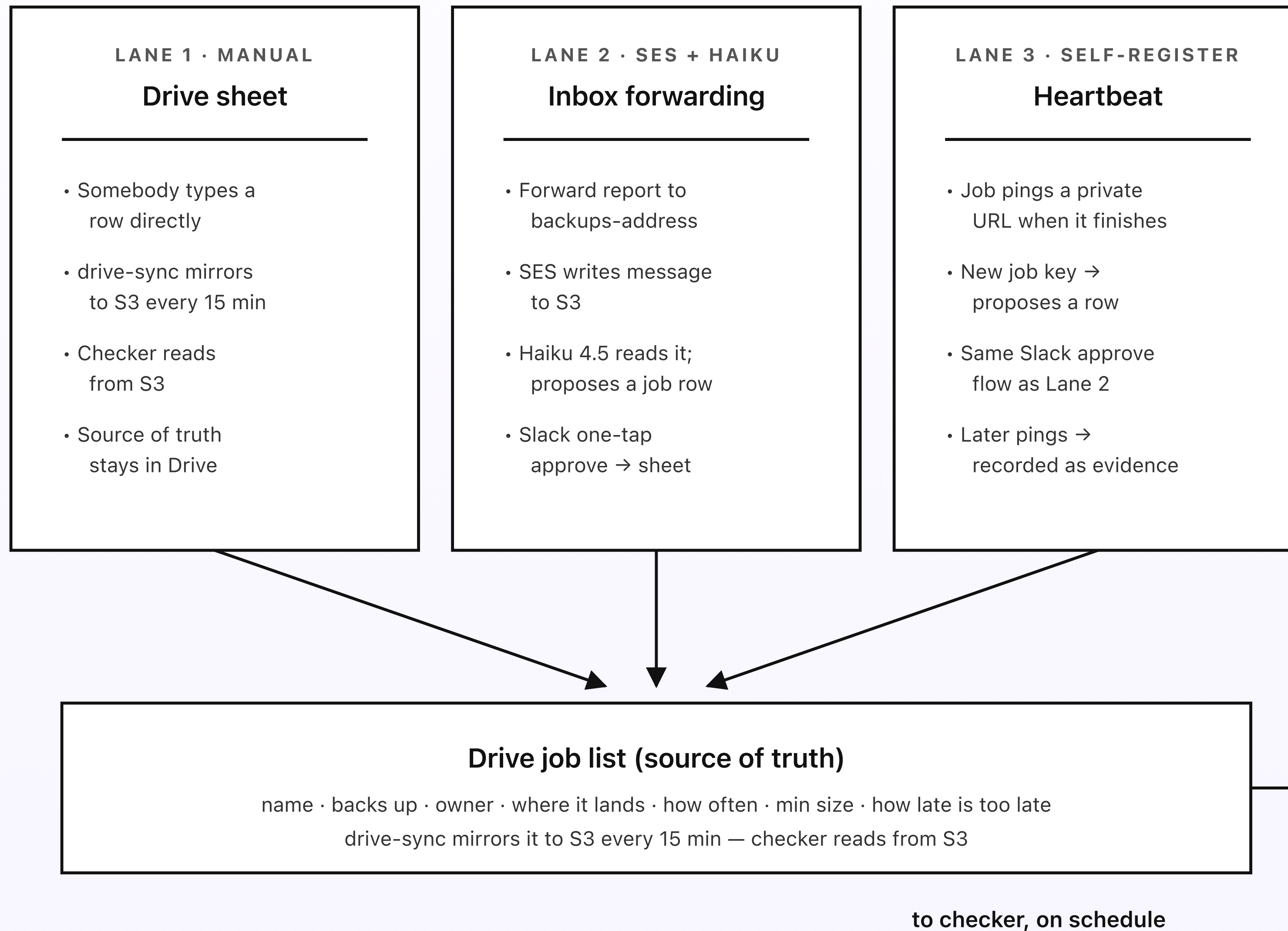
How a backup job gets registered

The sentinel only watches what's on the job list. So the first job is making sure the list actually reflects every backup your business runs. There are three ways a job gets on it: somebody types a row in the Drive sheet, somebody forwards the report their backup tool emails out, or a backup script pings a private web address when it finishes. The first one is obvious. The other two exist because in real life nobody types a row in a sheet for the cron job they wrote two years ago and forgot.

KEY TAKEAWAYS

- Three intake lanes feed one job list: the Drive sheet, an inbox-forwarding lane, and a heartbeat lane.
- Forwarded reports are read by a small parser; Bedrock Haiku 4.5 turns the email into a proposed row.
- Every proposed row goes to the team's Slack for one-tap approval before it lands on the list.
- A backup script can register itself by pinging a private Function URL when it finishes.
- The Drive sheet stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one job list



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

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

Lane 1: the Drive sheet itself

The simplest lane. Open the job-list sheet in Drive, add a row, save. The columns are short: name, what it backs up, owner email, where the backup lands, how often it should run, the smallest size you'd expect, and how late is too late. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://bk-registry-source/jobs.csv` if the sheet has changed since the last sync. The checker 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 a job exists, you know where it lands, and you can spend thirty seconds typing it in. Most existing jobs go in this way during the initial setup.

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

Most backup tools already email a report after each run — "Backup completed, 412 MB" or "Backup FAILED." Set up a dedicated inbound address — something like `backups@your-company.com` — via Amazon SES, and forward those reports to it (or have the tool send them there directly). SES writes the raw message to `s3://bk-raw-mime/`. The S3 write triggers a parser Lambda. The Lambda reads the email body and any attached log.

Then a Bedrock Haiku 4.5 call reads the text and proposes a structured row: job name, what it backs up, where the backup lands, the size it reported, and how often this report seems to arrive. The model prompt is short: “Read this backup report. Propose a job row for the watch list. Return JSON only. Mark each field with a confidence score. Don’t invent a path or size that isn’t in the text.” The output goes to a Slack interactive message that pings the team: 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 form pre-filled with the proposal. On *discard*, the message is logged and the email moved to a discarded prefix in S3 for audit.

The reason every proposed row goes to a human first is simple: a job the model misread — wrong path, wrong size threshold — is worse than a job that never made it onto the list. The misread one will quietly tell you everything is fine while watching the wrong folder.

Lane 3: heartbeat

Some backups are scripts you wrote, not tools you bought. A cron job that runs `pg_dump`, a rsync to a NAS, a nightly export of a cloud drive. For those, the cleanest signal is the job telling the sentinel directly that it finished. Add one line at the end of the script — a single web call to a private address, passing a short job key — and the script now “checks in” every time it completes.

That web call lands on a `heartbeat` Lambda behind a Function URL (a plain web address that runs a Lambda, with no API Gateway in front of it — cheaper and simpler). The first time a new job key checks in, the sentinel doesn’t know it yet, so it proposes a row in the same Slack flow as Lane 2 — one-tap approve to add it.

After that, each heartbeat is recorded as evidence that the job finished, with its timestamp and the size the script reported. The power of this lane is the *absence* of a heartbeat: if a job that usually checks in every night goes quiet, the next check sees no recent heartbeat and flips the job to alert. The job doesn't have to report its own failure — silence is the failure.

Why the job list stays the source of truth

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

Next post: how the checker actually reads each job's evidence, runs its three tests, and picks one of four states.

PART 3 OF 7

JUNE 8, 2026 PART 3 OF 7 · BACKUP SENTINEL SERIES ~5 MIN READ

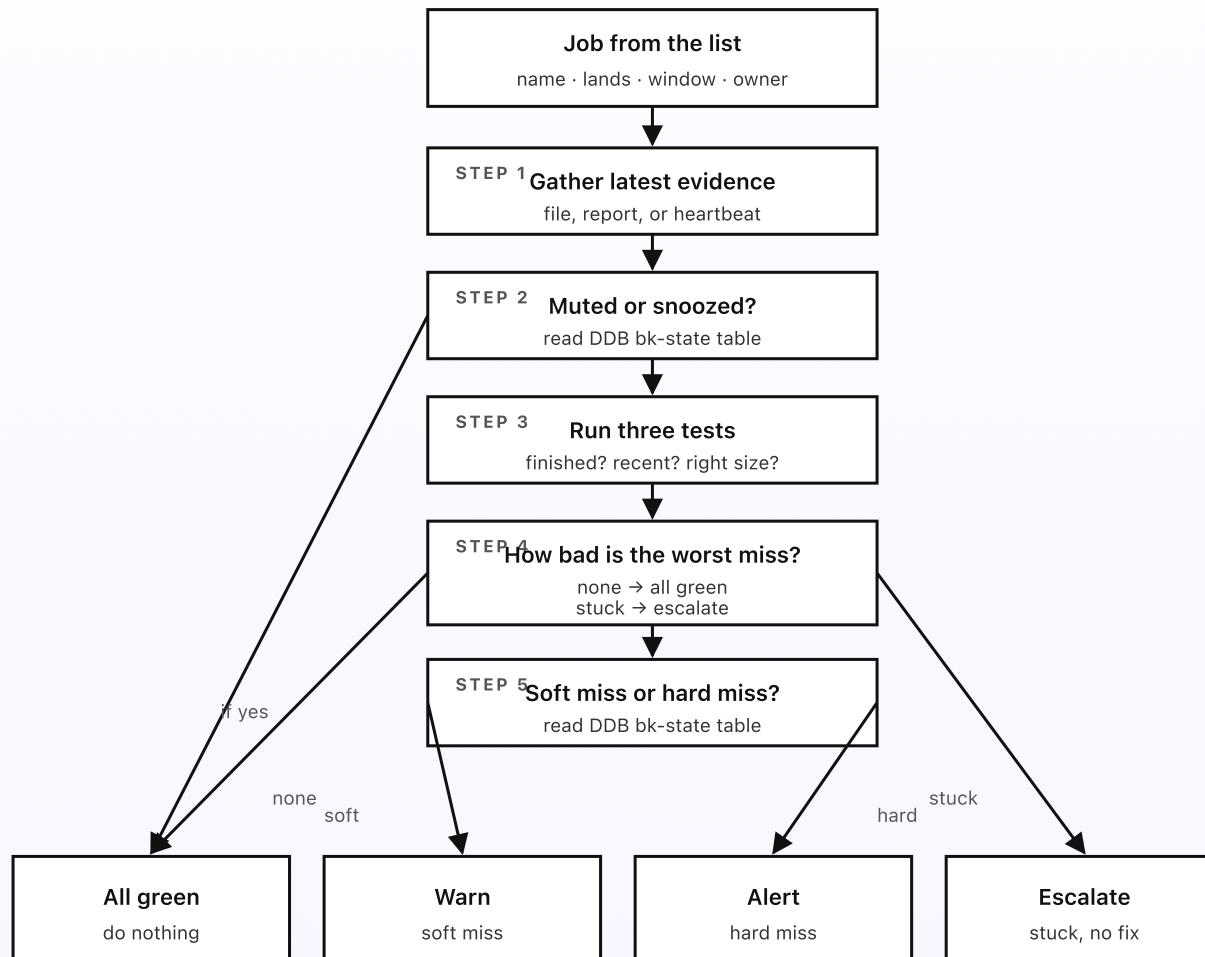
How a backup gets checked

On a schedule — hourly for jobs that run often, a few times a day for the rest — an EventBridge Scheduler rule fires the checker Lambda. The Lambda reads the job list, looks at one row at a time, gathers that job's latest evidence, runs three plain tests, and decides whether to stay quiet or raise something — and if so, how loud. The whole decision is plain Python. No model. No vector search. Every threshold lives in the rules doc, where a rep can edit it without a deploy.

KEY TAKEAWAYS

- The checker runs on a schedule via EventBridge Scheduler — hourly, or a few times a day per job.
- Three tests per job: did it finish, is it recent enough, is it the right size. All thresholds live in the rules doc.
- Four states per job, every check: all green, warn, alert, escalate.
- DynamoDB holds the last state per job so only a real change pings the owner.
- The checker itself never calls a model. The decision is entirely deterministic.

The decision flow, per job



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

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

Three tests: finished, recent, right size

Every job faces the same three questions on every check. They're simple on purpose — the value is in asking them every single day, not in any one being clever.

Did it finish? A backup that started and crashed halfway is not a backup. The checker looks for a clear success signal: a report that says "completed," a marker file the job writes only at the end, or a heartbeat that arrived after the run began. A run that started but never produced an end signal is treated as not finished.

Is it recent? The newest good run has to fall inside the job's freshness window from the rules doc. A nightly dump might allow 26 hours — a little slack past 24 so a job that runs at 2:05am instead of 2:00am doesn't trip. A weekly export might allow 8 days. If the last good run is older than the window, the job is stale, even if it once worked perfectly.

Is it the right size? Two parts. First, the latest backup must be at least the smallest size you'd expect — a 2 KB file where you expect 400 MB means the dump produced almost nothing. Second, it mustn't have shrunk sharply from the run before — a backup that drops from 412 MB to 30 MB overnight usually means a table got dropped or a folder got emptied. The rules doc holds both the minimum size and the "how much shrink is suspicious" percentage (default: more than 50% smaller than the previous good run).

Windows and sizes live in the doc, not the code

The rules doc has one short section per job, or per group of similar jobs. Each section names the thresholds in plain prose: "Orders database: should run every 24 hours, allow up to 26; expect at least 350 MB; flag if it shrinks more than 50%. Shared drive sync: every 24 hours, allow 30; expect at least 5 GB. Weekly accounting export: every 7 days, allow 8; expect at least 10 MB." The checker's code reads these numbers; it doesn't hard-code any of them. Loosen a window or raise a size floor by editing the doc, and the next check uses the new value — no deploy.

Four states, always

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

- **All green.** All three tests pass — finished, recent, right size — or the job is muted or snoozed. Do nothing. Most jobs, most checks, are green.
- **Warn.** A soft miss: slightly late but not yet past a comfortable margin, or a little smaller than usual but above the floor. Send a low-key heads-up so the owner can glance at it. Record the new state in the `bk-state` DynamoDB table.
- **Alert.** A hard miss: missing entirely, well past the freshness window, or far below the size floor / a sharp shrink. Send a clear alert with exactly what failed and a link to the evidence. Record the state.
- **Escalate.** The job was already failing on a previous check and has had a fair chance to be fixed (default: still broken at the next day's first check), and nobody has marked it fixed or snoozed it. Send to the escalation target named

in the rules doc — usually the owner's manager — in addition to the owner. A broken backup that nobody's touching is one of the few cases where pushing harder is the right answer.

State that makes the decision quiet

The checker reads one DynamoDB table every run. `bk-state` holds the current state per job: `(job_id, state, since, last_evidence_ts, last_size)`. With that one table, the decision is a few dozen lines of Python and zero magic. A given job with given evidence and a given window always produces the same state. And crucially, the dispatch in the next post only fires when the state *changes* — green-to-alert pings; alert-to-alert (still broken, not yet escalation time) stays silent. Re-running the check produces no extra pings, because the state in the table already reflects what the owner has seen.

A job that gets fixed is an explicit reset: when the next check sees all three tests pass again, the state flips back to all green and the "cleared" row is written for audit. Part 5 covers what the owner's buttons do to that state in detail.

Why the check uses no model

The checker could call a model to write a smarter alert, or to "judge" whether a missing backup is really a problem. 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 a job must run inside 26 hours and it didn't, the alert fires. A model in that loop introduces variance the team can't reason about, and a backup monitor that sometimes decides a failure is fine is worse than useless. Second, model calls

cost money, and most jobs most days are green, so the call would be wasted nine times out of ten.

Bedrock fires in exactly one place — the daily plain-English summary in Part 6 — turning the day's green/warn/alert states into a calm paragraph. Not on the check. The checker itself is plain Python that reads evidence and writes states.

Next post: how an alert finds the right person, how quiet hours and holidays are honored, and what the owner's buttons actually do.

PART 4 OF 7

JUNE 8, 2026 PART 4 OF 7 · BACKUP SENTINEL SERIES ~5 MIN READ

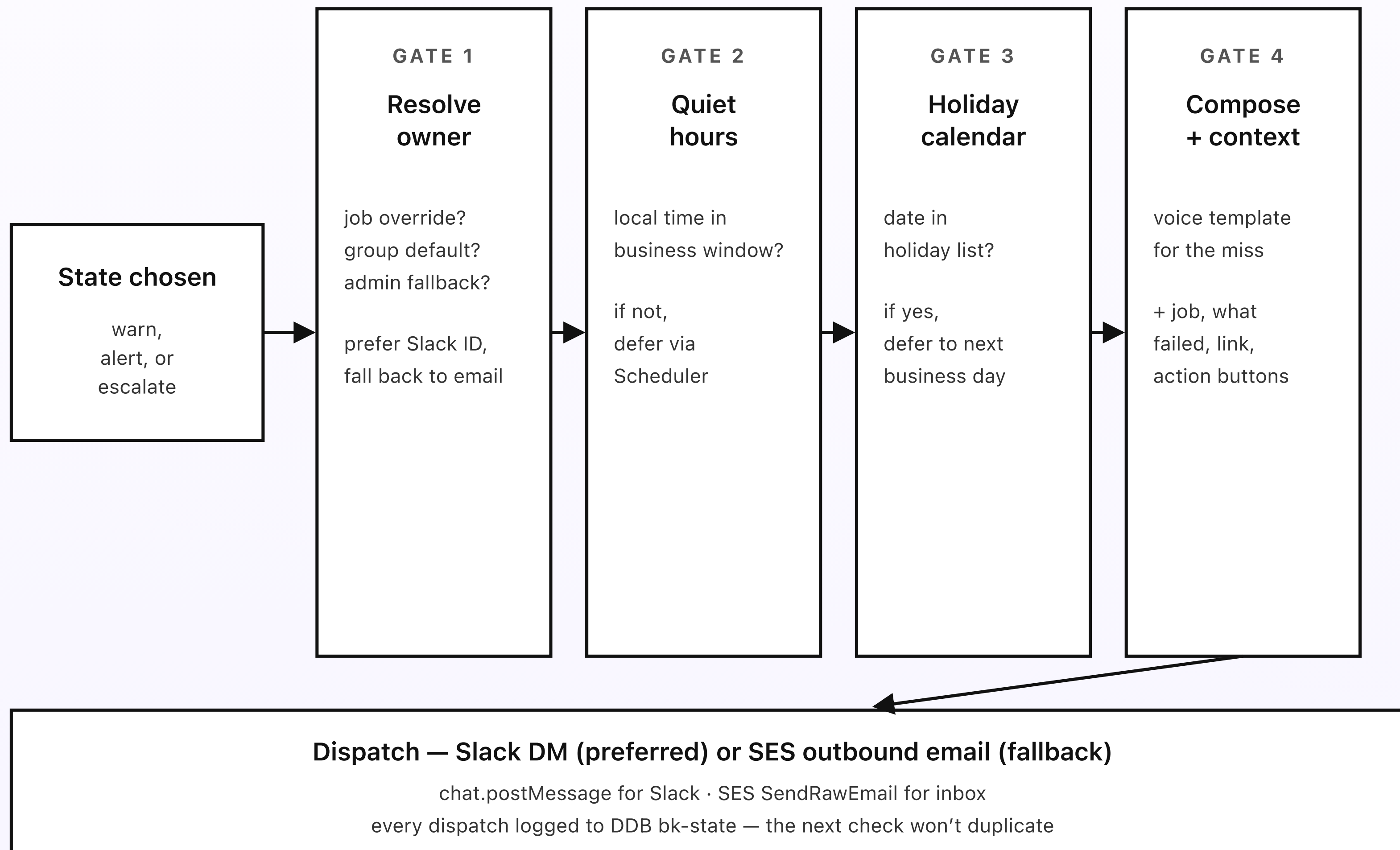
How a backup alert reaches its owner

The checker picked a state — warn, alert, 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 vague "a backup failed," a notification to somebody who left the company three months ago. Four small guardrails sit between the state and the actual ping.

KEY TAKEAWAYS

- Owner resolution: per-job override beats per-group 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 job, exactly what failed, how late or how small, a link to the evidence, and action buttons.
- Escalation pings the named target 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 at 2am.

Fig 4. Four guardrails between the state 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 check doesn't duplicate.

Gate 1: resolve the owner

Three places the dispatch Lambda looks for the owner of a job, in order. First, the job list's per-job `owner_email` column — if a row has a specific person assigned, that person owns it regardless of any group default. Second, the per-group default in the rules doc ("all database backups default to the ops lead"). Third, the configured admin fallback — the person who set up the sentinel and gets every unowned alert. The fallback should never fire in steady state; if it does, the daily summary names every job that hit the fallback so the rules doc can be fixed.

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

A backup usually runs overnight, so a failure is often detected in the small hours — exactly when you don't want a Slack ping waking somebody up for a job that can wait until the morning. 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.

There's one exception, and it's configurable: an *escalate* state can be allowed to override quiet hours for jobs marked critical in the rules doc. A nightly dump of your billing database going dark for two days is the kind of thing some teams do want a 3am ping for. By default, even escalations wait for business hours — the override is opt-in per job.

Gate 3: holiday calendar

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

The list is on purpose — the sentinel 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. (As with quiet hours, a critical-job escalation can be set to ignore the holiday calendar too.)

Gate 4: compose with full context, then ship

The voice doc has one Slack message template per kind of problem — missing, stale, too small, shrank — each with placeholders for the job name, what failed, how late or how small, the last good run, and a link to the evidence. The dispatch Lambda fills the placeholders, attaches *Mark fixed*, *Snooze*, and *Mute* buttons, and ships the message via the Slack API. The bot token itself lives in Secrets Manager.

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

An escalate state adds a second recipient: the escalation target named in the rules doc for that job. The owner is still pinged (the escalation isn't a substitute for the owner's ping — both go out), but the manager now sees it too. The escalate template is slightly different: it includes how long the job has been broken and the previous alert dates, so the manager has the trail at hand.

Every dispatch — Slack or email, owner or escalate — updates the job's row in `bk-state` in DynamoDB. The next check reads that row and knows the owner has already been told, so it won't re-ping the same unchanged failure.

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 3am for something that can wait, skip the day everyone's off, include enough context that the recipient knows exactly what broke without opening a single folder. 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 a backup status gets cleared once the owner has acted — what mark-fixed, snooze, and mute each do to the state, and how the audit trail stays clean.

PART 5 OF 7

JUNE 8, 2026 PART 5 OF 7 · BACKUP SENTINEL SERIES ~5 MIN READ

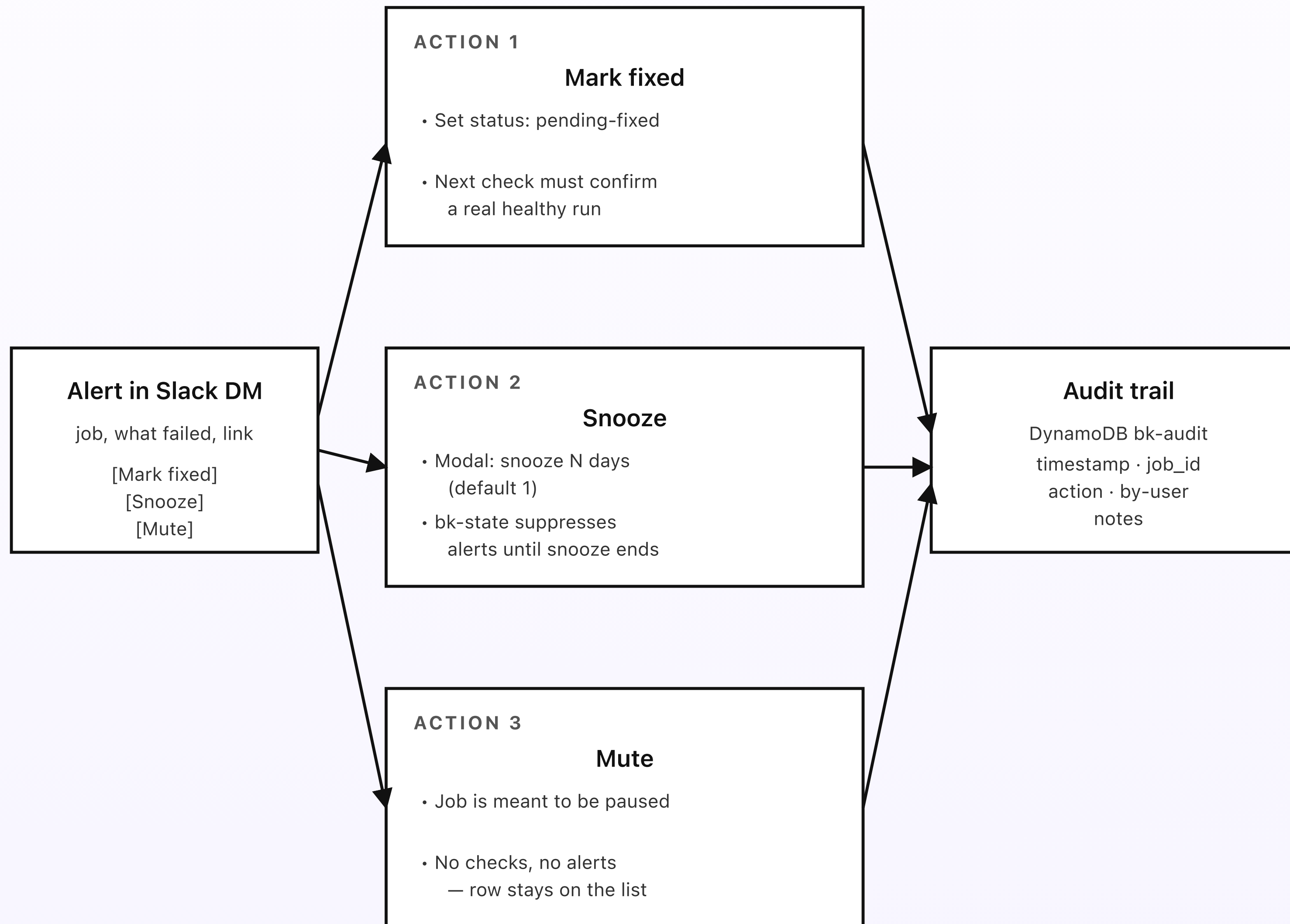
How a backup status gets cleared

An alert lands in Sam's Slack DM at 8:03am. The orders database backup is 30 hours late. There are three buttons: Mark fixed, Snooze, Mute. What happens when he taps one? The honest answer is "it depends on what he actually did." This post walks through the three things the sentinel can do when an owner acts — mark fixed, snooze, mute — and how the job's state and the audit trail stay in sync.

KEY TAKEAWAYS

- Three actions per alert: *mark fixed* (clear, resume checks), *snooze* (quiet while you work), *mute* (job is paused).
- Mark fixed only sticks if the next check actually finds a healthy run — you can't mark a broken job green forever.
- Each action updates the `bk-state` table and writes an audit row.
- Snooze is bounded — you can only snooze a few times before the sentinel escalates anyway.
- The buttons are a Slack interactive message backed by a Function URL.

Three actions on an alert



Mark fixed isn't a magic word — the next check has to confirm a real healthy run.

Fig 5. Three actions per alert, three different effects. Mark fixed clears it pending the next check's confirmation. Snooze quiets the job while you work. Mute pauses a job that's meant to be off. Every action writes to the audit trail.

Action 1: mark fixed (the most common)

Sam cleared the full disk and re-ran the dump. A fresh 415 MB file landed. He taps *Mark fixed*. A Function URL Lambda sets the job's status in `bk-state` to *pending-fixed* and writes a `fixed` row to `bk-audit` with his name and the timestamp. The Slack message updates to "Marked fixed by Sam — confirming on next check."

Here's the important part: *mark fixed* doesn't turn the job green on its own. The very next scheduled check has to actually find a healthy run — finished, recent, right size — before the status flips fully to all green. This is deliberate. If somebody taps the button out of optimism ("I re-ran it, should be fine") but the job is still broken, the next check sees it's still broken and the alert comes right back, now noting "marked fixed at 8:05am but still failing." You can't silence a broken backup by clicking a button — only a real, verified run clears it. That one rule is what keeps the sentinel honest.

Action 2: snooze (the deferral)

Some fixes take longer than the alert wants to wait. The vendor whose tool runs the backup is slow to respond. The disk replacement is on order. The person who knows the script is out until Thursday. Sam isn't able to fix it right now, but he's on it — he just needs the sentinel to be quiet about this one job for a day.

Snooze opens a small modal asking for the number of days, with a 1-day default and a max of 7. On save, a `snooze_until` value is written to the job's row in `bk-state`. The next check reads that in the "muted or snoozed?" step from Part 3 and keeps the job quiet until the snooze ends — but it keeps checking, so the moment the job recovers on its own, the state flips to green without anyone tapping anything. When the snooze ends, the checker re-evaluates — if the job is still broken, the next alert may be an escalation.

Snooze is bounded. The rules doc has a configurable `max_snoozes_per_break` setting (default three). After that many snoozes on the same break, further snooze attempts are rejected with a "You've hit the snooze cap on this job; please fix or escalate" reply, and the next check alerts normally regardless. This exists because the most dangerous failure mode is snoozing a dead backup to nowhere — week after week — until you need it.

Action 3: mute (the "this one is supposed to be off")

Sometimes a job is failing because it's *meant* to be paused. The seasonal store you only back up in Q4. The old database you decommissioned last week but left on the list. The export you turned off on purpose while you migrate. The owner doesn't want to fix it and doesn't want to be reminded — it's correct that it isn't running.

Mute writes a `muted: true` flag to the job's row in `bk-state`. The checker skips muted jobs entirely — no tests, no alerts — but the row stays on the list so it isn't forgotten. A muted job shows up in the daily summary with a small "muted by Sam on 2026-06-10" note, so a paused backup never quietly becomes a *forgotten*

backup. When the season comes around or the migration finishes, the owner un-mutes it and normal checks resume on the next run.

The difference between mute and snooze matters. Snooze says “this should be working, give me time.” Mute says “this is correctly not working, leave it alone.” Keeping them separate means a snoozed job always comes back to bite you if you forget it, while a muted job is an explicit, logged decision you can see in the summary every day.

Every action is logged, every action is reversible

The `bk-audit` table records every mark-fixed, snooze, and mute with the user who took the action, the timestamp, and a snapshot of the job’s state before and after. If a job gets muted by mistake (somebody thought it was decommissioned, it wasn’t), a rep can run an “undo last action” through a small admin command that reads the previous-state snapshot and restores it. The undo is itself an audit row, so the trail of changes stays clean.

This kind of reversibility matters most for the decisions you’ll only think about once. The next time that decommissioned database turns out to have been needed after all, the audit trail is the only record of who muted it and when. A backup monitor that lets things disappear silently is just a quieter version of the problem it’s meant to solve.

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

PART 6 OF 7

JUNE 8, 2026 PART 6 OF 7 · BACKUP SENTINEL SERIES ~3 MIN READ

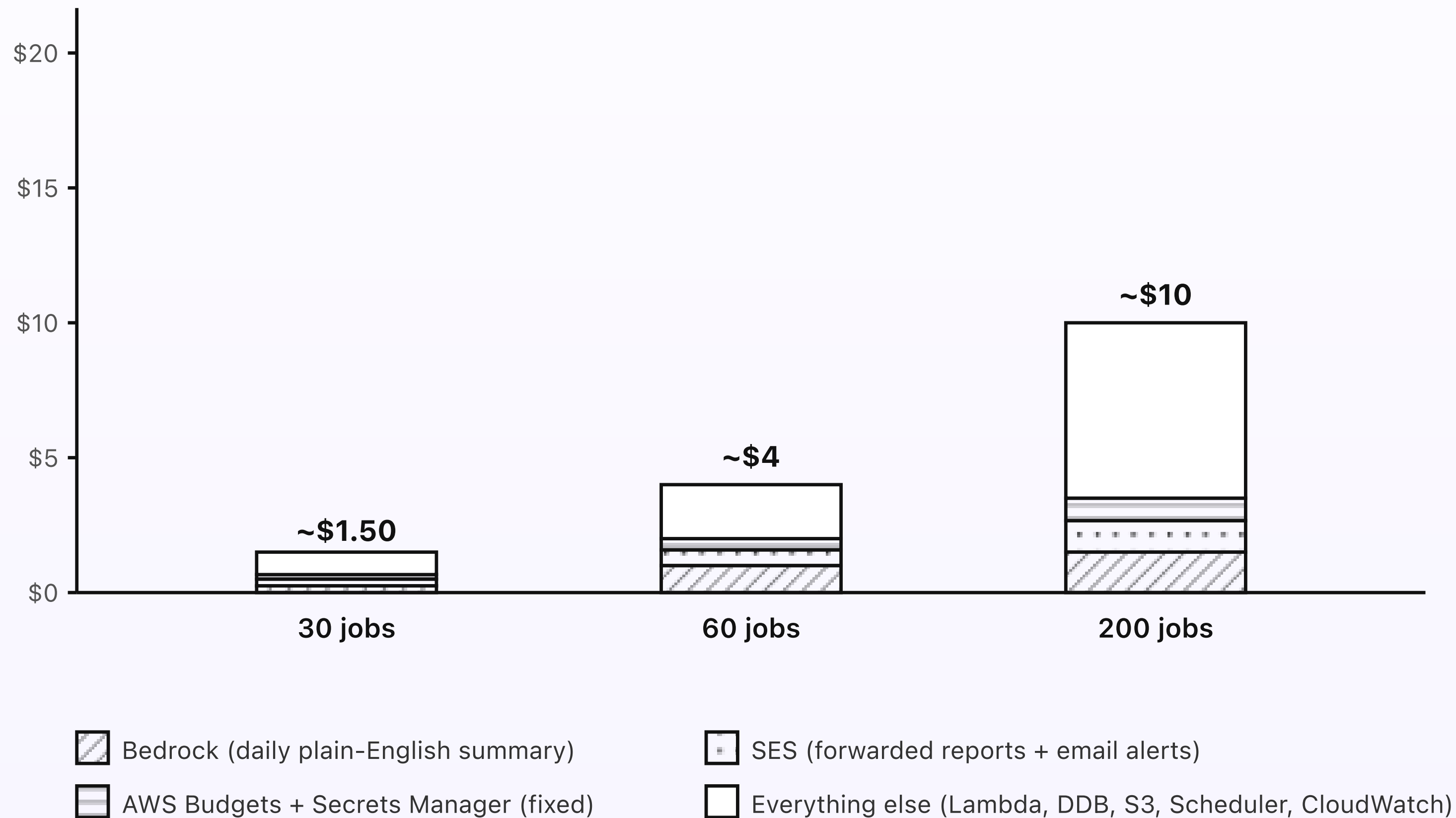
What the backup sentinel costs

The sentinel is one of the cheapest systems in this whole series. Each check reads a small job list from S3, looks at each job's latest evidence, does a little date-and-size arithmetic, writes a few rows to DynamoDB, and posts a message to Slack only when something actually changed. It calls no model on the check. Bedrock fires once a day for the plain-English 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 30 jobs checked a few times a day).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- Each check costs pennies — no model calls.
- Bedrock fires once a day for the summary; that's the only model cost.
- At 60 jobs the bill is around $4. At 200 jobs checked hourly it's around $10.

Cost at three volumes



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

Fig 6. Monthly cost at three job volumes. Bedrock and SES are small slivers because Bedrock only fires for the daily summary and SES only carries forwarded reports and email-fallback alerts. The dominant cost is the everything-else bucket: each scheduled check reading every job.

Where the dollars actually go

Lambda runtime (the bulk). The checker runs on a schedule. Each run reads the job list from S3, and for each job gathers its latest evidence (a small S3 listing, a report, or a heartbeat lookup), computes whether it's finished, recent, and the right size, and decides on a state. At 30 jobs a few times a day, that's a few hundred milliseconds per run. At 200 jobs checked hourly it's a couple of seconds per run, twenty-four times a day. Either way it's pennies a month. Add the dispatch Lambda firing only on state changes, the Function URL Lambda for the buttons and heartbeats, and the drive-sync Lambda every fifteen minutes — the Lambda total still lands under a couple of dollars at all three volumes.

DynamoDB on-demand. Two small tables: `bk-state` and `bk-audit`. Reads are dominant during each check (one read per job per check). Writes are state changes and audit rows. Pennies a month at any of these volumes.

S3 + Storage. The mirrored job-list CSV plus the raw forwarded reports. A few hundred KB total at SMB volume. Effectively free. (The backups themselves live wherever you already keep them — the sentinel only reads listings and sizes, it doesn't copy your data.)

EventBridge Scheduler. The check schedule plus the occasional deferred-dispatch one-off from quiet-hours and holiday gates. A handful of invocations a day. Pennies.

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

Bedrock (only the daily summary). The check uses no Bedrock. Once a day, a single Haiku 4.5 call turns the day's green/warn/alert states into one calm paragraph: a few hundred input tokens (the state list) and a few hundred output tokens. A fraction of a cent per day, so cents a month at any volume. There is no PDF parsing and no Textract in this system — backup reports are short emails, not scanned documents, so a plain model read is all it needs.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the heartbeat and button endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The checker sleeps between runs.
- **A Knowledge Base.** The job list is structured rows, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the check.** The decision on every check is plain Python. Bedrock fires only once a day for the summary.
- **Storing your backups.** The sentinel reads where your backups already live; it never copies or re-stores them, so it adds no backup-storage bill.

How the cost scales

Lambda runtime grows with the number of checks — jobs times how often you check them — because every job is evaluated on every run. DynamoDB grows the same way. Bedrock is flat: one summary a day regardless of how many jobs you

watch. SES is tied to how many reports you forward and how often you fall back to email, not to job count. So the bill at 500 jobs checked hourly is around $22; at 1,000 it's around $45. Past those volumes you'd slow the check cadence for jobs that only run weekly (no point checking an 8-day job every hour), which flattens the curve again — an optimization, not a redesign.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The sentinel's normal-volume bill stays well under that ceiling.

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

PART 7 OF 7

JUNE 8, 2026 PART 7 OF 7 · BACKUP SENTINEL SERIES ~8 MIN READ

Engineering reference: the backup sentinel architecture

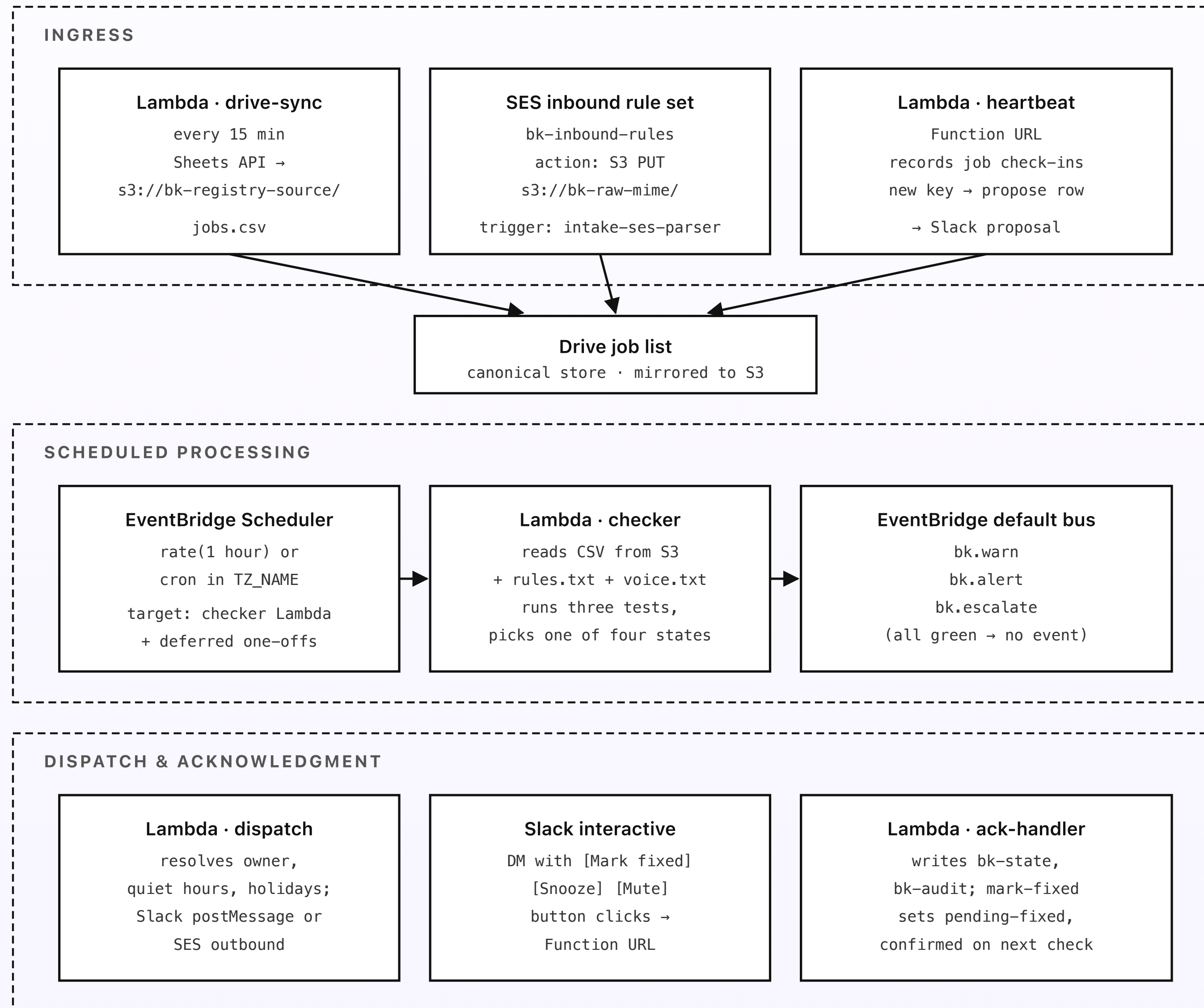
Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, EventBridge Scheduler config, the DynamoDB schemas, and the Slack interactive flow. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound, Bedrock Global cross-Region inference, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is missing a backup failure, not a regional outage. Worth noting the dependency loop: the sentinel watches your backups, so don't host it in the same account or region as the systems it watches — run it in a dedicated account so a blast that takes out a workload doesn't also blind the watcher. One AWS account dedicated to the sentinel keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. Cross-

account read access to the buckets it inspects is granted via narrowly-scoped resource policies, never by running the sentinel inside the watched account.

Topology



It only reads — and every interaction is logged to bk-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the job list), scheduled processing (the checker emitting events on state change), 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 `bk/drive/sa`) to export the job-list sheet as CSV and write to `s3://bk-registry-source/jobs.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://bk-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`heartbeat`** — Lambda Function URL, `AuthType: NONE`; authenticates each check-in with a per-job HMAC key (stored under `bk/heartbeat/keys`) passed in the request, so a leaked URL alone can't forge a heartbeat. Records each check-in to `bk-heartbeats` with `(job_id, ts, reported_size)`. The first time an unknown `job_id` checks in, posts a Slack interactive proposal to register it. Memory: 256 MB. Timeout: 15 s.
- **`intake-ses-parser`** — S3 PUT trigger on `s3://bk-raw-mime/`. Parses MIME, extracts the report body and any attached log 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 job row. Posts the proposal to Slack via `chat.postMessage` with Approve/Edit/Discard buttons. Reports are short text emails — no Textract or OCR is needed; if a backup tool only emits a structured JSON/XML status file, the parser reads it directly and skips the model call. Memory: 256 MB. Timeout: 30 s.

- **`checker`** — EventBridge Scheduler target, on a schedule (default `rate(1 hour)`; jobs with a weekly cadence are evaluated on a slower companion schedule to save cost). Reads `s3://bk-registry-source/jobs.csv` and the rules and voice docs. For each row: gathers evidence (S3 `ListObjectsV2` on the job's target prefix for newest key + size, or latest row from `bk-heartbeats`), runs the three tests, reads last state from `bk-state`, decides on a state. Emits one event per job whose state changed: `bk.warn`, `bk.alert`, or `bk.escalate`, with the job context as the event payload. All-green jobs emit nothing. Memory: 512 MB. Timeout: 120 s. *No Bedrock calls.*
- **`dispatch`** — EventBridge rule on the three state events. Resolves owner, checks quiet hours and holiday calendar (with a per-job `critical` flag that can override both for escalations), formats the alert from the voice template, and ships via Slack `chat.postMessage` (`bk/slack/bot-token` 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. Updates the job's row in `bk-state` 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 (Mark-fixed/Snooze/Mute) and by email-link clicks. Writes to `bk-state`

and `bk-audit`; mark-fixed sets `pending-fixed` (the next `checker` run confirms a real healthy run before it flips to all-green), snooze writes `snooze_until`, mute writes `muted: true`. Memory: 256 MB. Timeout: 15 s.

- **`summary`** — EventBridge Scheduler target, daily at 8am local. Reads the current `bk-state` across all jobs and the past day of `bk-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph "all green / here's what's wrong" narrative, plus a per-job status line; posts to a configured Slack channel and emails via SES. Memory: 512 MB.

Storage

- **DynamoDB · `bk-state`** — one row per job, current state. PK `job_id`; attributes: `state` (all_green/warn/alert/escalate/pending_fixed), `since`, `last_evidence_ts`, `last_size`, `prev_size`, `snooze_until`, `muted`, `last_dispatched_state`. On-demand. No TTL.
- **DynamoDB · `bk-audit`** — one row per write action of any kind (state change, mark-fixed, snooze, mute, register). PK `(job_id, ts)`; attributes: `action`, `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB · `bk-heartbeats`** — one row per heartbeat check-in. PK `job_id`; sort key `ts`; attributes: `reported_size`, `source_ip`. On-demand. TTL at 90 days — the checker only needs the most recent rows.
- **S3 · `bk-registry-source`** — mirrored CSV from the Drive job-list sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.

- **S3 · `bk-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3 · `bk-raw-mime`** — raw inbound MIME from forwarded backup reports. Lifecycle to Glacier at 30 days; expiry at 1 year.
- **Watched targets** — the sentinel only holds `s3:GetObject` + `s3:ListBucket` (read-only) on the buckets where backups land, granted via resource policies on those buckets. It never writes to or deletes from them.

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 proposing a job row from a forwarded report, and `summary` for the daily plain-English narrative. No Sonnet path is justified here — both tasks are short and structured, and Haiku 4.5 handles them well within budget.
- **Embeddings.** Not used. The job list is structured rows; deterministic lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors, no Titan embeddings.
- **Quotas.** Default account quotas are more than enough at SMB volume. The checker itself doesn't call Bedrock; the parsing lane fires a few times a month and the summary once a day.

EventBridge Scheduler config

- `bk-hourly-check` — `rate(1 hour)`. Target: `checker` Lambda (jobs whose cadence is daily or faster).
- `bk-slow-check` — `cron(0 9 * * ? *)` in TZ. Target: `checker` Lambda in slow mode (weekly/monthly jobs only).
- `bk-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `bk-daily-summary` — `cron(0 8 * * ? *)` 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. `backups.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `bk-inbound-rules`: one rule with recipient `backups@your-company.com` → spam scan → S3 PUT to `s3://bk-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the email-fallback alerts and daily summary: verify a sender identity at `sentinel@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:

- **checker role:** `s3:GetObject` on the registry, rules, and voice keys; `s3:GetObject` + `s3:ListBucket` (read-only) on each watched target bucket/prefix; `dynamodb:Query` + `GetItem` + `PutItem` on `bk-state` and `bk-heartbeats`; `events:PutEvents` on the default bus. *No* `bedrock:*`, and *no* write or delete on any watched bucket.
- **dispatch role:** `scheduler:CreateSchedule` for the deferred-dispatch one-offs; `secretsmanager:GetSecretValue` on the Slack bot-token secret; `ses:SendRawEmail` from the verified sender identity; `dynamodb:PutItem` on `bk-state`; outbound network access to `slack.com`.
- **ack-handler role:** `dynamodb:PutItem` + `UpdateItem` on `bk-state` and `bk-audit`; `secretsmanager:GetSecretValue` on the Slack signing-secret; `dynamodb:Query` for state lookup.
- **intake-ses-parser role:** `s3:GetObject` on `bk-raw-mime`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot-token.
- **drive-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the registry and rules buckets; outbound network to `www.googleapis.com`.
- **heartbeat role:** `dynamodb:PutItem` on `bk-heartbeats`; `secretsmanager:GetSecretValue` on the per-job HMAC keys; `secretsmanager:GetSecretValue` on the Slack bot-token for new-job proposals.

Slack interactive flow

Alert messages are posted via the `chat.postMessage` Web API 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` (`mark_fixed`, `snooze`, `mute`), opens a modal if needed (Snooze opens a days modal; Mark-fixed and Mute are one-tap), and processes the response when the modal is submitted. The same handler serves the email-fallback links via a signed query token.

The Slack app needs `chat:write`, `im:write`, and the Interactivity URL configured. The bot token lives in Secrets Manager under `bk/slack/bot-token`. The signing secret is `bk/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:** checker Lambda failures > 0 in a day (the check is the one piece that has to run — a sentinel that silently stops checking is the exact failure it exists to prevent, so this alarm pages directly, not through the sentinel itself); dispatch failure rate > 1% in 24h; ack-handler signature-verification failures > 5/hour (might mean the Slack secret rotated).
- **Self-watch:** the checker emits its own heartbeat to a CloudWatch metric on every run; a metric-absence alarm (no check in 90 minutes) pages the admin independently of the sentinel's own alert path.

- **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 `bk-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive and Sheets APIs live in Secrets Manager under `bk/drive/sa`. Slack bot token and signing secret under `bk/slack/*`. Per-job heartbeat HMAC keys under `bk/heartbeat/keys`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, holiday list reference, quiet-hours window, default shrink threshold, and admin fallback owner all live in Parameter Store under `/bk/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) and AWS SAM. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `bk-registry-source` and `bk-rules-source` so a bad Drive edit can be rolled back in one click, grant the watched-bucket read access via resource policies in a separate stack so a target account's removal can't break the core, and version the EventBridge Scheduler timezone setting so you don't accidentally start checking in UTC after a CI rotation. Total deployable surface: around seven Lambdas, three DDB tables, three S3 buckets owned by the sentinel, 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**.