7-PART SERIES · FREE COMPANION

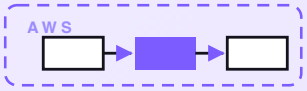


Photo tagger

A serverless tagger that saves an online shop hours of typing. When new product photos land, it looks at each one and drafts the boring-but-important details — a clear title, alt text for accessibility, suggested tags and category, and a short description — so listings stay consistent and searchable. The owner reviews and approves; it never publishes a listing on its own, and it flags low-quality or wrong-looking images for a human. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Photo tagger

PART 1 OF 7

JUNE 6, 2026 PART 1 OF 7 · PHOTO TAGGER SERIES ~5 MIN READ

A product photo tagger on AWS for a few dollars a month

An online shop takes a hundred photos of new stock and then sits down to the part nobody enjoys: typing a title for each one, writing alt text so the listing is readable for shoppers using a screen reader, picking tags and a category, and writing a short description. It is hours of repetitive work, and the results drift — one person writes “blue mug,” the next writes “Ceramic Coffee Cup — Navy,” and the catalog gets harder to search. This post walks through the design of a small tagger that looks at each new photo, drafts those details in one consistent voice, and hands them to the owner to approve. It never publishes on its own, and it flags any photo that looks wrong.

KEY TAKEAWAYS

- Two ways a photo gets in: a Drive folder and a direct S3 drop. A new file starts the work.
- A cheap deterministic step resizes the photo and runs quality checks before any model looks at it.
- One Bedrock vision call drafts five fields: title, alt text, tags, category, and a short description.
- Every draft waits for a human. Approve writes it to your store; nothing goes live on its own.
- Designed on AWS for about $2.40/month at typical small-business volume.

The whole system on one page

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

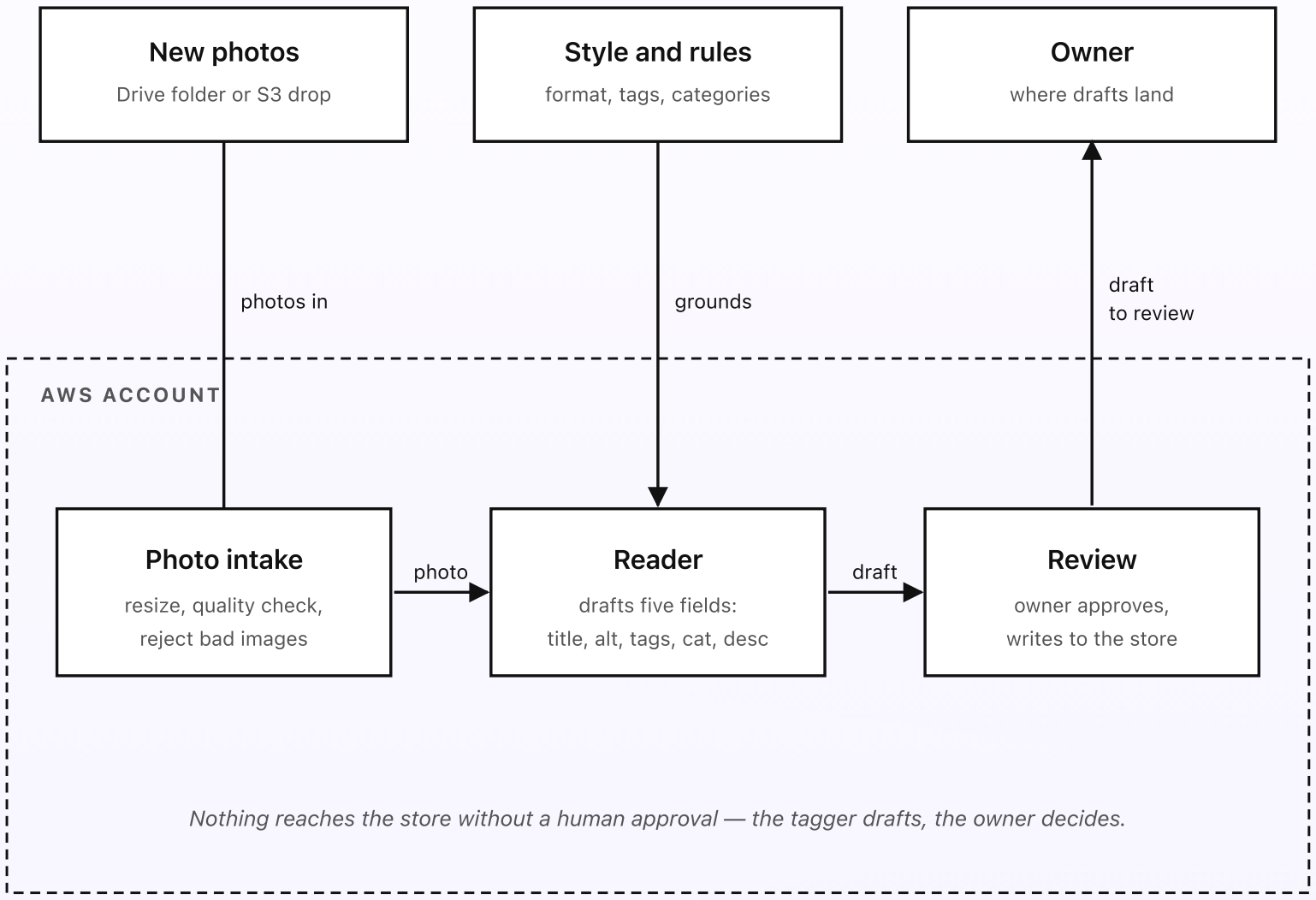

New photos
Drive folder or S3 drop
Style and rules
format, tags, categories
Owner
where drafts land
photos in
grounds
draft
to review
AWS ACCOUNT
Photo intake
resize, quality check,
reject bad images
photo
Reader
drafts five fields:
title, alt, tags, cat, desc
draft
Review
owner approves,
writes to the store
Nothing reaches the store without a human approval — the tagger drafts, the owner decides.

Fig 1. Two sources outside, three pieces inside AWS. Photos flow in from a Drive folder or a direct S3 drop. The Reader drafts five listing fields. Review shows the draft to the owner and writes the approved fields to the store.

What you set up once (the outside)

- **New photos.** A Google Drive folder where your team drops new product shots, or a direct S3 drop folder if you already upload to AWS. Either way, a new file is the trigger — the moment a photo lands, the tagger picks it up. You don't fill in a form or press a button; you just put the photo where it goes. Part 2 covers both lanes and the resize-and-check step that runs before anything else.
- **A style and rules folder.** Two short Google Docs in a Drive folder. The *style* doc covers how a title should read (for example, "Material + Product + Colour"), the list of tags and categories you actually use, and the words to prefer or avoid (your brand says "navy," not "dark blue"). The *rules* doc holds the quality thresholds — how dark, how blurry, or how small a photo can be before it's rejected — and the list of things that mean "this isn't a product shot," like a screenshot or a receipt. A rep can edit either doc without a deploy.
- **Owner.** The person who reviews each draft — usually the shop owner or whoever runs the catalog. Each draft lands as a card with the photo and the five drafted fields, plus three buttons: *Approve*, *Edit*, and *Reject*. The card arrives by email and on a simple web page, so the owner can clear a batch in a few minutes.

What runs on every photo (the inside)

- **The photo intake.** When a photo lands, a small Lambda makes a smaller copy of it (a full-size photo is far bigger than the model needs), then runs plain

quality checks: is it bright enough, sharp enough, and large enough to be a real product shot? Too dark, too blurry, or too small, and the photo is rejected straight away with a reason — no model is called, because there's no point reading an image that's clearly unusable. Good photos move on to the reader.

- **The reader.** One call to Bedrock Claude Haiku 4.5 with vision (a model that can look at a picture, not just text). It reads the photo and drafts five fields: a clear title, alt text that describes the item for a shopper using a screen reader, suggested tags, a category, and a short description — all in the voice your style doc sets. The model also marks how confident it is in each field and says if the photo doesn't look like a clean product shot. The reader never invents a detail it can't see; if it can't tell the colour, it says so rather than guessing.
- **Review.** The draft is stored and shown to the owner as a card. *Approve* writes the fields to your store (or an export sheet you import later) and archives the draft. *Edit* opens a form pre-filled with the draft so the owner can fix a word and then approve. *Reject* sends the photo to a flagged folder with a reason. Anything the reader marked low-confidence or wrong-looking is held back for a human instead of slipping through. Every action is logged, so a mistake can be undone and the trail is clear.

In plain words

Your team photographs a new ceramic mug in navy. The photo lands in the Drive folder at 2pm. The intake shrinks it and checks it — bright, sharp, big enough — so it passes. The reader looks at it and drafts: title "Ceramic Mug — Navy," alt text "Navy ceramic coffee mug with a curved handle, shown on a white background," tags "mug, ceramic, navy, kitchen, drinkware," category "Drinkware," and a two-

line description in your house voice. The owner gets a card at 2:01pm, glances at it, fixes “navy” to “midnight” because that’s the actual product name, and taps Approve. The fields are written to the store. The whole thing took the owner ten seconds instead of three minutes, and the next twenty mugs read the same way.

The cost of running this is about $2.40 a month at SMB volume. The cost of *not* running it is the afternoon someone loses typing listings by hand, the inconsistent titles that hurt search, and the listings that quietly ship with no alt text at all — which both shoppers using screen readers and search engines notice.

DESIGN RULES THAT SHAPED EVERY DECISION

- Nothing reaches the store without a human approval. The tagger drafts; the owner decides.
- Cheap checks first. A bad photo is rejected by plain code before any model is paid to read it.
- The model never invents what it can’t see. Unsure means “say so,” not “guess.”
- Wrong-looking images are flagged for a human, not drafted as if they were fine.
- The style lives in a doc. Changing a tag list or a title format doesn’t need a deploy.
- Every action is logged. Approve, edit, or reject — the trail is there to audit later.

Why this shape

Most shops handle photo listings in one of two ways: somebody types every field by hand, or somebody pastes the same template and barely edits it. The hand-typing is slow and drifts — ten people produce ten styles, and the catalog gets harder to search every week. The pasted template is fast and useless — every mug gets the same generic blurb and no real alt text, so the listing is worse than if someone had thought about it for thirty seconds.

The setup above puts a careful first draft in front of the owner for every photo, written in one consistent voice from the shop's own style doc. The owner is still in charge — they read, fix, and approve — but they start from something good instead of a blank field. The boring 90% is done; the human spends their time on the 10% that actually needs judgement. And because a wrong or low-quality photo is caught before it ever becomes a draft, the owner isn't wading through nonsense to find the real work.

The next four posts walk through each piece in turn: how a product photo gets read, how the details get drafted, how a bad photo gets flagged, and how a listing gets approved. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 6, 2026 PART 2 OF 7 · PHOTO TAGGER SERIES ~4 MIN READ

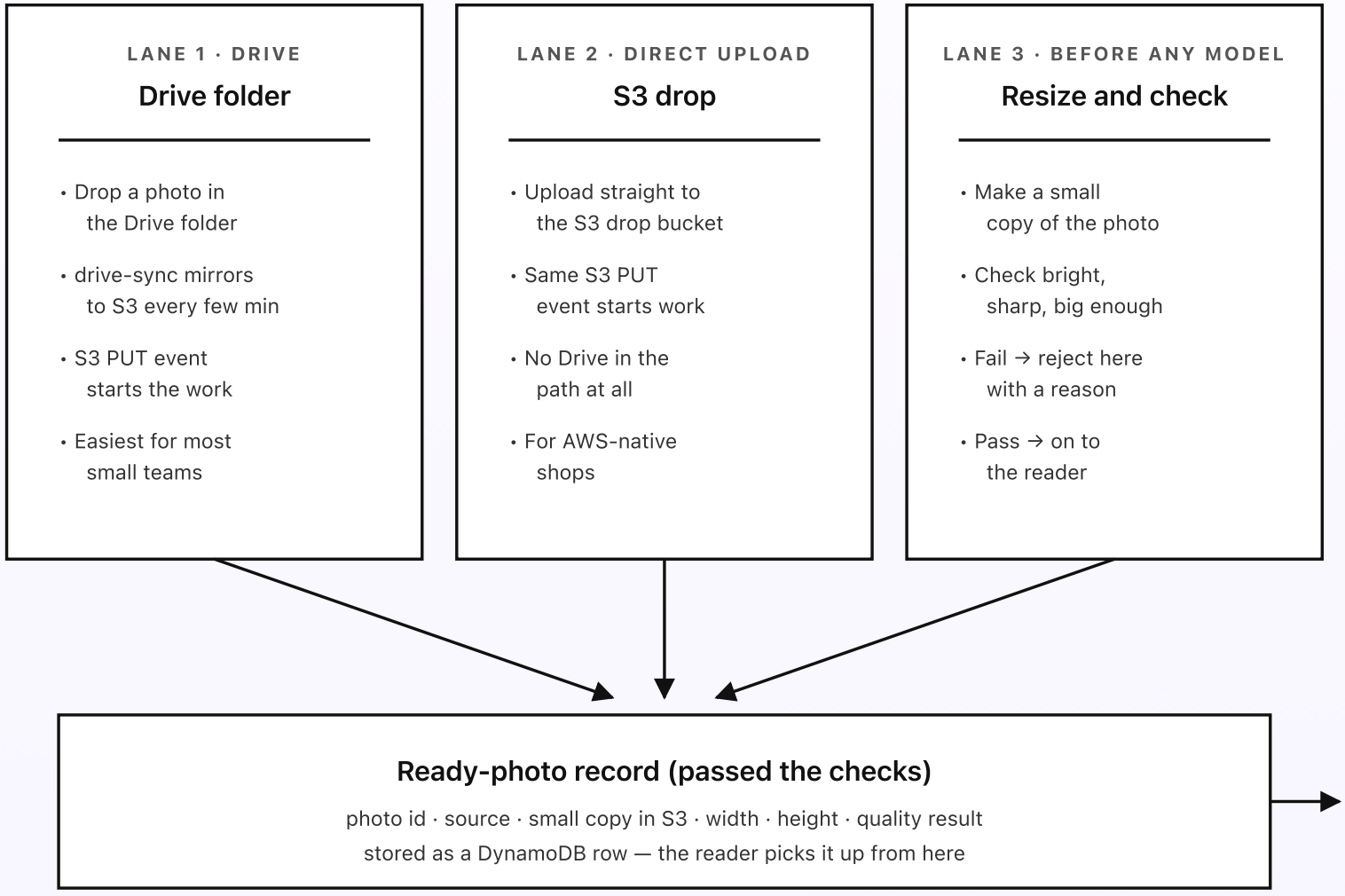
How a product photo gets read

The tagger only works on photos it can see. So the first job is getting each new photo in and getting it ready. There are two ways a photo gets in: somebody drops it in a Drive folder, or it's uploaded straight to an S3 bucket. Either way, before any model is called, a small step shrinks the photo and runs plain quality checks. That step matters more than it sounds — it's what keeps the model from being paid to read a blurry mess, and it's the cheapest place to catch a photo that should never have been sent.

KEY TAKEAWAYS

- Two intake lanes feed one queue: a Drive folder and a direct S3 drop.
- A `drive-sync` Lambda mirrors new Drive files to S3 every few minutes.
- Every photo is resized to a small copy before anything else — the model doesn't need the full file.
- Plain quality checks (too dark, too blurry, too small) reject bad photos with no model call.
- Only a photo that passes the checks moves on to the reader in Part 3.

Two lanes, then one resize-and-check



The resize-and-check runs before any model — the cheapest place to drop a bad photo is before you read it.

Fig 2. Two lanes converge on one resize-and-check step. The Drive lane and the S3-drop lane both end in the same S3 PUT event; the intake Lambda shrinks the photo and runs plain quality checks. Only a photo that passes becomes a ready-photo record for the reader.

Lane 1: the Drive folder

The simplest lane for most teams. Drop a photo into the shared Drive folder and walk away. A small Lambda — `drive-sync` — runs every few minutes, looks for files it hasn't seen before, and copies each one to `s3://pt-photo-drop/` using the Google Drive API (its credentials live in Secrets Manager). The copy into S3 fires an S3 PUT event, and that event is what actually starts the tagger. The Drive folder stays the place humans interact with; S3 is where the work happens.

This lane covers the common case: a team that photographs products on a phone or camera, dumps the shots into a shared Drive folder, and doesn't want to learn anything new. They already use Drive; nothing changes for them.

Lane 2: the direct S3 drop

Some shops already upload images to AWS — their store pulls product images from an S3 bucket, or they have a build step that puts photos there. For them, the Drive hop is pointless. Lane 2 lets a photo be uploaded straight to the same `s3://pt-photo-drop/` bucket (or a prefix in it), and the exact same S3 PUT event starts the work. There's no second code path to maintain — both lanes end at the same event, so everything downstream is identical.

A shop can use one lane, the other, or both. A team might drop phone photos in Drive and have their web build push studio shots straight to S3; both kinds of

photo flow through the same checks and the same reader.

Lane 3: resize, then check — before any model

This is the step that earns its keep. When the S3 PUT event fires, a small intake Lambda loads the photo and does two things. First, it makes a smaller copy — a product photo straight off a phone can be several megabytes and many millions of pixels, and the model reads it just as well at a fraction of that size. The smaller copy is cheaper to send, faster to process, and is what every later step uses; the original is kept untouched.

Second, it runs plain quality checks against the thresholds in the rules doc. Is the photo bright enough, or is it nearly black? Is it sharp, or is it a blur? Is it large enough to be a real product shot, or a tiny thumbnail? Is the shape sensible, or is it a long thin banner that's clearly not a product? These are simple measurements — no model, no AI, just arithmetic on the pixels. A photo that fails is rejected right here, with a reason written to its record ("too dark"), and it never reaches a model. Part 4 goes deeper on flagging; the point for now is that the obvious rejects are caught before anyone pays to read them.

A photo that passes becomes a ready-photo record: the small copy in S3, plus a DynamoDB row with the photo id, where it came from, its size, and its quality result. That record is what the reader picks up in the next post.

Why a record, not an immediate model call

The intake step doesn't call the model itself — it writes a record and lets the next step pick it up. That's deliberate. Photos arrive in bursts: a team uploads forty

shots at once after a photoshoot. If each upload tried to call the model the instant it landed, a big batch could hit rate limits or pile up. Instead, ready-photo records go onto a queue, and the reader pulls them at a steady pace. A burst of forty photos becomes forty calm records that get read one after another, and a failure on one photo never blocks the rest.

Next post: how the reader takes a ready photo, calls Bedrock vision once, and drafts the five listing fields — title, alt text, tags, category, and description — each with a confidence score.

PART 3 OF 7

JUNE 6, 2026 PART 3 OF 7 · PHOTO TAGGER SERIES ~5 MIN READ

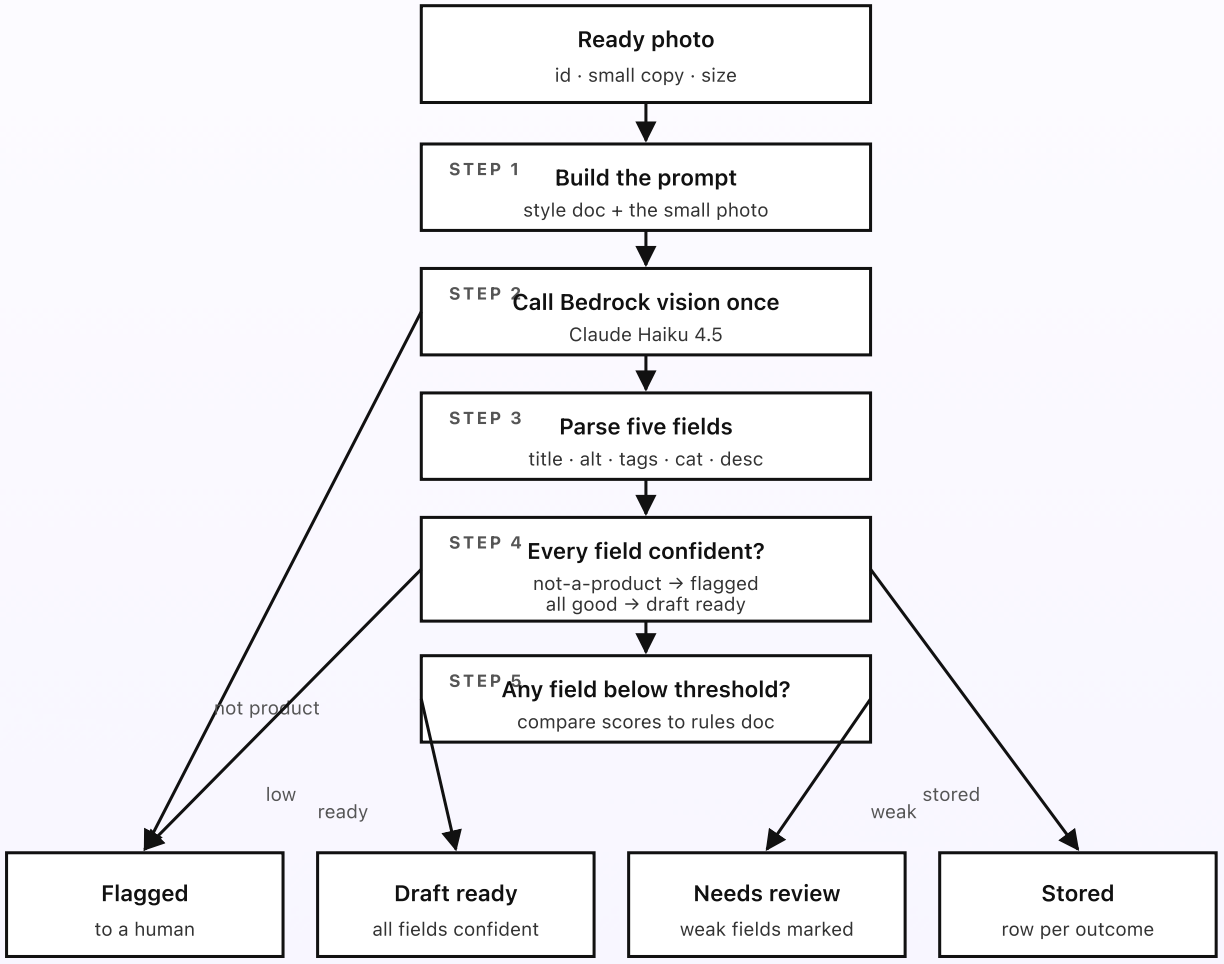
How photo details get drafted

A photo passed the checks and is ready. Now the reader does the actual tagging: it calls Bedrock vision once, gives it the small photo and the rules from your style doc, and asks for five fields back — a title, alt text, tags, a category, and a short description. The model returns them as plain structured data, with a confidence score on each field, and it is told plainly not to make up anything it can't see. This is the one place a model is used, and it's used carefully.

KEY TAKEAWAYS

- One Bedrock Claude Haiku 4.5 vision call per photo — no second model, no loop.
- The prompt includes the style doc: title format, the real tag list, the category list, words to prefer or avoid.
- Five fields come back as structured data: title, alt text, tags, category, description.
- Each field carries a confidence score; the model says when it can't tell rather than guessing.
- Low-confidence or wrong-looking results route to a human instead of becoming a clean-looking draft.

The drafting flow, per photo



The model is told to say when it can't tell — an honest blank beats a confident wrong title.

Fig 3. The reader's flow, per photo. One vision call drafts five fields; confidence scores decide whether the draft is ready, needs a closer look, or should be flagged for a human. The style doc shapes every word.

The style doc: what makes the drafts sound like you

A model with no guidance writes generic listings. The style doc is what makes the drafts sound like your shop instead of a stock catalog. It has a few short sections: the title format ("Material + Product + Colour," or whatever your store uses), the actual list of tags you allow (so the model picks from your tags, not invented ones), the list of categories (so it places the item in a real bucket), and a small word list — words your brand prefers ("midnight," not "dark blue") and words to avoid. The doc lives in Drive and is mirrored to S3, so a rep can change a tag or a phrase without touching code.

Because the tag and category lists come from the doc, the model can't invent a tag you don't use. If it sees something that doesn't fit any of your tags, it says so rather than making one up — and that becomes one of the low-confidence signals that routes the photo to a human.

One call, five fields

The reader sends one request to Bedrock Claude Haiku 4.5 with the small photo and the style doc. The prompt is short and firm: "Look at this product photo. Draft these five fields. Return them as plain structured data. For each field, give a confidence score from 0 to 1. Only describe what you can actually see. If you can't

tell the colour, the material, or what the item is, say so — do not guess. If this isn't a clean product photo, set the not-a-product flag." The five fields:

- **Title.** A clear, consistent name in your title format. Short, no filler, no all-caps shouting.
- **Alt text.** A plain description of the item for a shopper using a screen reader — what it is, its colour, and any obvious feature. This is the field most catalogs skip, and the one that helps both accessibility and search the most.
- **Tags.** A handful of tags chosen from your allowed list. Not a hundred — the few that actually fit.
- **Category.** One category from your list. If nothing fits, the model says so rather than forcing it.
- **Description.** Two or three plain sentences in your house voice. Honest about what the photo shows; no invented features, no made-up materials.

Confidence is the safety valve

The confidence score on each field is what keeps the system honest. A photo of a plain mug on a white background is easy — the model is confident on all five fields, and the draft is ready for the owner. A photo where the colour is ambiguous under odd lighting, or the item is partly out of frame, produces lower confidence on the fields it's unsure about. The reader compares each score against the threshold in the rules doc and routes accordingly: all fields confident means the draft is ready; some fields weak means the draft is stored but the weak fields are marked so the owner's eye goes straight to them; the not-a-product flag, or a

uniformly low result, means the photo is flagged for a human instead of dressed up as a clean draft.

This is the whole reason the model is asked for confidence rather than just answers. A confident wrong title — “Red Mug” on a photo of a blue one — is worse than no title, because it sails through review when nobody’s looking closely. An honest “I’m not sure of the colour” sends the photo to the right place: a human’s eyes.

Why one call, and why Haiku

The reader makes exactly one model call per photo. No back-and-forth, no second model double-checking the first. One good vision call gets all five fields at once, which is cheaper and simpler than chaining calls, and the confidence scores plus the human review in Part 5 are the safety net — not a second model. Claude Haiku 4.5 is the right model here because the task is bounded and concrete: look at a clear product photo and describe it in five short fields. That doesn’t need the heaviest, most expensive model; the cheap fast one does it well, and the cost page shows just how cheap that makes the whole system.

Next post: how a bad photo gets flagged — the deterministic quality gate from Part 2 and the model’s own not-a-product check, working together so a wrong image goes to a human instead of becoming a tidy draft.

PART 4 OF 7

JUNE 6, 2026 PART 4 OF 7 · PHOTO TAGGER SERIES ~5 MIN READ

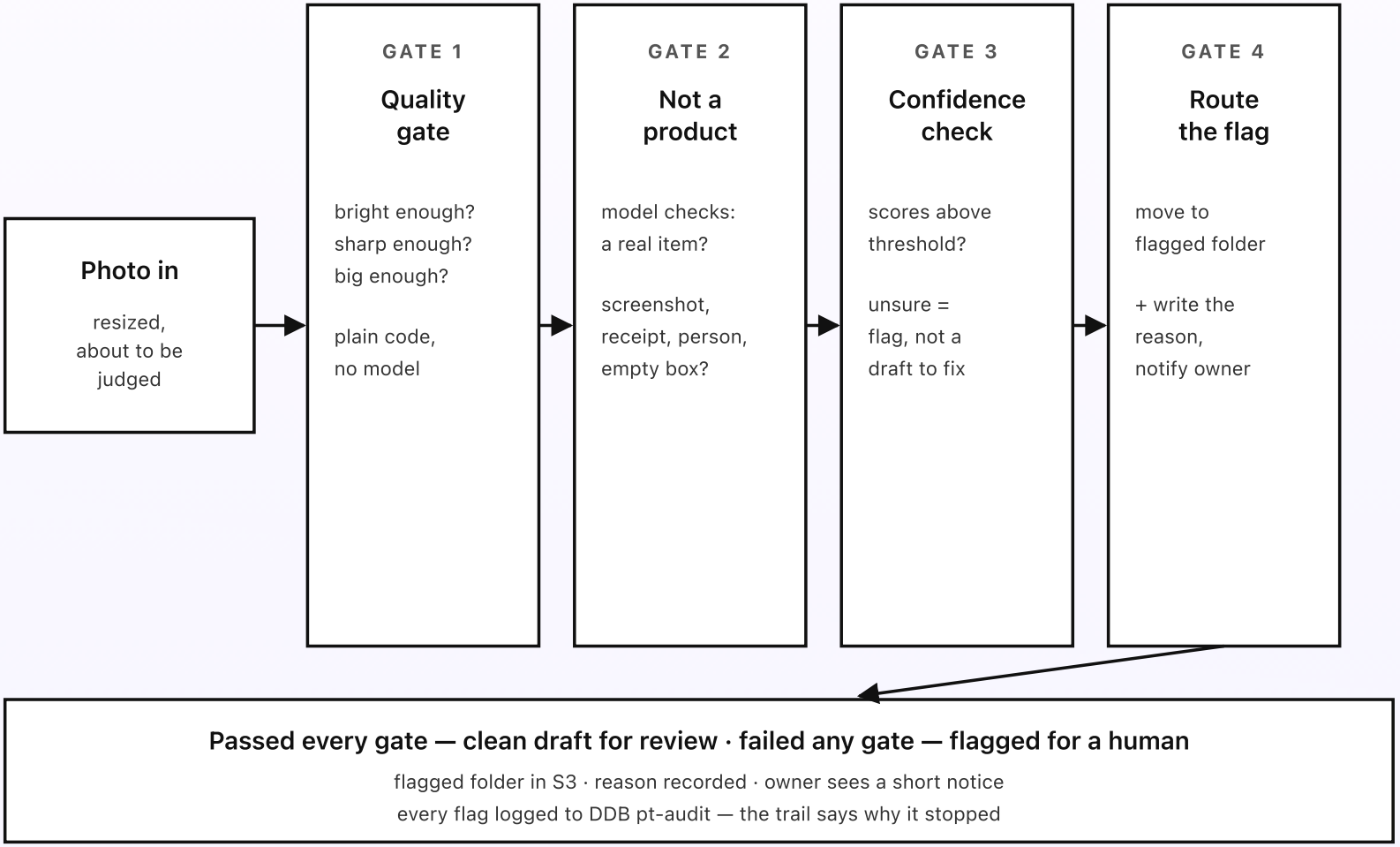
How a bad photo gets flagged

Not every file that lands in the folder is a good product photo. Somebody drops a screenshot by mistake. A photo comes out too dark to use. A shot of an empty box gets uploaded with the real ones. The wrong thing slips into the batch. The tagger's job here is simple to say and important to get right: catch those before they become a tidy-looking draft that sails through review. It does that in two layers — a plain quality gate before any model, and the model's own check on whether the photo even looks like a product.

KEY TAKEAWAYS

- Layer one is plain code: too dark, too blurry, too small, or odd shape — rejected before any model.
- Layer two is the model's not-a-product check: screenshots, receipts, people, empty boxes.
- Low confidence on the fields is treated as a flag, not a draft to clean up later.
- A flagged photo goes to a flagged folder with a reason; a person decides what to do.
- Nothing flagged ever reaches the store — flagging is a stop, not a slow-down.

Two layers between a photo and a draft



A flag is a full stop, not a maybe — nothing flagged is ever written to the store.

Fig 4. Two layers, four gates, between a photo and a clean draft. Plain code catches the unusable ones; the model catches the wrong-looking ones; low confidence is treated as a flag; the flag is routed to a human with a reason. Nothing flagged reaches the store.

Gate 1: the quality gate (plain code, no model)

This is the same gate introduced in Part 2, and it runs first because it's the cheapest. Plain code measures a few things about the photo and compares them to the thresholds in the rules doc. Brightness: is the average light level reasonable, or is the photo nearly black or blown-out white? Sharpness: does the image have real detail, or is it a smear? Size: is it big enough to be a real product shot, or a tiny thumbnail? Shape: is it a sensible rectangle, or a long thin banner that's obviously not a product photo? A photo that fails any of these is rejected here, with the exact reason recorded, and no model is ever called on it. The thresholds are numbers in a doc, so a shop with a darker product style can loosen the brightness floor without a deploy.

Gate 2: is this even a product?

A photo can be perfectly bright and sharp and still be the wrong thing. A screenshot of an order confirmation is a clean, crisp image. So is a photo of a receipt, a photo of a person, or a photo of an empty box. The quality gate has no way to know those are wrong — they pass every measurement. So the reader's prompt (from Part 3) explicitly asks the model: is this a clean product shot of an item for sale? If not, set the not-a-product flag and say what it looks like instead.

When that flag comes back set, the photo is treated as flagged no matter how confident the other fields look.

This is the one place the model is used as a check rather than a drafter, and it's worth it: spotting "this is a screenshot, not a mug" is exactly the kind of judgement plain code can't make and a vision model can.

Gate 3: low confidence is a flag, not a fixer-upper

Part 3 showed the model returns a confidence score on each field. Gate 3 uses those scores as a safety check, not just a hint for the reviewer. If the model wasn't sure what the item is, couldn't tell the colour, or couldn't place it in any of your categories, the photo is flagged for a human rather than presented as a clean draft with a few weak spots. The line between "needs a closer look" and "flag it" is a threshold in the rules doc, so a cautious shop can set it strict and a high-volume shop can set it looser.

The reason this matters: a draft that *looks* finished is the one a busy owner approves without reading. By turning genuine uncertainty into a flag instead of a tidy-looking draft, the system makes sure the photos that need human eyes actually get them.

Gate 4: route the flag to a person

A flagged photo doesn't just disappear. Gate 4 moves it to a flagged folder in S3, writes a row with the reason ("too dark," "screenshot," "low confidence on colour"), and notifies the owner with a short notice — not a full review card, just

“three photos were flagged today, here’s why.” A person then decides: retake the photo, delete it, or override the flag and tag it by hand. Nothing about a flag is silent, and nothing flagged is ever written to the store.

Why a flag is a stop, not a slow-down

It would be easy to let a borderline photo through with a “maybe” label and hope the reviewer catches it. The design says no on purpose. The whole value of the tagger is that the owner can trust the drafts — glance, approve, move on. The moment a few bad drafts slip through, the owner has to start reading every one carefully again, and the time savings evaporate. Treating a flag as a hard stop keeps the clean drafts genuinely clean, so the fast path stays fast. The cost of a flag is one photo a human has to look at; the cost of a false “looks fine” is the owner’s trust in the whole system.

Next post: how a listing gets approved — the three actions on every clean draft card, and how each one updates the store, the draft state, and the audit trail.

PART 5 OF 7

JUNE 6, 2026 PART 5 OF 7 · PHOTO TAGGER SERIES ~5 MIN READ

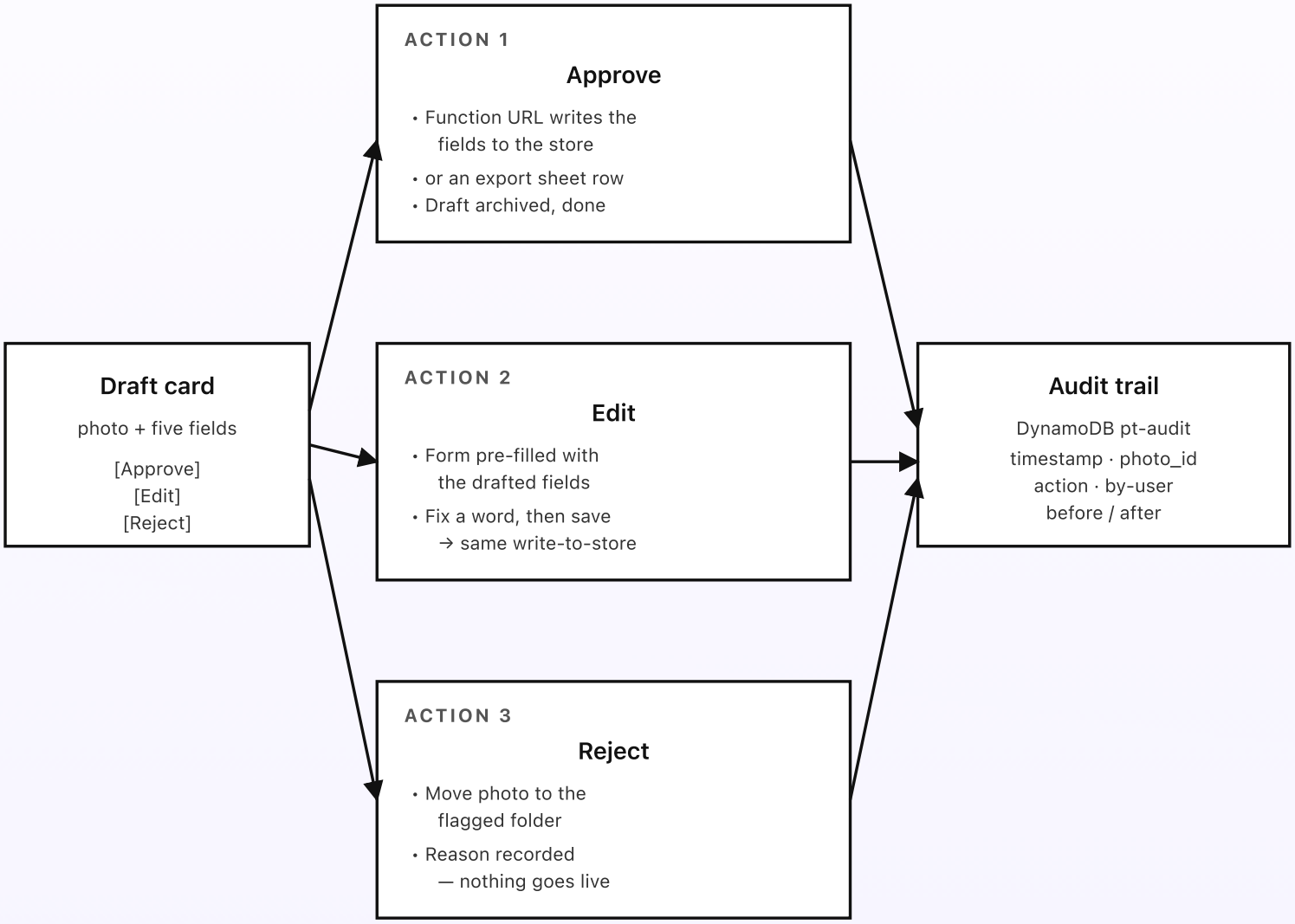
How a listing gets approved

A clean draft lands in the owner's review queue at 2:01pm: the photo of the navy mug, with a title, alt text, tags, a category, and a short description, each one ready to read. There are three buttons. What happens when the owner taps one? That's the whole point of the system — the tagger drafts, the human decides. This post walks through the three things the owner can do — approve, edit, reject — and how the store, the draft state, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Three actions per draft: *approve* (write to the store), *edit* (fix, then approve), *reject* (flag it).
- Approve writes the listing fields to your store, or to an export sheet you import later.
- Edit opens a form pre-filled with the draft — change a word, then approve.
- Reject moves the photo to a flagged folder with a reason; nothing goes live.
- Every action is logged with who did it, when, and the before-and-after — so it's reversible.

Three actions on a draft



The tagger drafts and the human decides — approve is the only path to the store.

Fig 5. Three actions per draft, three different effects. Approve writes the fields to the store. Edit lets the owner fix a word, then writes. Reject flags the photo and writes nothing. Every action writes to the audit trail.

Action 1: approve (the most common)

The draft looks right. The owner taps *Approve*. The button submits to a Function URL Lambda — a small web endpoint that runs the approve step without any heavy web server behind it. Two things happen. First, the listing fields are written where they belong: directly to the store through its API for shops that have one connected, or appended as a row to an export sheet for shops that prefer to import a batch by hand. Which path you use is a setting; the rest is identical. Second, the draft is archived, the photo is marked done, and an `action: approved` row is written to the audit trail with the owner's name, the timestamp, and the exact fields that went out.

From the owner's side this is one tap. The photo of the navy mug is now a real listing with a clean title, proper alt text, sensible tags, the right category, and a readable description — and it took ten seconds instead of three minutes.

Action 2: edit (fix one word, then approve)

The draft is close but not quite. The model wrote "navy," but the product's actual name is "midnight." The owner taps *Edit*. A form opens, pre-filled with all five drafted fields, so the owner isn't starting from scratch — they're fixing one word in an otherwise finished draft. They change "navy" to "midnight," leave everything else, and save.

Saving runs the exact same write-to-store step as Approve, with one difference recorded in the audit row: the action is `edited`, and the before-and-after snapshot shows what the owner changed. That snapshot is quietly useful over time — if the model keeps writing “navy” where the shop says “midnight,” the pattern shows up in the edits, and that’s a hint to add the preferred word to the style doc so future drafts get it right on their own.

Action 3: reject (this one’s wrong)

Sometimes the draft is fine but the photo isn’t the one that should be listed — it’s a duplicate, a bad angle, or an item that’s being discontinued. The owner taps *Reject* and picks a reason from a short list (wrong item, bad photo, will retake, discontinued). The photo is moved to the flagged folder in S3, the same place the automatic flags from Part 4 land, and nothing is written to the store. An `action: rejected` row records the reason and who rejected it.

Reject is deliberately a clean dead end, not a deletion. The photo and its draft stay in the flagged folder, so if someone rejected the wrong card by mistake, it can be found and re-queued. Nothing is ever truly thrown away — it’s moved, with a reason, where a person can find it again.

Every action is logged, every action is reversible

The `pt-audit` table records every approve, edit, and reject with the user who acted, the timestamp, and a snapshot of the fields before and after. If a listing went out with a typo, or the wrong photo got approved, a rep can run an “undo last action” through a small admin command that reads the previous-state

snapshot and restores it — pulling the listing back or re-queuing the draft. The undo is itself an audit row, so the trail of who-changed-what stays clean.

This matters most when more than one person reviews. A small shop might have two people clearing the queue on a busy launch day. The audit trail is the shared memory: who approved which listing, what the model originally drafted, and what a human changed before it went live. When a customer later asks why a listing says what it says, the answer is one lookup away.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the one vision call per photo is the only line that really moves.

PART 6 OF 7

JUNE 6, 2026 PART 6 OF 7 · PHOTO TAGGER SERIES ~3 MIN READ

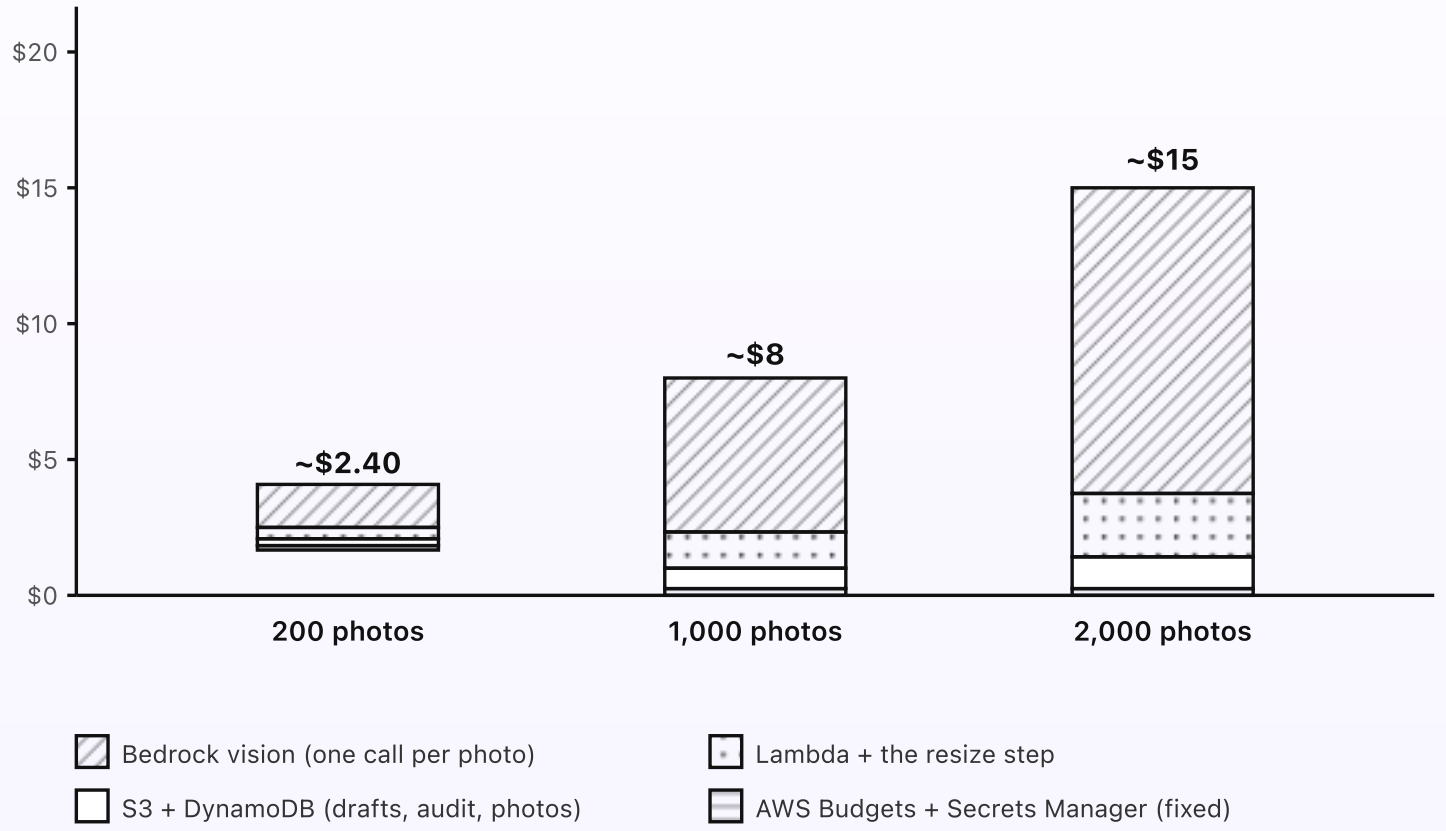
What the photo tagger costs

The photo tagger only does work when a photo lands. There's no daily tick, no always-on server, nothing humming in the background. A photo arrives, it gets resized and checked, one vision call drafts the details, and the draft waits for a human. The one line that actually moves with volume is that vision call — one per photo. At typical SMB volume the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (around 200 photos a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The one Bedrock vision call per photo is the dominant cost — everything else is pennies.
- The resize-and-check step is plain code; rejected photos never reach the paid model call.
- At 1,000 photos a month the bill is around $8. At 2,000 it's around $15.

Cost at three volumes



The vision call is the dominant cost — and even that is a fraction of a cent per photo.

Fig 6. Monthly cost at three photo volumes. The Bedrock vision call is the dominant slice because it fires once per photo and grows with volume. Lambda, the resize step, S3, and DynamoDB stay small; the fixed slice barely moves.

Where the dollars actually go

Bedrock vision (the bulk). One Claude Haiku 4.5 vision call per photo: the small resized image plus a short prompt going in, and five short fields with confidence scores coming back. That's a fraction of a cent per photo. At 200 photos a month it's a couple of dollars; at 2,000 it's the biggest single line on the bill but still in the low teens. Because the photo is resized first, the model reads a small image rather than a multi-megabyte original, which keeps each call cheap. And because the quality gate rejects bad photos before this call, you never pay to read a photo that was never going to be usable.

Lambda + the resize step. Every photo runs through a few small Lambdas: the resize-and-check intake, the reader, the approve handler. The resize step does real work (shrinking an image) so it's the heaviest, but it's still milliseconds and a small memory size. The Lambda total lands well under a dollar even at 2,000 photos.

DynamoDB on-demand. Three small tables hold draft state, acknowledgments, and the audit trail. A handful of reads and writes per photo. Pennies a month at any of these volumes.

S3 + Storage. The original photos, the small resized copies, and the flagged folder. Images add up faster than text, but at SMB volume it's still a few dollars at most, and a lifecycle rule moves older originals to cheaper storage.

SQS + SES. The queue that smooths out batch uploads costs almost nothing. SES outbound for the review email is $0.10 per thousand sent — negligible.

AWS Budgets + Secrets Manager (fixed). A couple of secrets and a budget alarm. A small flat amount that doesn't change with how many photos you tag.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the approve and edit buttons.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The system only runs when a photo lands.
- **A second model.** One vision call does the whole draft; nothing double-checks it but a human.
- **A vector store.** The tagger reads a photo and writes fields — there's nothing to search, so no embeddings, no Knowledge Base, no S3 Vectors.

How the cost scales

The bill grows almost entirely with the vision call, which fires once per photo — so cost tracks photo volume in a straight line. Lambda and DynamoDB grow with it but stay small. The fixed pieces don't move at all. So a shop tagging 5,000 photos a month lands around $35; at 10,000 it's around $70. Past those volumes you'd look at batching photos into fewer, larger calls or pre-filtering duplicates, but those are tunings for a high-volume catalog — not redesigns.

Set an AWS Budgets alarm at $25/month so anything unusual — a runaway upload loop, a misconfigured retry — pages you before the bill matters. The tagger's

normal-volume bill stays well under that ceiling.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, DynamoDB schemas, the S3 and SQS event wiring, and the Bedrock model IDs.

PART 7 OF 7

JUNE 6, 2026 PART 7 OF 7 · PHOTO TAGGER SERIES ~8 MIN READ

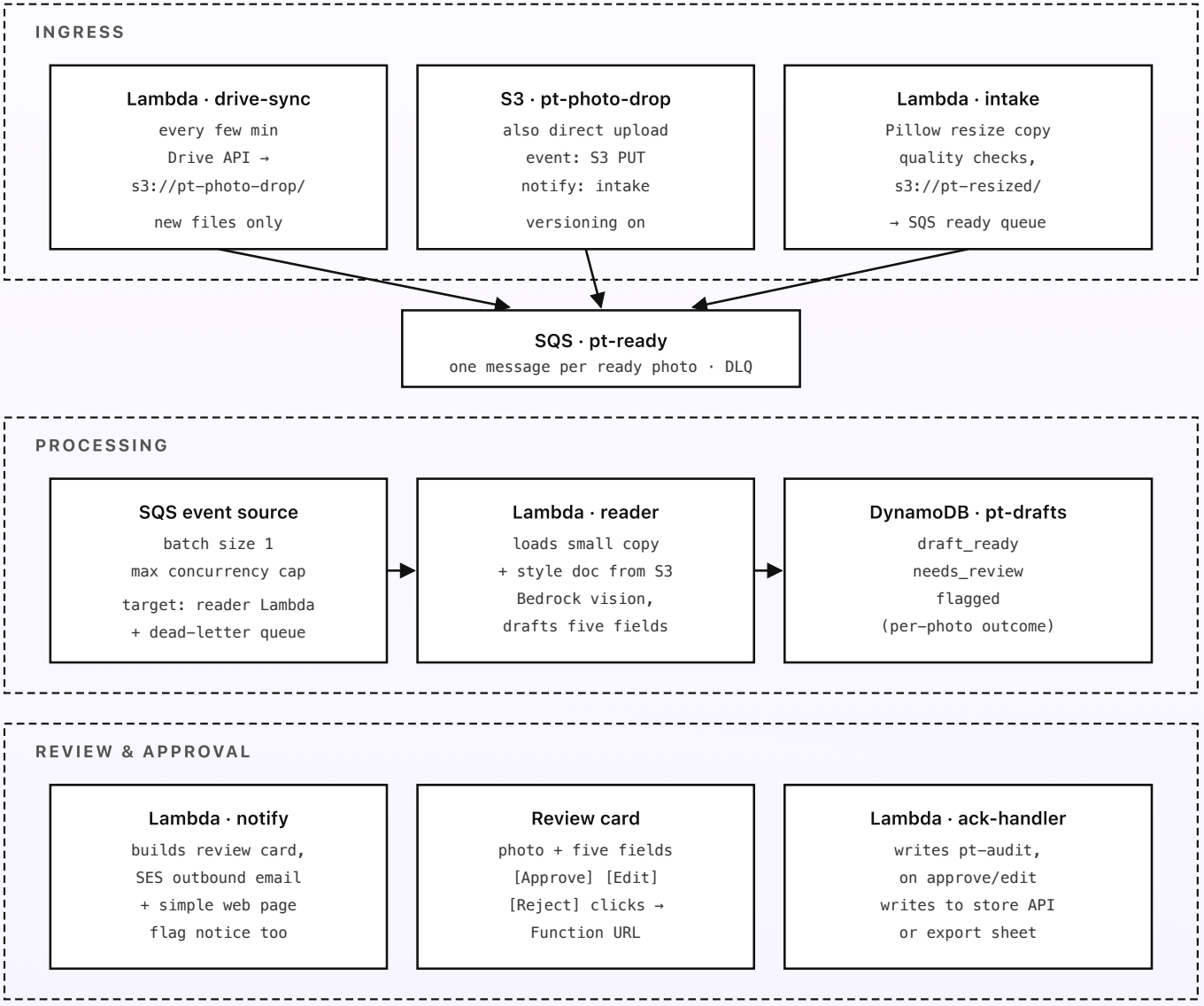
Engineering reference: the photo tagger architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the S3 and SQS event wiring, the resize step, the DynamoDB schemas, and the Function URL review flow. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). Bedrock cross-Region inference and S3 event notifications are all in good shape there, and it keeps data close for an Asia-Pacific SMB. A second region for resilience isn't worth the extra setup at this volume — the failure mode for a shop is a draft arriving a few minutes late, not a regional outage. One AWS account dedicated to the tagger (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Nothing reaches the store without a human approval — and every interaction is logged to pt-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (two lanes plus the resize-and-check step), processing (the reader draws from SQS and drafts via Bedrock vision), review and approval (the owner's decision writes to the store and 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, 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 few minutes (`rate(5 minutes)`). Uses the Google Drive API (service-account credentials in Secrets Manager under `pt/drive/sa`) to list the watched folder, diff against a small state object, and copy any new image to `s3://pt-photo-drop/<file-id>`. The same pattern syncs the style and rules docs to `s3://pt-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`intake`** — S3 PUT trigger on `s3://pt-photo-drop/`. Loads the image, resizes it with **Pillow** to a bounded max edge (e.g. 1024 px) and writes the copy to `s3://pt-resized/`. Runs deterministic quality checks — mean luminance, a Laplacian-variance sharpness estimate, pixel dimensions, and aspect ratio — against thresholds from `s3://pt-rules-source/rules.json`. On pass, enqueues a ready-photo message on the `pt-ready` SQS queue; on fail, writes a `flagged` row to `pt-drafts` with the reason and moves the original to `s3://pt-flagged/`. Pillow is the standard, stable image library in 2026 and well-maintained; if HEIC inputs from newer phones become common, add

`pillow-heif` as a decoder shim rather than swapping the library. Memory: 1024 MB (image work). Timeout: 60 s.

- **`reader`** — SQS event source on `pt-ready`, batch size 1, with a reserved/maximum-concurrency cap so a burst upload can't fan out into a Bedrock throttle. Loads the resized copy and `style.json` from S3, calls Bedrock Claude Haiku 4.5 with vision (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) using the Converse API with an image content block, and requests structured output for the five fields plus per-field confidence and a `not_a_product` flag. Writes a `pt-drafts` row with the outcome (`draft_ready`, `needs_review`, or `flagged`). On any unhandled error the message is retried, then lands on the DLQ. Memory: 512 MB. Timeout: 60 s. *The only Bedrock callsite in the system.*
- **`notify`** — DynamoDB Streams trigger on `pt-drafts` (or a small EventBridge rule on the reader's completion). For a `draft_ready` or `needs_review` row, formats a review card and emails it via SES `SendRawEmail` with links to the approve/edit/reject Function URL endpoints; for a `flagged` row, batches a short daily flag notice instead of one email per flag. Memory: 256 MB. Timeout: 30 s.
- **`ack-handler`** — Lambda Function URL, `AuthType: NONE`, with a signed-token check on every request (the token is minted into the review-card links by `notify` and verified here). Handles Approve, Edit, and Reject. On approve or edit, writes the listing fields to the store API (or appends a row to the export sheet via the Sheets API) and archives the draft; on reject, moves the photo to `s3://pt-flagged/` with the chosen reason. Writes an audit row for every action. Memory: 256 MB. Timeout: 15 s.

- `digest` — EventBridge Scheduler target, weekly. Reads `pt-drafts` and `pt-audit` for the past week; emails a short summary — photos tagged, approved, edited, rejected, and flagged — to a configured address. No Bedrock; a plain summary table. Memory: 256 MB.

Storage

- **DynamoDB** · `pt-drafts` — one row per photo. PK `photo_id`; attributes: `source` (drive/s3), `resized_key`, `outcome` (draft_ready/needs_review/flagged), the five drafted fields, per-field `confidence`, `not_a_product`, `flag_reason`. On-demand. Streams enabled for `notify`.
- **DynamoDB** · `pt-ack` — one row per review action. PK `photo_id`; sort key `ack_ts`; attributes: `action` (approved/edited/rejected), `by_user`, `reject_reason`, `store_target` (api/sheet). On-demand.
- **DynamoDB** · `pt-audit` — one row per write action of any kind. PK `(photo_id, ts)`; attributes: `action`, `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **S3** · `pt-photo-drop` — original uploads from the Drive lane and direct upload. Versioning enabled. Lifecycle to a cheaper storage class at 30 days; expiry at 2 years.
- **S3** · `pt-resized` — the small bounded copies the reader actually sends to Bedrock. Lifecycle expiry at 30 days — they're cheap to regenerate from the original if ever needed.

- **S3 · `pt-rules-source`** — mirrored `style.json` and `rules.json` from the Drive docs. Versioning enabled.
- **S3 · `pt-flagged`** — photos rejected by the quality gate, the not-a-product check, or a human Reject. Kept for review and possible re-queue.

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`. One callsite: `reader`, with a single vision request per photo via the Converse API. Claude Sonnet 4.6 (`anthropic.claude-sonnet-4-6-...`) is available as a per-photo escalation if a shop's catalog has genuinely hard images (fine print on packaging, near-identical variants), gated behind a config flag — but Haiku 4.5 handles the common case and is the default.
- **Embeddings.** Not used. The tagger reads a photo and writes fields; there's nothing to retrieve. No Knowledge Base, no S3 Vectors, no Titan embeddings.
- **Quotas.** Default account quotas are more than enough at SMB volume. The SQS concurrency cap on `reader` keeps a burst upload from spiking Bedrock requests past the per-minute limit.

Queue and event wiring

- **`pt-photo-drop` S3 notification** — `s3:ObjectCreated:*` on the bucket (or a prefix), target: `intake` Lambda. Suffix filter on common image extensions so non-image uploads are ignored.

- `pt-ready` **SQS queue** — standard queue, visibility timeout > the reader timeout, redrive policy to `pt-ready-dlq` after 3 receives. The `reader` event-source mapping uses batch size 1 and a maximum-concurrency setting.
- `pt-ready-dlq` — dead-letter queue. A CloudWatch alarm on `ApproximateNumberOfMessagesVisible > 0` pages the admin; messages are re-drivable after a fix.
- `pt-drafts` **DynamoDB Stream** — new-image view, target: `notify` Lambda, so a freshly written draft triggers the review card without polling.
- **Scheduler rules** — `pt-drive-sync` at `rate(5 minutes)` → `drive-sync`; `pt-weekly-digest` at `cron(0 18 ? * SUN *)` in TZ → `digest`.

SES and the review surface

- SES outbound for review cards and flag notices: verify a sender identity at `tagger@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request.
- The review card links carry a short-lived signed token; clicking Approve/Edit/Reject hits the `ack-handler` Function URL, which verifies the token before doing anything. The same Function URL backs a minimal web review page for clearing a batch in one place.
- No inbound SES is needed — photos arrive via Drive or S3, not email — which keeps the mail setup to a single verified outbound identity.

IAM (least privilege per Lambda)

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

- **intake role:** `s3:GetObject` on `pt-photo-drop`; `s3:PutObject` on `pt-resized` and `pt-flagged`; `s3:GetObject` on the `rules.json` key; `sqs:SendMessage` on `pt-ready`; `dynamodb:PutItem` on `pt-drafts`. *No* `bedrock:*`.
- **reader role:** `sqs:ReceiveMessage` + `DeleteMessage` on `pt-ready`; `s3:GetObject` on `pt-resized` and the style/rules keys; `bedrock:InvokeModel` on the Haiku ARN (and the Sonnet ARN if the escalation flag is enabled); `dynamodb:PutItem` on `pt-drafts`.
- **notify role:** `dynamodb:GetItem` on `pt-drafts`; stream read permissions; `ses:SendRawEmail` from the verified sender; `secretsmanager:GetSecretValue` on the token-signing secret.
- **ack-handler role:** `dynamodb:PutItem` on `pt-ack` and `pt-audit`; `dynamodb:UpdateItem` on `pt-drafts`; `s3:CopyObject` + `DeleteObject` for moving to `pt-flagged`; `secretsmanager:GetSecretValue` on the store-API and Sheets-API secrets; outbound network to the store API host and `sheets.googleapis.com`.
- **drive-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on `pt-photo-drop` and `pt-rules-source`; outbound network to `www.googleapis.com`.

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:** `pt-ready-dlq` depth > 0; reader Bedrock throttle count > 0 in 5 min (lower the concurrency cap if it fires); ack-handler token-verification failures > 5/hour (might mean the signing secret rotated).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `pt-cost-alarm` subscribed to the on-call admin's email.

Config and secrets

Service-account credentials for the Drive and Sheets APIs live in Secrets Manager under `pt/drive/sa`. The store-API key lives under `pt/store/api`; the review-link token-signing secret under `pt/token/signing`. The resized max edge, the quality thresholds, the confidence threshold, the store target (api or sheet), and the admin notify address all live in Parameter Store under `/pt/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 — building and deploying with AWS SAM. The opinionated bits: turn on S3 versioning for `pt-photo-drop` and `pt-rules-source` so a bad upload or a bad style-doc edit can be rolled back in one click, set the `reader` maximum concurrency conservatively and raise it once you've watched real burst behaviour, and keep the resize step in its own Lambda with more memory so the image work doesn't bloat the cheaper functions. Total deployable surface: around six Lambdas, three DynamoDB tables,

four S3 buckets, one SQS queue plus its DLQ, a couple of Scheduler rules, one SES sender identity, 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 shop, see Work with me.