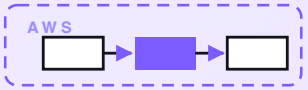
7-PART SERIES · FREE COMPANION



Job status notifier

A repair shop, a trades crew, a framing studio — they all move every job across the same quiet board: received, diagnosing, in progress, ready. Between each step the customer hears nothing and quietly wonders whether anything is happening at all. This is the design of a small serverless notifier that watches that board and, the moment a job changes stage, sends the customer one honest update in the shop's own voice — with a photo of the actual work and a real ETA for the next stage. A daily sweep chases any job that has sat too long in one place, so nobody is ever left guessing. It never invents a stage and never sends in the middle of the night. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Job status notifier

PART 1 OF 7

JUNE 25, 2026 PART 1 OF 7 · JOB STATUS NOTIFIER SERIES ~10 MIN READ

A job status notifier on AWS for a few dollars a month

Every service business runs the same quiet board: jobs come in, move through a few stages, and go out. The customer who handed over a cracked phone, a bike, or a suit to be altered hears nothing in between, and starts to wonder. This post walks through the design of a small notifier that watches that board and, the moment a job moves, tells the customer where it is — in the shop's voice, with a photo and an honest ETA — and chases anything that goes quiet.

KEY TAKEAWAYS

- The shop already moves each job across a board — received, diagnosing, in progress, ready. That board is the only trigger.
- Every genuine stage change sends the customer one update by SMS or email, in the shop's voice, with a photo and an honest ETA.
- The stage facts and the ETA come from the job record; one Bedrock call only phrases them. It never invents a stage.
- A daily sweep chases any job that has sat too long in one stage, so a customer is never left guessing.
- Designed on AWS for about $2.30/month at roughly 200 jobs a month. Quiet hours mean it never texts at 2am.

The whole system on one page

Before any code, here's the shape of what we're designing. A service business — a phone-repair counter, a bike workshop, a tailoring studio — takes a job in, moves it through a handful of stages, and hands it back. The customer who dropped off a cracked phone on Monday hears nothing until it's done, and spends the week quietly wondering whether anyone has even looked at it. The system below closes that gap: it watches the board the shop already keeps, and every time a job moves, it tells the customer where their job is and when to expect the next step. Nothing more.

This is deliberately *not* a parcel tracker. There is no courier and no tracking number. The thing being followed is work happening on a bench, and the only source of truth about it is the shop's own board.

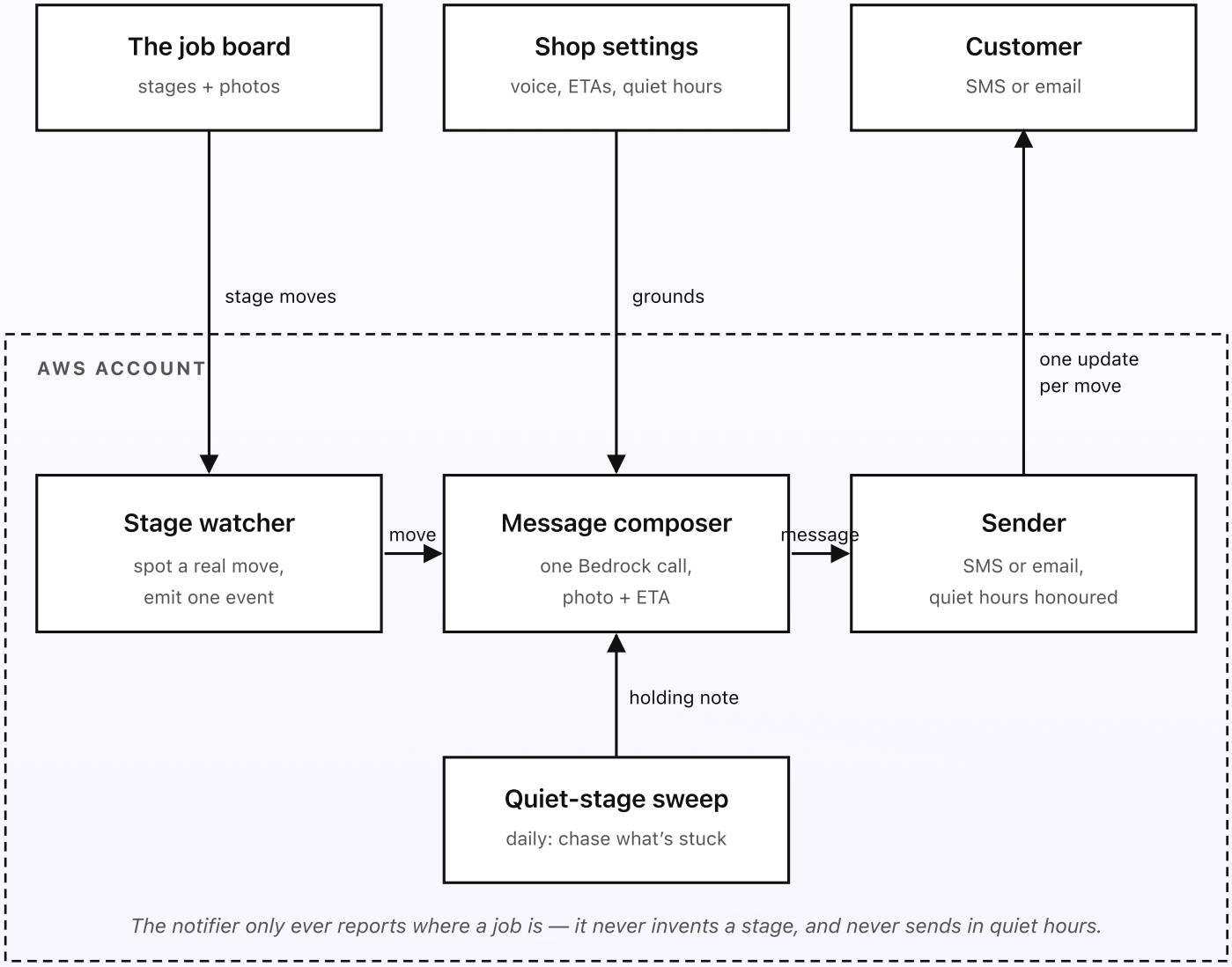

The job board
stages + photos
Shop settings
voice, ETAs, quiet hours
Customer
SMS or email
stage moves
grounds
AWS ACCOUNT
one update
per move
Stage watcher
spot a real move,
emit one event
move
Message composer
one Bedrock call,
photo + ETA
message
Sender
SMS or email,
quiet hours honoured
holding note
Quiet-stage sweep
daily: chase what's stuck
The notifier only ever reports where a job is — it never invents a stage, and never sends in quiet hours.

Fig 1. Two sources outside, three pieces and a sweep inside AWS. The job board is the only trigger. The Stage watcher spots a real move, the Message composer writes one update with a photo and an ETA, and the Sender delivers it by SMS or email. A daily sweep chases anything stuck.

What you set up once (the outside)

- **The job board.** Wherever the shop already tracks work — a Google Sheet with one row per job, or the board view of whatever job system the shop uses. Each row has a job id, the customer's name and contact, what the job is ("iPhone 13, cracked screen"), the customer's chosen channel, and the one field that matters most: the current stage. The shop moves a job by changing that stage, exactly as they do now. They can also drop a photo of the work against the job — the screen mid-repair, the finished hem — covered in Part 4. The board is the single trigger for everything downstream; if it doesn't move, nothing is sent.
- **Shop settings.** One short doc in the same place. It holds the things you'll want to change without a deploy: the shop's name and the voice for messages (warm and brief, or formal), the list of valid stages in order, the typical duration of each stage (used to work out an honest ETA for the next one), the quiet hours during which nothing should send, and the dwell threshold per stage that decides when a quiet job gets chased. Change a stage name or a quiet-hours window here and the system follows; no code changes.
- **The customer.** Nothing for them to install. When the job is booked in, they give a mobile number or an email and pick how they'd like updates. From then on they get one short, friendly message each time their job genuinely moves — and a holding note if it ever sits still too long — on the channel they chose.

What runs on every move (the inside)

- **The stage watcher.** A small Lambda polls the board every few minutes and compares each job's current stage against the last stage it recorded. A genuine move — a known stage, a step forward, not a duplicate — gets written to a stage-history table and emitted as one `stage.changed` event. A typo, a sideways correction, or the same move seen twice is ignored. This is Part 2.
- **The message composer.** Triggered by that event. It gathers the facts: which stage the job moved to, the honest ETA for the next stage from the settings, and the photo if there is one. One Bedrock Haiku 4.5 call turns those facts into a single message in the shop's voice — "your repair is now in progress, ready by Thursday." The model is handed the facts and told to use only those; it never decides the stage or the ETA. This is Part 3 (and Part 4 for the photo and the ETA).
- **The sender.** Delivers the finished message on the customer's channel — SMS through SNS, or email through SES — and records what went out. If the message would land inside quiet hours, it's held until morning. Every update is de-duplicated against what was already sent, so the same news never goes twice. The daily quiet-stage sweep that chases stalled jobs is Part 5.

In plain words

Maya drops her cracked iPhone at the repair counter on Monday. It's booked as job J-2042 and she picks text updates. Tuesday morning the technician moves J-2042 from *received* to *in progress* and snaps a photo of the opened phone. Within a few minutes Maya gets a text: "Hi Maya — good news, we've started on your iPhone screen repair. We expect it ready to collect by Thursday afternoon. Here's

a quick look: [photo]. — The Repair Bench." Thursday lunchtime the job moves to *ready* and she gets a second text: "Your iPhone is repaired and ready to collect any time before 6pm. See you soon!" Two messages, both true, both in the shop's voice, and nobody at the counter had to stop and write them.

Now the version that goes wrong without this. A different job, J-2051, goes into *awaiting parts* on Tuesday and the part is back-ordered. Nobody thinks to tell the customer, who turns up Friday expecting a finished job and leaves annoyed. With the sweep running, on Thursday the system notices J-2051 has sat in one stage for three days, sends the customer an honest holding note — "your repair is waiting on a part; we'll update you the moment it arrives" — and flags it to the shop. The customer is told before they have to ask, and the job stops being forgotten.

DESIGN RULES THAT SHAPED EVERY DECISION

- The board is the only trigger. The notifier reports the shop's own stages and never gets ahead of them.
- One genuine move, one message. Forward changes only — never a typo, a sideways edit, or a duplicate.
- Facts come from the record, words come from the model. Bedrock phrases the update; it never decides the stage or the ETA.
- Honest ETAs only. A "ready by" is shown only when the shop has set one, with a buffer — never a wishful guess.
- Never go quiet on a customer. A job stuck too long in one stage gets a holding note before they have to chase.
- Respect the clock. Quiet hours hold overnight messages until morning; no one is texted at 2am.

Why this shape

Most small shops keep customers informed one of three ways: someone remembers to call, the customer phones to ask, or nobody says anything until the job is done. Remembering to call is the first thing that slips on a busy day. Waiting for the customer to ask turns every job into an interruption — and a customer who has to chase is already half-unhappy. And saying nothing until the end means the one time something genuinely slips, the customer finds out by turning up to a job that isn't ready.

The shape above keeps the board the shop already maintains as the single source of truth, and adds a small system that turns each move on that board into one honest message — with a photo that proves work is really happening and an ETA the customer can plan around. The dull, easy-to-forget updates send themselves; the few jobs that genuinely stall get caught and explained before they become a complaint. And because the model only ever phrases facts it's handed, the customer never gets told their job is "ready" when the board says it isn't.

The next four posts walk through each piece in turn: how a stage change gets detected and trusted, how a customer update gets composed, how photos and ETAs get attached, and how a quiet stage gets chased. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 25, 2026 PART 2 OF 7 · JOB STATUS NOTIFIER SERIES ~8 MIN READ

How a job stage gets updated

Before a customer can be told their job moved, the system has to be sure it actually did. This post is about that step alone: how a quiet poll of the board turns “the stage cell changed” into one trustworthy stage-change event — and how it refuses to fire on a typo, a sideways edit, or the same move seen twice.

KEY TAKEAWAYS

- A small watcher polls the board every few minutes and compares each job’s stage against the last one it recorded.
- A change only counts if it’s a known stage and a step forward — not a typo, a sideways edit, or a move backwards.
- The same poll seen twice never fires twice; the last-recorded stage is the dedupe key.
- A genuine move is appended to a stage-history table, then emitted as one `stage.changed` event on a custom bus.
- Detecting the move and sending the message are split, so a slow message can never hold up watching the board.

Watching a board that wasn't built for this

The job board is whatever the shop already uses — usually a spreadsheet or the board view of a job tool. It was built for people to look at, not for a machine to listen to, and that's the whole challenge of this step. There's no neat "a job moved" signal to subscribe to. All the system has is a grid of jobs with a stage written in a cell, and that cell quietly changing whenever someone at the counter drags a card or edits a row.

So the watcher does the only reliable thing: it looks at the board on a schedule and remembers what it saw last time. A small Lambda, `jsnr-stage`, runs every few minutes on an EventBridge Scheduler rule, reads the current stage of every open job, and compares each one against the stage it recorded in the `jsnr-jobs` table on its previous pass. Most passes find nothing changed and do nothing. When a job's stage no longer matches what was recorded, that job is a candidate for a move — but a candidate is not yet a message.

When a change actually counts

A cell changing is not the same as a job moving forward. People fix typos, correct a mistaken drag, or relabel a stage. If every edit fired a customer text, the system would be worse than useless — it would be a liar that texts "your job is ready!" because someone fat-fingered the wrong row. So a candidate change has to clear three checks before it counts:

- **Is it a known stage?** The new value has to be one of the valid stages listed in the shop settings — *received*, *diagnosing*, *in progress*, *awaiting parts*, *ready*,

collected. A stray value ("in pgoress", a blank, a note someone typed in the wrong cell) is not a stage. It's logged and ignored, and nothing is sent.

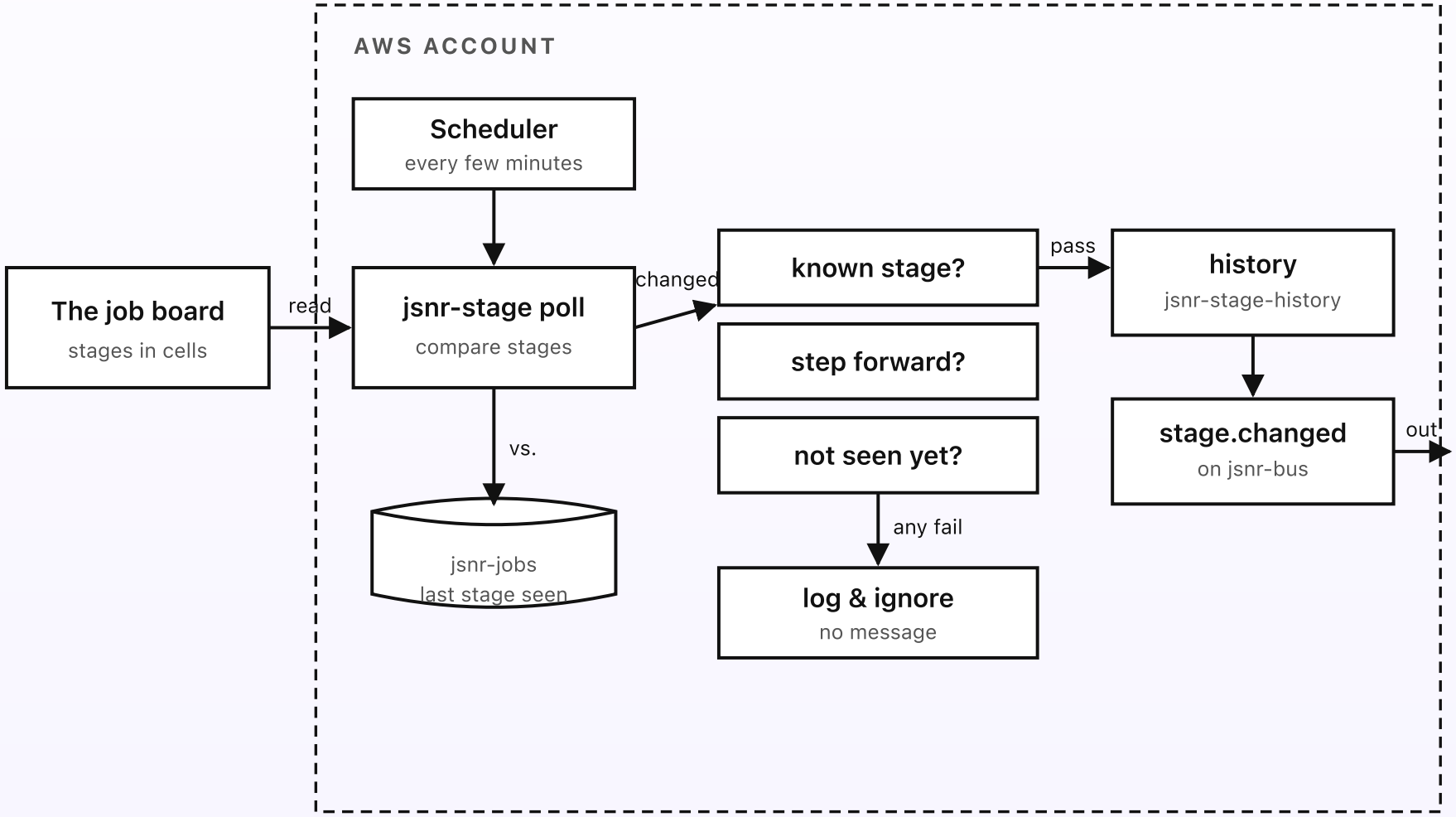
- **Is it a step forward?** Stages have an order. A move from *received* to *in progress* is forward and notifiable; a move from *ready* back to *in progress* is a correction, not news for the customer. Backward and sideways moves are recorded in the history for the shop's own record but don't send a message — nobody wants a text saying their finished job is somehow unfinished again.
- **Have we already seen it?** Because the watcher polls, the same new stage will be sitting there on the next pass too. The last-recorded stage in `jsnr-jobs` is the dedupe key: once a move to *in progress* has been recorded and emitted, the stage *is in progress* as far as the watcher is concerned, so the next poll sees no change and stays quiet. One move produces exactly one event.

Only a candidate that is a known stage, a step forward, and genuinely new survives all three. That is what the rest of the system treats as "the job moved."

Record first, then announce

When a change counts, the watcher does two things in order. First it writes the move to `jsnr-stage-history` — an append-only row keyed by job and timestamp, capturing the from-stage, the to-stage, and when it happened — and updates the last-recorded stage on the job. That write is the system's memory; it's what makes the dedupe work and gives the shop an honest timeline of every job later. Only then does it put a single `stage.changed` event onto a custom EventBridge bus, `jsnr-bus`, carrying the job id, the new stage, and the timestamp.

Recording before announcing matters. If the history write succeeds but the event somehow fails, the worst case is a missed message, not a wrong one — and the next poll won't re-fire, because the stage is already recorded as moved. The system would rather stay quiet than tell a customer something that isn't backed by a recorded fact.



A cell changing is not a job moving — only a known, forward, not-yet-seen change becomes one event.

Fig 2. The watcher polls the board, compares each stage against the last one it recorded, and only a known, forward, not-yet-seen change survives. The move is written to history, then emitted as a single `stage.changed` event.

Why a poll, and why an event

Two design choices in this step are worth defending. The first is polling rather than waiting for the board to push. Spreadsheets and most lightweight job tools simply don't offer a reliable "tell me when a cell changes" hook, and the ones that do are fiddly and break quietly. A poll every few minutes is boring, cheap, and robust: if a poll is missed, the next one catches up, because the truth is always sitting in the board. A few minutes' delay between a counter staff member moving a card and the customer getting a text is completely fine for this job — nobody is timing it to the second.

The second is splitting detection from sending by putting an event on a bus rather than calling the composer directly. The watcher's job is to watch the board and get out of the way; composing and sending a message — which involves a model call and a network send — is slower and can fail. By emitting a `stage.changed` event and letting a rule route it onward (through a queue, in Part 3), a slow or failed message never blocks the next poll, and a message that fails can be retried without re-reading the board. The watcher stays a fast, simple sensor; everything that might be slow lives downstream.

DESIGN RULES THAT SHAPED STAGE DETECTION

- The board is read, never trusted blindly. A changed cell is a candidate, not a confirmed move.
- Three checks, every time. Known stage, step forward, not already seen — or it doesn't fire.
- The last-recorded stage is the dedupe key. One move produces exactly one event, however often we poll.
- Record before you announce. History is written first; the worst failure is a missed message, never a false one.
- Detection and sending are separate. The watcher emits an event; nothing slow gets to hold up the board.

PART 3 OF 7

JUNE 25, 2026 PART 3 OF 7 · JOB STATUS NOTIFIER SERIES ~7 MIN READ

How a customer update gets composed

By the time this step runs, the system knows a job has genuinely moved to a new stage. All that is left is to say it like the shop would. This post is about the only place a model is allowed near the system: the single Bedrock call that turns real stage facts into one warm, accurate message — and the fences that keep it honest.

KEY TAKEAWAYS

- The `stage.changed` event is routed through an SQS queue to the composer, so a slow model call never blocks the watcher.
- The composer gathers the facts first — the new stage, the next-stage ETA, the photo — before any model runs.
- One Bedrock Haiku 4.5 call turns those facts into a single message in the shop's voice. The model only phrases; it never decides.
- The draft is checked against the facts: it must name the real stage, and it can't claim an ETA the facts didn't carry.
- If anything is missing or the draft fails the check, the system sends a plain templated line rather than something invented.

From an event to a set of facts

A rule on the bus matches every `stage.changed` event and drops it onto an SQS queue, `jsnr-notify`, which triggers the composer, `jsnr-composer`. The queue is doing quiet but important work: it absorbs a flurry of moves at opening time without dropping any, and if a message fails it can be retried — after a few attempts a poison message lands in a dead-letter queue rather than spinning forever. The watcher from Part 2 has already moved on; composing happens on its own time.

The first thing the composer does is *not* call a model. It gathers facts. From the event it has the job id and the new stage. From `jsnr-jobs` it pulls the customer's name, what the job is, and their channel. From the shop settings it works out the honest ETA for the *next* stage — the mechanics of that are Part 4. And it checks the photos bucket for a picture attached to this job at this stage. That bundle — stage, item, name, ETA, photo — is the complete, factual basis for the message. Everything in the text that follows has to come from this bundle and nowhere else.

The one place a model runs

Composing the wording is the only place in the whole system a model is allowed near. It's a good fit for one: the difference between "Status: IN_PROGRESS" and "Good news — we've started on your repair and expect it ready Thursday" is exactly the warmth a small shop wants and a template can't fake. So `jsnr-composer` makes a single Bedrock Haiku 4.5 call, handing the model the gathered facts and the shop's voice from settings, with strict instructions: use only these

facts, name this exact stage, mention the ETA only if one is given, keep it to a sentence or two, and write it the way this shop talks.

The crucial line in that prompt is the fence: the model is told what stage the job is in and is forbidden from changing it. It does not get to decide that “in progress” really means “nearly done”, or to promise a day the settings didn’t supply. It is a writer handed the facts, not a clerk deciding them. The stage came from the board; the ETA came from the settings; the model just makes them sound human.

Checking the draft before it leaves

A model handed strict facts is still a model, so the draft doesn’t go out unread. The composer runs the returned text through a couple of plain checks before passing it to the sender. It confirms the message refers to the stage the job actually moved to — a draft that somehow talks about being “ready” when the stage is “in progress” is rejected. And it confirms the message doesn’t assert a collection time or date unless an ETA was in the facts; if no ETA was supplied, the message must not invent one.

If the draft passes, it’s the message. If it fails either check, or if the model call itself errors or times out, the composer falls back to a plain templated line built straight from the facts — “Your iPhone screen repair is now in progress. We’ll let you know when it’s ready.” It’s less warm, but it is always true. The system would rather send something flat and correct than risk something charming and wrong. A customer is never the test subject for a model’s bad day.

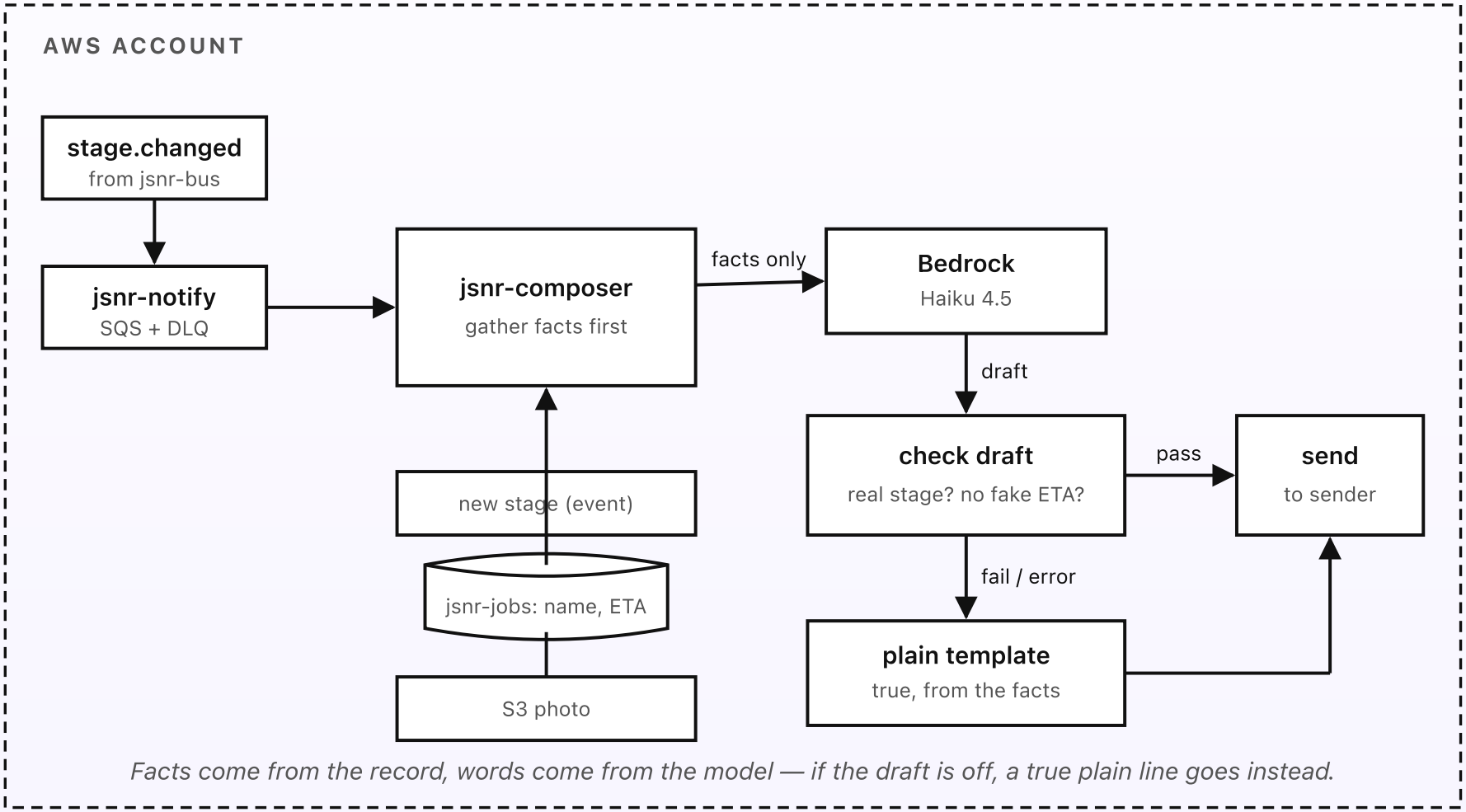

AWS ACCOUNT
stage.changed
from jsnr-bus
jsnr-notify
SQS + DLQ
jsnr-composer
gather facts first
facts only
Bedrock
Haiku 4.5
draft
check draft
real stage? no fake ETA?
pass
send
to sender
fail / error
plain template
true, from the facts
new stage (event)
jsnr-jobs: name, ETA
S3 photo
Facts come from the record, words come from the model — if the draft is off, a true plain line goes instead.

Fig 3. The composer gathers the facts before any model runs, makes one Bedrock call to phrase them in the shop's voice, then checks the draft names the real stage and invents no ETA. A failed check or a model error falls back to a plain, true line.

What a composed message looks like

The same facts, the same shop, two different moves. When J-2042 reaches *in progress* with a Thursday ETA and a photo, the composer sends: "Hi Maya — we've started on your iPhone screen repair and expect it ready to collect by Thursday afternoon. Here's a quick look at the work: [photo]. Any questions, just reply. — The Repair Bench." When it reaches *ready*, with no further stage to estimate: "Great news, Maya — your iPhone is repaired and ready to collect any time before 6pm today. See you soon!" Both are warm, both name the real stage, and the second one carries no invented "next" date because there isn't one. That restraint — saying only what's true — is the whole point of feeding the model facts and checking what it gives back.

Because the wording, the voice, and the rules all live in the prompt and the settings rather than in code, the shop can change how it sounds without a deploy. Switch the voice from breezy to formal, add a sign-off, ask for emoji or ban them — it's an edit to the settings doc, and the next message follows. The facts are locked down; the personality is free.

DESIGN RULES THAT SHAPED THE COMPOSER

- Gather facts before calling the model. The message is built on the record, not on the model's imagination.
- The model phrases, it never decides. Stage and ETA are handed in; the model only makes them sound human.
- Name the real stage, invent no ETA. The draft is checked against the facts before it can leave.
- A true plain line beats a charming wrong one. Any failure falls back to a template built from the facts.
- Voice lives in settings. How the shop sounds changes without a deploy; what's true never bends.

PART 4 OF 7

JUNE 25, 2026 PART 4 OF 7 · JOB STATUS NOTIFIER SERIES ~7 MIN READ

How photos and ETAs get attached

An update that just says “in progress” is barely better than silence. What turns it into something a customer trusts is a photo of their actual job and a straight answer to “when?” This post is about those two attachments: how a snap taken at the bench reaches the right job, and how an honest ETA for the next stage is worked out rather than guessed.

KEY TAKEAWAYS

- A photo taken at the bench lands in S3 under a key built from the job id and the stage, so it attaches itself to the right update.
- A small handler resizes the photo and links it to the job; an oversized or unreadable file is dropped, never sent broken.
- Photos are optional. A move with no photo still sends a perfectly good update — the picture is a bonus, not a blocker.
- The ETA is computed, not guessed: the next stage’s typical duration from settings, plus an honest buffer, rounded to a friendly time.
- If the shop hasn’t set a duration for the next stage, no ETA is shown — better silent than wrong on a promise.

Why a photo and a time matter so much

An update that only says “your job is in progress” is honest but thin. Two things turn it into something a customer actually trusts. The first is a photo: a picture of their own phone opened on the bench, their bike on the stand, their suit pinned for alteration. It’s proof that real work is happening to their real thing, and it does more for confidence than any wording. The second is a straight answer to the only question the customer really has: *when?* “In progress” without a when still leaves them guessing. This post is about those two attachments — where the photo comes from, and how the ETA is worked out so it’s honest rather than hopeful.

How a photo reaches the right job

The shop already takes photos — most benches have a phone within reach. The trick is getting a snap to the right job without anyone doing fiddly admin. The photo is dropped against the job: uploaded from the board row, or sent to a shared folder named for the job. However it arrives, it lands in the `jsnr-photos` S3 bucket under a key built from the job id and the current stage — something like `J-2042/in-progress.jpg`. That key is the whole linking mechanism. When the composer gathers facts for a move to *in progress* on J-2042, it looks for exactly that key, and if a photo is there, it’s the one attached to this update.

A small handler, `jsnr-photo`, tidies each upload as it lands: it confirms the file really is an image, resizes it down to something sensible for a text or email (a phone photo can be several megabytes; nobody wants that arriving as an MMS), strips location data, and records the link on the job. A file that isn’t a valid image, or is absurdly large, is dropped with a log line rather than passed on — a broken

attachment is worse than none. The bucket keeps versioning on, so a replaced photo doesn't silently lose the old one, and a lifecycle rule clears photos a while after the job is collected, since there's no reason to store a stranger's repair photos forever.

Crucially, photos are optional. If a move happens and no photo was dropped, the update sends anyway with no gap and no apology — the composer simply doesn't mention a picture. The photo enriches the message when it's there; it never holds one up when it isn't.

How an honest ETA is built

The ETA is the part most worth getting right, because a missed promise does more damage than no promise at all. So it's computed from facts, never guessed by the model. The shop settings carry a typical duration for each stage — *diagnosing* takes about half a day, *in progress* about two days, *awaiting parts* depends on the supplier. When a job moves into a stage, the composer looks up how long the *next* step typically takes, adds that to now, applies an honest buffer so the shop is more likely to beat the estimate than miss it, and rounds the result to a friendly point — "by Thursday afternoon", not "at 15:42 on Thursday".

A few rules keep it honest. The buffer always pushes the estimate later, never earlier — it's better to say Thursday and be ready Wednesday than the reverse. The estimate respects opening hours, so a job that finishes Friday evening reads "ready Saturday morning" rather than implying a midnight collection. And the load-bearing rule: if the settings have *no* duration for the next stage — or there is no next stage, because the job just reached *ready* — then no ETA is produced, and

the composer is told there isn't one. As Part 3 covered, the model is forbidden from inventing a time the facts didn't carry. A missing ETA means a quieter message, not a made-up one.

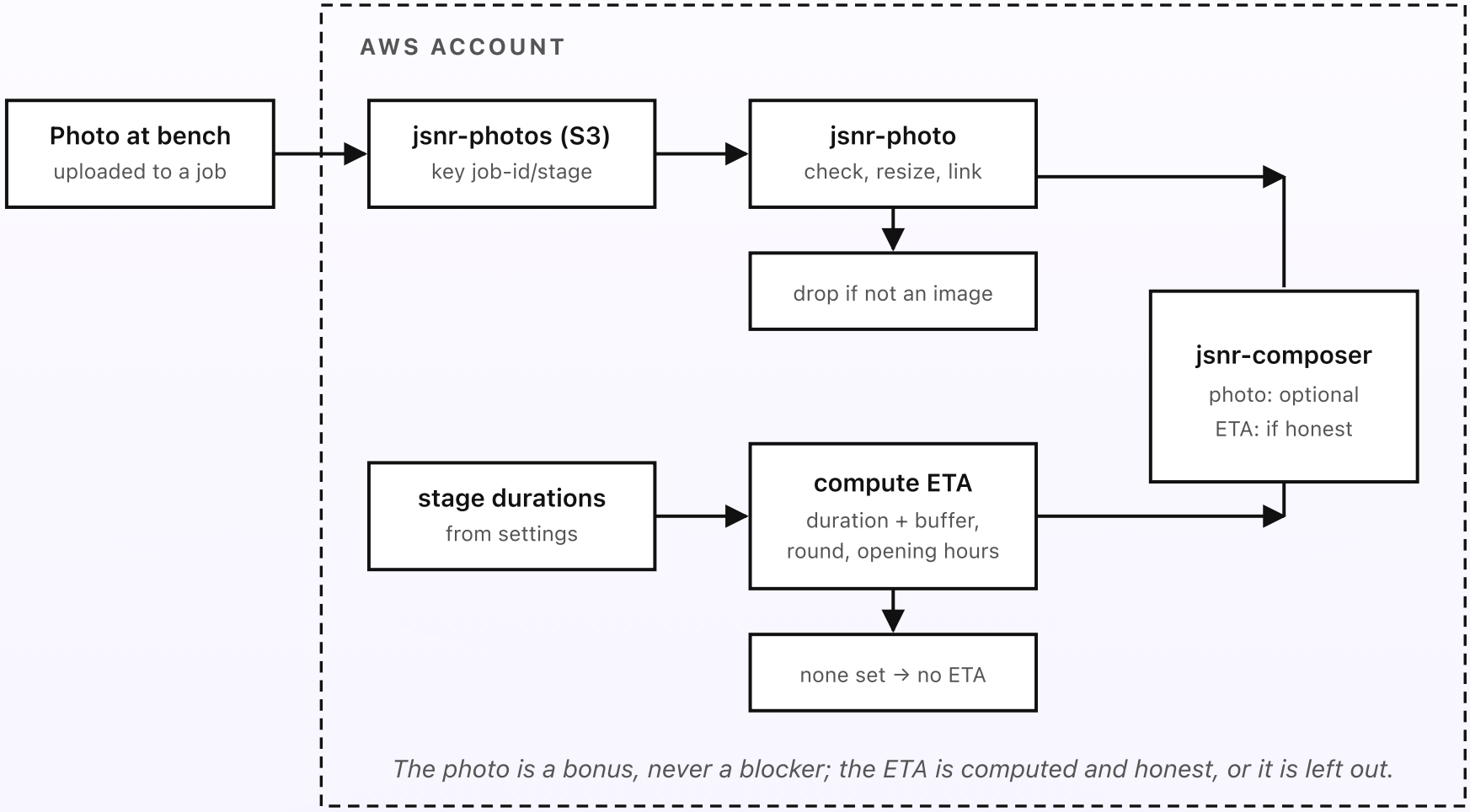

AWS ACCOUNT
Photo at bench
uploaded to a job
jsnr-photos (S3)
key job-id/stage
jsnr-photo
check, resize, link
drop if not an image
jsnr-composer
photo: optional
ETA: if honest
stage durations
from settings
compute ETA
duration + buffer,
round, opening hours
none set → no ETA
The photo is a bonus, never a blocker; the ETA is computed and honest, or it is left out.

Fig 4. Two lanes feed one update. A bench photo lands in S3 under a job-and-stage key, gets validated and resized, and links to the job. The ETA is computed from the next stage's duration plus a buffer — or omitted when the shop hasn't set one.

The payoff, and the restraint

Put together, these two attachments are what make a one-line text feel like a shop that's on top of things. The bike-repair customer gets a photo of their gears stripped down and "ready Saturday morning"; the tailoring-studio customer gets a snap of the pinned hem and "ready for a fitting Wednesday afternoon." The photo proves it's real; the ETA lets them plan. Neither needed anyone to write anything.

The restraint is just as important as the richness. A photo is dropped rather than sent broken; an ETA is omitted rather than guessed. A customer would forgive a plain "in progress" far sooner than they'd forgive a confident "ready Thursday" that turns out to be Monday. By computing the time from real durations, always buffering later, and falling silent when there's nothing solid to say, the system keeps the one currency that matters here: a customer who believes what your messages tell them.

DESIGN RULES THAT SHAPED PHOTOS AND ETAS

- The key does the linking. A photo stored under job-id and stage attaches itself to the right update.
- Tidy or drop. Validate and resize every photo; a broken attachment never goes out.
- Photos are optional. No picture is a quieter message, never a blocked one.
- Compute the ETA, don’t guess it. Next-stage duration plus an honest buffer, rounded and within opening hours.
- Buffer late, or stay silent. The estimate only ever moves later; no duration set means no ETA at all.

PART 5 OF 7

JUNE 25, 2026 PART 5 OF 7 · JOB STATUS NOTIFIER SERIES ~7 MIN READ

How a quiet stage gets chased

The notifier is judged less by the updates it sends when things move than by what it does when they don't. This post is about the daily sweep: how it finds a job that has been stuck in one stage too long, sends the customer an honest holding note rather than leaving them guessing, and nudges the shop — without ever nagging or texting at midnight.

KEY TAKEAWAYS

- A daily sweep runs on EventBridge Scheduler and checks how long each open job has sat in its current stage.
- Each stage has its own dwell threshold in settings — "awaiting parts" is allowed longer than "ready to collect".
- A job past its threshold gets two things: an honest holding note to the customer and a nudge to the shop.
- The sweep de-duplicates, so a stuck job is chased once, not every day — it never turns into a nag.
- It respects quiet hours and never invents progress; a holding note says the truth, that the job is still at this stage.

The updates that don't fire themselves

Everything so far reacts to movement: a job changes stage, a message goes out. But the moments that actually lose a customer's trust are the ones where *nothing* moves. A part is back-ordered and the job sits in *awaiting parts* for a week. A repair is finished but nobody texts, so it waits in *ready* while the customer assumes it isn't done. A job quietly falls behind a busy bench and is simply forgotten. None of these produce a stage change, so none of them would ever trigger a message — which is exactly why a customer is left guessing. The sweep exists to catch the absence of movement.

How the sweep finds a stuck job

Once a day, on an EventBridge Scheduler rule timed for early morning, `jsnr-sweep` wakes up and walks the open jobs. For each one it works out how long the job has been sitting in its current stage — the time since the last entry in `jsnr-stage-history`. Then it compares that dwell time against the threshold for that stage, read from the shop settings. The thresholds differ by stage on purpose: a day in *diagnosing* might be fine, two days in *in progress* normal, but more than a few hours in *ready* means a finished job nobody's collected and the customer probably hasn't been told. *Awaiting parts* gets the longest leash, because everyone understands a supplier delay — but even that has a limit, beyond which silence becomes neglect.

A job under its threshold is left alone; the sweep is meant to be quiet, not chatty. A job over its threshold is a stuck job, and that's where the sweep acts.

What chasing actually does

A stuck job gets chased in two directions at once. Outward, the customer gets an honest holding note — routed through the same composer and sender as any other message, so it lands in the shop's voice on the customer's channel. The wording is built on the one true fact available: the job is still at this stage, and here's what that means. "Hi Maya — a quick update: your repair is still waiting on a part from our supplier. We haven't forgotten it, and we'll text you the moment it's back in. Thanks for your patience." It promises nothing it can't keep, and it never pretends the job moved — the composer's rule against inventing a stage holds here just as it does on a real move.

Inward, the shop gets a nudge — a short digest of every job over threshold, so the bench can act before the customer has to chase. That's often the more valuable half: the holding note buys patience, but the nudge is what actually gets the part chased or the finished job handed over. The customer-facing message keeps goodwill; the internal one fixes the cause.

The guard that makes this safe to run daily is dedupe. Left naive, the sweep would re-send the same holding note every morning a job stayed stuck — turning a thoughtful update into a daily nag that trains the customer to ignore you. So the sweep records, in the notifications log, that it has already chased a given job at a given stage, and won't chase the same job-and-stage again until either the job moves or a much longer re-chase interval passes. A stuck job is acknowledged once, not every day. And like every other message, a holding note that would land inside quiet hours is held until morning — a stalled job is never urgent enough to justify a 3am text.

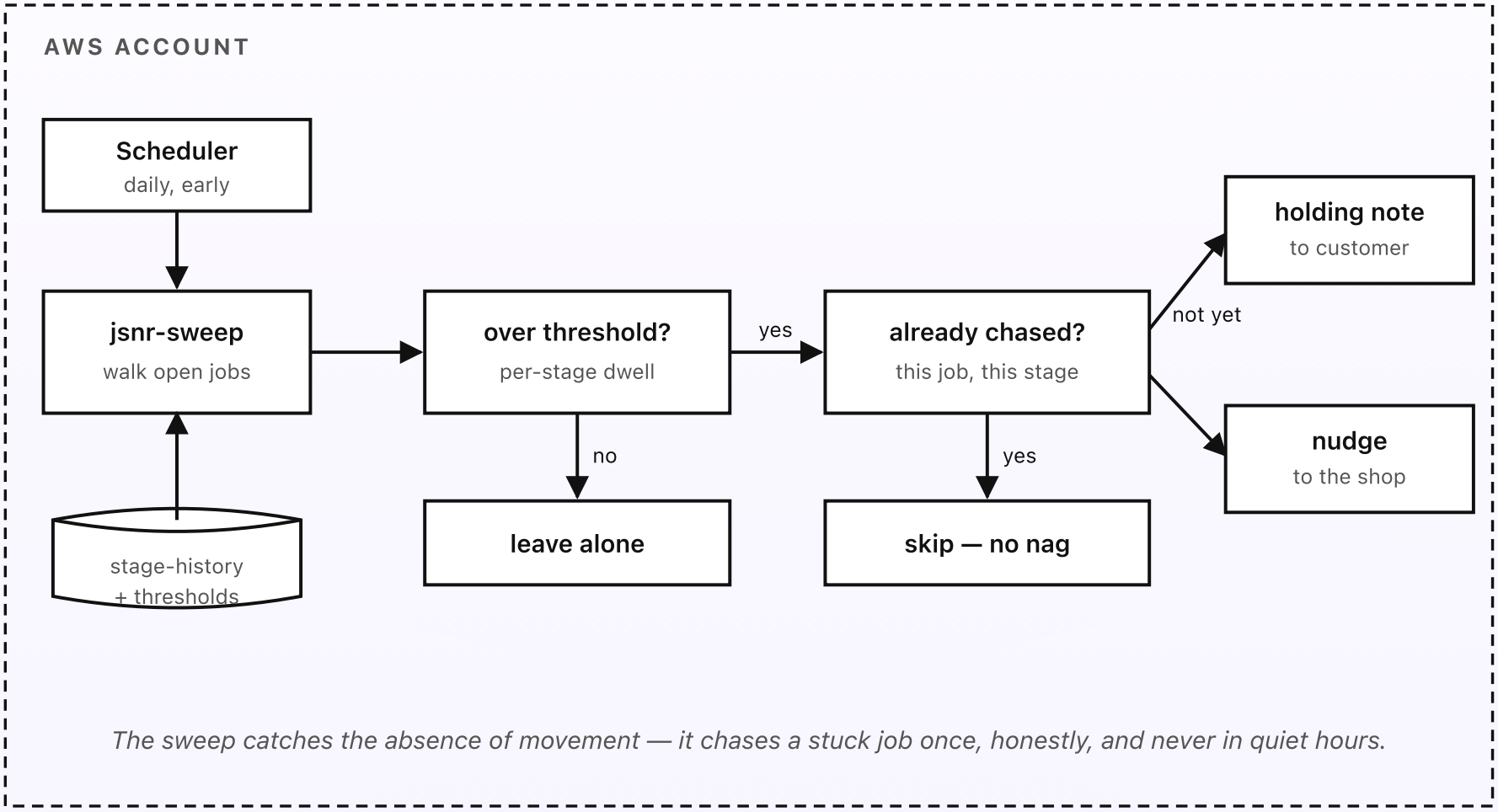

AWS ACCOUNT
Scheduler
daily, early
jsnr-sweep
walk open jobs
stage-history
+ thresholds
over threshold?
per-stage dwell
yes
no
leave alone
already chased?
this job, this stage
not yet
yes
holding note
to customer
nudge
to the shop
skip — no nag
The sweep catches the absence of movement — it chases a stuck job once, honestly, and never in quiet hours.

Fig 5. The daily sweep measures how long each job has sat in its stage, compares that against the per-stage threshold, and for anything stuck — and not already chased — sends an honest holding note to the customer and a nudge to the shop.

Why this half earns its keep

The reactive half of the system — a move, a message — is the part that's easy to demo. The sweep is the part that earns the goodwill. Customers rarely remember the routine "in progress" text; they vividly remember the time a shop went silent on them for a week, and the time a different shop got ahead of the problem with "just so you know, we're still waiting on your part." The first becomes a one-star review; the second becomes a regular. A stalled job is going to happen — suppliers are late, benches get busy — and the difference between a complaint and a shrug is almost always whether the customer heard about it before they had to ask.

And because the sweep reuses the same composer, sender, quiet-hours, and dedupe machinery as everything else, it adds this whole safety net for the cost of one scheduled function and a handful of thresholds in a doc. The shop tunes how patient or proactive it wants to be by editing numbers, not code.

DESIGN RULES THAT SHAPED THE SWEEP

- Catch the absence of movement. The sweep watches for jobs that *aren't* moving, which is what reactive updates miss.
- Thresholds per stage. "Awaiting parts" gets a long leash; "ready" gets a short one.
- Chase both ways. A holding note keeps the customer's trust; a nudge to the shop fixes the cause.
- Once, not daily. Dedupe per job-and-stage turns a useful update into something other than a nag.
- Same guards as everything else. Honest wording, no invented progress, and quiet hours always apply.

PART 6 OF 7

JUNE 25, 2026 PART 6 OF 7 · JOB STATUS NOTIFIER SERIES ~6 MIN READ

What the job status notifier costs

A notifier that costs more than the goodwill it earns is a toy. This post is the cost breakdown: every AWS service this system touches, what each one adds up to at around 200 jobs a month, why the total lands near $2.30 — with SMS as the biggest swing — and what happens to the bill when the shop gets ten times busier.

KEY TAKEAWAYS

- About $2.30/month at roughly 200 jobs — around five stage changes each, so a little over a thousand transitions watched.
- SMS is the biggest variable line: every text is metered per message, so it grows fastest as you push more updates to text.
- The only real fixed cost is Secrets Manager — $0.40 per secret — for the SMS-sender and board credentials.
- One small Bedrock Haiku 4.5 call per update writes the wording; at this volume that's well under half the bill.
- At ten times the volume the bill lands near $16, not $23 — the fixed lines don't move, and email scales far cheaper than SMS.

Where the money goes

The notifier is serverless end to end. Nothing runs while the board is still, so there's no instance ticking over overnight and no idle bill — you pay only when a job actually moves. At a typical small-shop volume — call it 200 jobs a month, each passing through about five stages, so a little over a thousand transitions watched and roughly 800 customer-facing updates sent — here's the whole bill, line by line. Of those 800 updates, say about 250 go by SMS to customers who chose text and 550 by email, which is the split that shapes the table.

AWS service	What it does here	Monthly
SNS (SMS)	Text updates to customers who chose SMS (~250 messages)	$0.90
Bedrock (Claude Haiku 4.5)	One compose call per update (~800)	$0.40
Secrets Manager	Two secrets — SMS sender, board access ($0.40 each)	$0.80
DynamoDB (on-demand)	Jobs and stage history — small reads and writes	$0.06
CloudWatch Logs	Function logs, 7-day retention	$0.05
SES (outbound)	Email updates to customers who chose email (~550)	$0.04

AWS service	What it does here	Monthly
Lambda (Python 3.14, arm64)	Stage watcher, composer, sender, sweep, photo handler	$0.02
S3	Job photos, resized and expired after collection	$0.01
SQS + DLQ	Buffering stage events to the composer	$0.01
EventBridge (bus + Scheduler)	Stage-change events, board poll, daily sweep	$0.01
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~200 jobs/month**	**$2.30**

The shape of that bill is the point. SMS is the largest line, and it's the only one that's genuinely sensitive to a design choice you make — how many updates you push to text rather than email. Each text is metered per message, so 250 of them outweigh everything the system actually *does*. The model that writes every message, by contrast, costs $0.40, because a Haiku 4.5 call on a short bundle of facts is a fraction of a cent. The watching, the matching, the photo handling, the sweep — all the real work — rounds to pennies, because it's plain Python and tiny reads and writes.

The one real fixed cost

It's worth naming Secrets Manager, because it's the only thing here that costs money while the system sleeps. Two secrets at $0.40 each is $0.80 a month no matter what — more than a third of the bill at this volume — whether you move one job or a thousand. One holds the SMS-sender credential (the registered sender ID or number SNS delivers from); the other holds the board access the watcher uses to read stages. Everything else on the list is genuinely usage-priced and rounds to zero at idle, which is exactly what you want from a system that only does work when a job moves.

One cost that is *not* on this table: SMS pricing varies a lot by country and carrier — from a fraction of a cent for domestic transactional messages to several cents for some international destinations. The $0.90 here assumes mostly domestic texts; a shop sending internationally, or one that lets every customer pick SMS, would see that single line grow faster than all the others combined. That's the real lever: email is effectively free at this scale, so steering customers who are happy with email toward email keeps the bill flat.

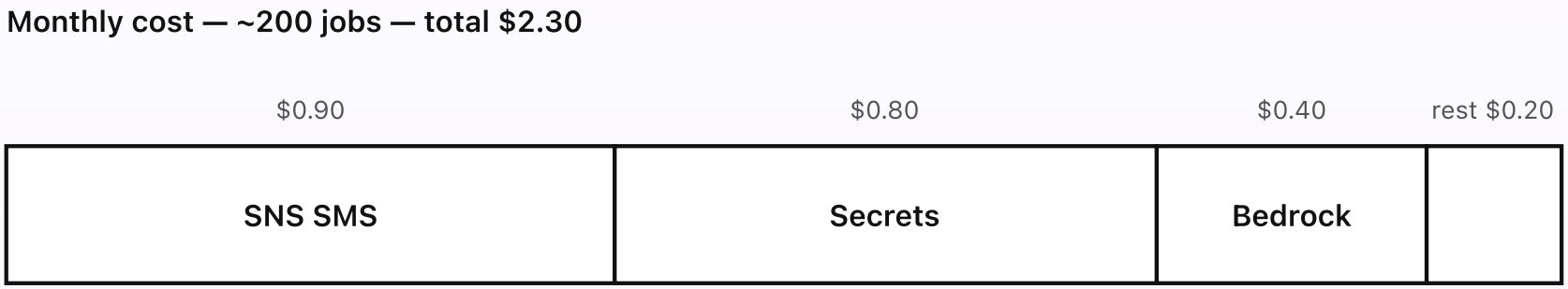


Fig 6. The monthly bill at about 200 jobs. SMS and Secrets Manager are most of it; the model that writes every message is $0.40, and everything that does the real work rounds to cents.

What ten times the volume costs

Push this to a busy shop — 2,000 jobs a month, ten times the volume — and the bill lands near $16, not $23. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules and the bus stay at a cent, and AWS Budgets stays free. What scales is the genuinely usage-priced work —

roughly $9 of SMS for ten times the texts, about $4 of Bedrock for ten times the messages, and a dollar or so more spread across DynamoDB, SES, Lambda, and logs. Even at that size, SMS is still the line doing the most to set the total, which is why the channel mix is the thing to watch as you grow.

The honest way to read this: the AWS bill is rounding error against the alternative. A shop phoning or texting 800 status updates a month by hand — and remembering to chase the stalled ones — is hours of stop-start interruption every week, and the chasing is the part that quietly never gets done. $2.30, or even $16, buys those hours back and closes the gap that turns into bad reviews. The few jobs that genuinely go wrong still get a person's attention, because the sweep put them in front of the bench instead of letting them rot.

DESIGN RULES THAT SHAPED THE COST

- Pay per move, not per hour. No always-on compute means no idle bill.
- Watch the channel mix. SMS is metered per message and is the line that grows fastest; email is nearly free.
- Spend the model sparingly. One short Haiku call per update, and only to write — never to decide a stage.
- Know your one fixed cost. Secrets Manager is the only thing that bills while the board is still.
- Cheap work stays cheap. Watching, matching, photos, and the sweep are plain Python and tiny reads.

PART 7 OF 7

JUNE 25, 2026 PART 7 OF 7 · JOB STATUS NOTIFIER SERIES ~7 MIN READ

Engineering reference: the job status notifier architecture

This is the job status notifier with the friendly labels removed: the real resource names, the runtime, the table key schemas, the event bus and its rule, the two schedules, and the IAM scope. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Five Lambda functions, all Python 3.14 on arm64, decoupled by a custom EventBridge bus and one SQS queue with a dead-letter queue.
- Three DynamoDB tables, all on-demand: the jobs board mirror, an append-only stage history, and a notifications log.
- Stage changes travel as `stage.changed` events on `jsnr-bus`; a rule routes them to SQS — no API Gateway anywhere.
- Two EventBridge Scheduler rules: a few-minute board poll and a daily quiet-stage sweep.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by the composer. Single region, `eu-west-2`.

The architecture, for engineers

This is the same system as Part 1 with the friendly labels removed and the real resources named. Everything is in one region, `eu-west-2` (London), in one account. There is no API Gateway, no NAT Gateway, and nothing always-on; the board is read by a scheduled poll, stage changes move as events on a custom bus, and work is buffered on a single SQS queue.

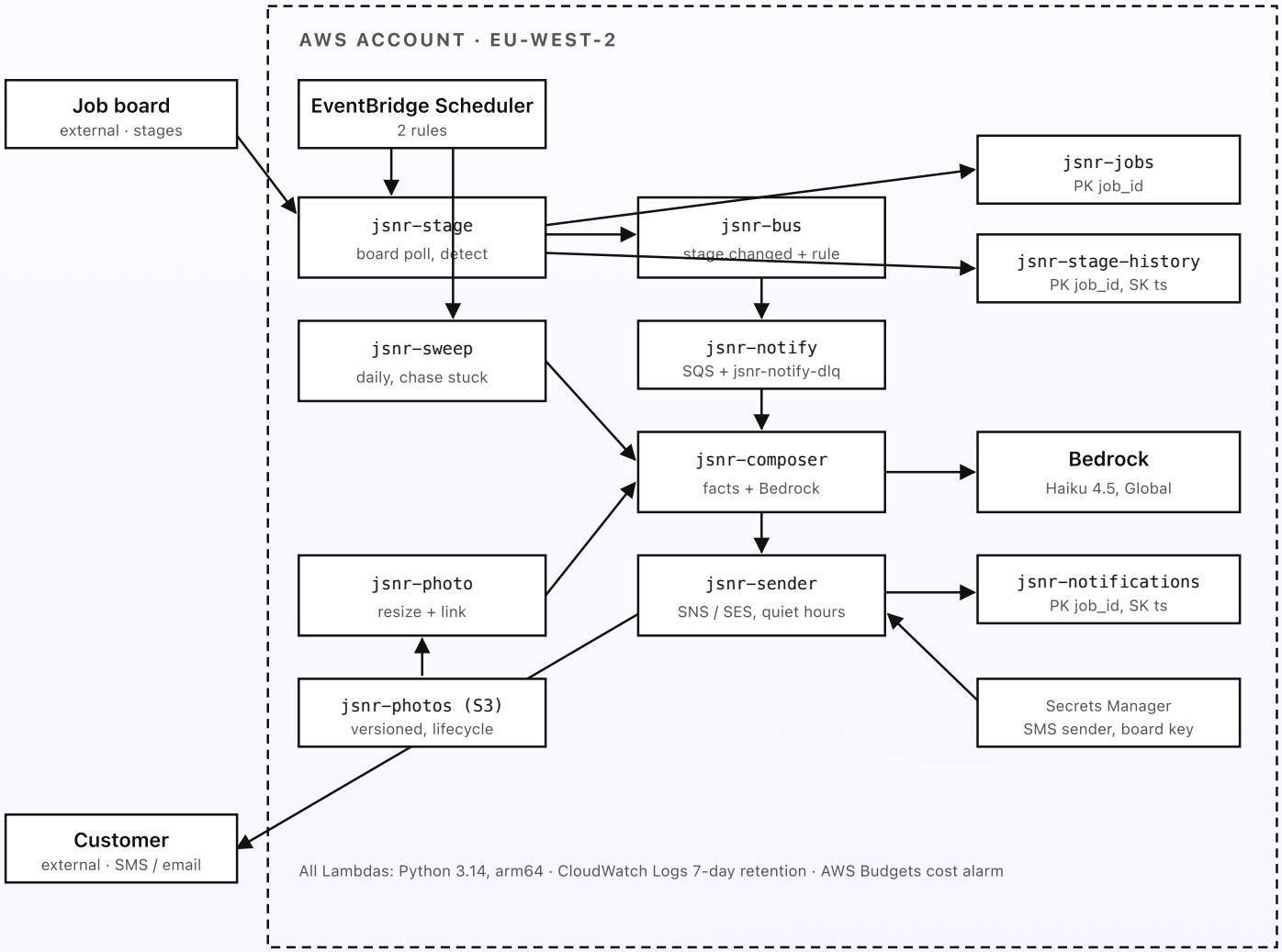

AWS ACCOUNT · EU-WEST-2
Job board
external · stages
EventBridge Scheduler
2 rules
jsnr-stage
board poll, detect
jsnr-bus
stage.changed + rule
jsnr-jobs
PK job_id
jsnr-stage-history
PK job_id, SK ts
jsnr-sweep
daily, chase stuck
jsnr-notify
SQS + jsnr-notify-dlq
jsnr-composer
facts + Bedrock
Bedrock
Haiku 4.5, Global
jsnr-photo
resize + link
jsnr-sender
SNS / SES, quiet hours
jsnr-notifications
PK job_id, SK ts
jsnr-photos (S3)
versioned, lifecycle
Secrets Manager
SMS sender, board key
Customer
external · SMS / email
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm

Fig 7. The job status notifier drawn for engineers: a scheduled board poll, a custom event bus, an SQS-buffered composer and sender, three DynamoDB tables, an S3 photos bucket, Bedrock called only by the composer, and two scheduled jobs. One region, one account, no API Gateway.

Lambda functions

Five functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. Each does one job and hands off; the custom bus and the SQS queue (`jsnr-notify`, with `jsnr-notify-dlq` as its dead-letter queue after five attempts) decouple watching the board from the slower model and send calls.

- `jsnr-stage` — scheduled poll. Reads the board, compares each job's stage against the last recorded value in `jsnr-jobs`, validates known-stage / forward-only / not-seen, appends to `jsnr-stage-history`, updates the job, and emits one `stage.changed` event onto `jsnr-bus`. No customer contact of its own.
- `jsnr-composer` — SQS-triggered off `jsnr-notify`. Gathers facts from `jsnr-jobs`, the settings, and the photos bucket; computes the next-stage ETA; makes the single Bedrock call; validates the draft against the facts; and hands a finished message to the sender. The only function with `bedrock:InvokeModel`.
- `jsnr-sender` — delivers on the customer's channel (SNS for SMS, SES for email), enforces quiet hours, de-duplicates against `jsnr-notifications`, and writes the sent record.
- `jsnr-sweep` — scheduled daily. Walks open jobs, measures dwell time from `jsnr-stage-history` against the per-stage threshold, and for anything stuck

(and not already chased) routes a holding note through the composer and sends a digest nudge to the shop.

- `jsnr-photo` — triggered by uploads landing in `jsnr-photos`. Confirms the file is an image, resizes it, strips metadata, and records the link on the job. Drops anything invalid.

Data stores, events, and channels

- **DynamoDB (all on-demand).** `jsnr-jobs` — PK `job_id`; the board mirror, holding customer name, contact, channel, item, current stage, last-recorded stage, and photo links. `jsnr-stage-history` — PK `job_id`, SK `ts`, append-only, one row per recorded move with from-stage and to-stage. `jsnr-notifications` — PK `job_id`, SK `ts`, the sent log used for dedupe and the audit trail of every message and the facts it was built from.
- **S3.** `jsnr-photos` — job photos keyed `job_id/stage`, versioning on, with a lifecycle rule that expires photos a set period after a job is collected.
- **EventBridge.** A custom bus, `jsnr-bus`, carrying `stage.changed` events, with one rule matching them to the `jsnr-notify` SQS target. Scheduler holds two rules — the board poll at `rate(3 minutes)` and the sweep at a daily `cron` (early morning, before opening).
- **SNS and SES.** SNS sends transactional SMS from a registered sender ID / originating number; SES sends email replies and the shop nudge from a verified domain with DKIM.
- **Secrets Manager.** One secret for the SMS-sender credential and one for the board access the poll uses; fetched at call time, never in env vars or the board

itself.

- **Bedrock.** Model id `anthropic.claude-haiku-4-5` via the Global cross-Region inference profile, invoked only by `jsnr-composer`.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `jsnr-stage` can read the board secret, read and write `jsnr-jobs`, write `jsnr-stage-history`, and put events to `jsnr-bus`; it cannot call Bedrock or send anything. `jsnr-composer` is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile, and it can read the jobs table and the photos bucket but cannot send directly. `jsnr-sender` can publish to SNS, send via SES, read the SMS secret, and write `jsnr-notifications` — and nothing else. `jsnr-sweep` can read the history and jobs and invoke the composer path, but holds no inbound surface. `jsnr-photo` can read and write only the photos bucket and update the job's photo link. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call for capacity and is not a data store. An AWS Budgets alarm watches the monthly spend and notifies if it drifts above a few dollars — the cheapest possible early warning that a poll loop or an SMS misconfiguration is running away.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Five small Lambdas beat one that does everything; the bus and the queue decouple the slow calls.
- No public surface. Nothing is reachable from outside; the board is polled, and everything else is events and schedules.
- Least privilege, per role. Only the composer can call Bedrock; only the sender can send.
- State in DynamoDB, photos in S3. Tables for jobs, history, and notifications; the bucket for images.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called once per update.
- A budget alarm is a smoke detector. The cheapest way to learn a poll looped or SMS ran away is a Budgets alert.