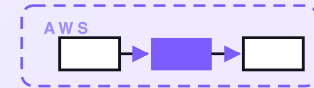
7-PART SERIES · FREE COMPANION



Product image cleaner

A shop owner photographs a new product on the kitchen counter with their phone and drops the snapshot into a folder. This is the design of a small serverless system that turns that raw photo into a full set of catalogue-ready images — background removed, auto-cropped and resized into the exact variant each sales channel needs, a subtle watermark where it's allowed, written back ready to use in seconds. The background removal runs on a container Lambda; the cropping and watermarking are plain image ops; and a low-confidence cutout is never published — it's flagged for a human to check. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

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

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/product-image-cleaner

CONTENTS

Product image cleaner

01 A product image cleaner on AWS for a few dollars a month

02 How a raw photo gets ingested

03 How a background gets removed

04 How channel variants get made

05 How a cleaned image gets published

06 What the product image cleaner costs

07 Engineering reference: the product image cleaner architecture

PART 1 OF 7

JUNE 24, 2026 PART 1 OF 7 · PRODUCT IMAGE CLEANER SERIES ~10 MIN READ

A product image cleaner on AWS for a few dollars a month

A small shop takes its own product photos, and they all start the same way: a phone snapshot on a counter or a windowsill, good enough to see the thing but nowhere near ready for a web store or a marketplace listing. Cleaning each one up by hand — cut out the background, square it off, resize it three ways, drop the logo on — is ten fiddly minutes per product that nobody enjoys. This post walks through the design of a small system that does all of it the moment a photo lands, and hands back doubtful ones instead of shipping them.

KEY TAKEAWAYS

- You drop a raw phone photo into a folder; it lands in S3, and the upload event itself starts the whole pipeline.
- Background removal runs in a container-image Lambda on arm64 with extra memory — the one heavy step, kept apart from the rest.
- Cropping, resizing, compositing onto white, and watermarking are plain deterministic image ops in Pillow.
- One photo becomes a full set of channel variants — web store, marketplace square, social — written back to S3 ready to use.
- Designed on AWS for about $3.40/month at roughly 400 images. A doubtful cutout is held for review, never published.

The whole system on one page

Before any code, here's the shape of what we're designing. A small shop takes its own product photos, and every one of them starts the same way: a snapshot on a phone, taken on a counter or against whatever wall was nearest. It's enough to see the product, and nowhere near ready to put on a web store, an Amazon listing, or an Instagram grid. The gap between "photo I took" and "image I can publish" is ten fiddly minutes of cutting out the background, squaring it off, resizing it three different ways, and dropping the logo on. The system below closes that gap automatically, and does nothing else.

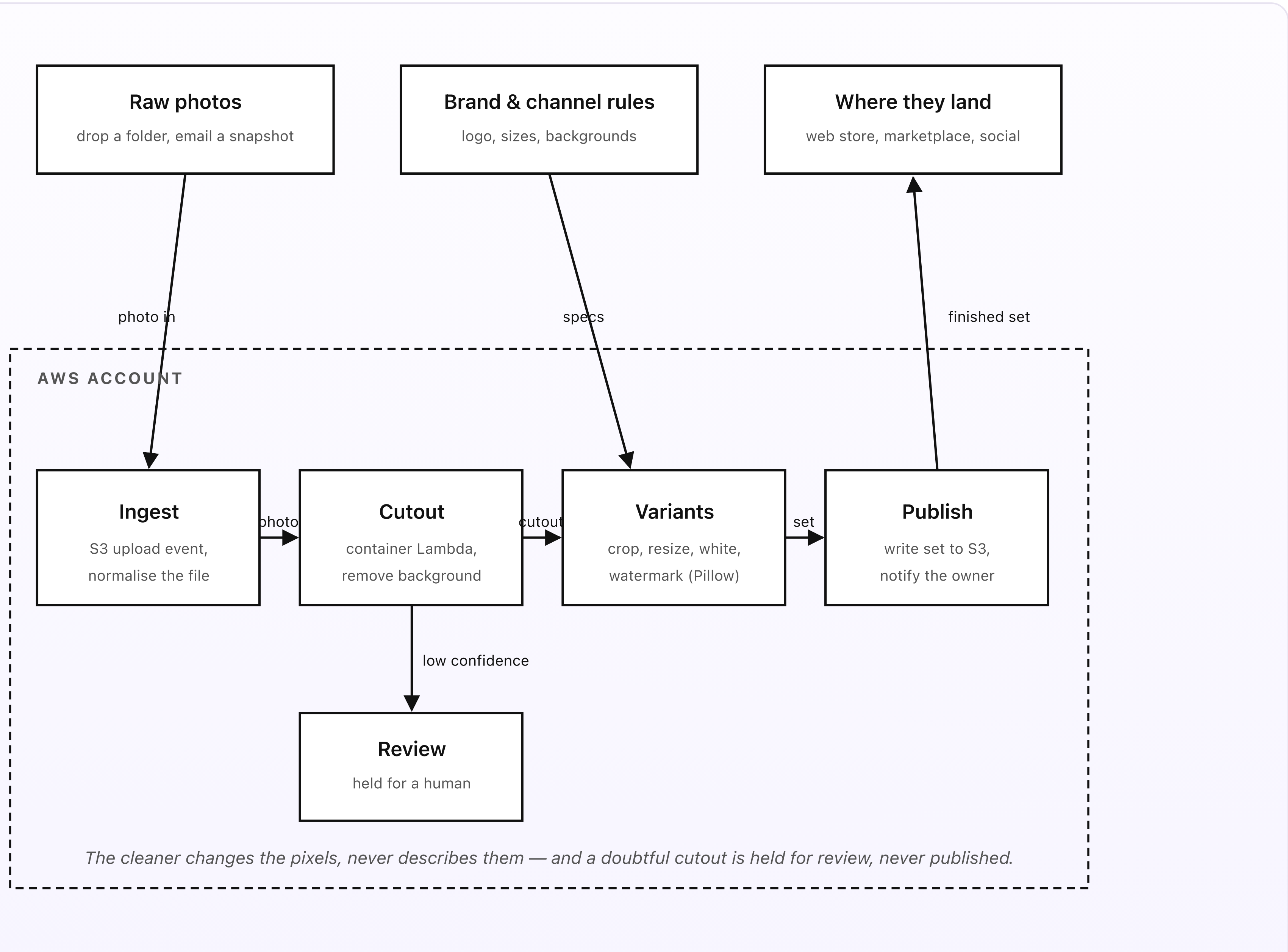

Raw photos
drop a folder, email a snapshot
Brand & channel rules
logo, sizes, backgrounds
Where they land
web store, marketplace, social
photo in
specs
finished set
AWS ACCOUNT
Ingest
S3 upload event,
normalise the file
photo
Cutout
container Lambda,
remove background
cutout
Variants
crop, resize, white,
watermark (Pillow)
set
Publish
write set to S3,
notify the owner
low confidence
Review
held for a human
The cleaner changes the pixels, never describes them — and a doubtful cutout is held for review, never published.

Fig 1. A dropped photo, an upload event, and four pieces inside AWS. The file lands in S3 and the upload starts the pipeline; a container Lambda removes the background, plain image ops cut the channel variants, and Publish writes the set back. A doubtful cutout branches to Review.

What you set up once (the outside)

- **Raw photos.** A folder you drop snapshots into — a Google Drive folder that mirrors into an S3 `inbox/` prefix, or the `inbox/` prefix itself through a simple upload link — plus an email lane for the times you'd rather just send a photo from your phone. Whichever lane a file comes in on, it ends up as one object in `inbox/`, and that single fact — an object appearing — is what starts everything. This is Part 2.
- **Brand and channel rules.** One short config you set once: your logo as a transparent PNG for watermarking, and the exact spec for each channel — pixel dimensions, background (transparent, or pure white), how much padding to leave around the product, and whether a watermark is allowed. A typical set is a 1600×1600 web-store image, a 2000×2000 white marketplace square, and a 1080×1080 social square. The marketplace variant carries no watermark on purpose, because the big marketplaces reject main images that do.
- **Where they land.** The places the finished images go and the person who uses them. The cleaner writes each variant back to S3 under a per-channel prefix and sends one notification — "Candle no. 4: 3 images ready" — with links. It never logs into your store or your marketplace account and never publishes a listing; it produces files and tells you they're ready. You stay in charge of what actually goes live.

What runs on every photo (the inside)

- **Ingest.** The photo lands in `inbox/` and the S3 upload event fires. A small Lambda checks the file really is an image, fixes the things phones get wrong — reads the EXIF rotation and turns the picture the right way up, converts HEIC to a workable format, caps the longest edge so a 12-megapixel snap doesn't cost more than it needs to — and opens a job row. This is Part 2.
- **Cutout.** The one heavy step, and the only place a model runs. A container-image Lambda on arm64 with extra memory runs a segmentation model (rembg with U^2-Net, or a hosted vision model) that decides, pixel by pixel, what is product and what is background, and composites the product onto transparency. It also scores the result — how much of the frame the product fills, how clean the edges are — so a bad cutout can be caught here. This is Part 3.
- **Variants.** With a clean cutout in hand, plain Pillow operations do the rest: trim to the product, pad it to the channel's safe area, resize to the exact pixel size, composite onto white where the channel wants white, and stamp a subtle watermark where it's allowed. No model is involved — it's deterministic arithmetic, so the same cutout always yields the same set. This is Part 4.
- **Publish.** The finished variants are written back to S3 under `out/web/`, `out/market/`, and `out/social/`, a manifest records exactly what was produced, and the owner gets a single notification. The original is never touched. This is Part 5.

In plain words

A homeware shop adds a new candle to the range. The owner stands it on the kitchen counter, takes one photo on her phone — 4,032×3,024 pixels, the toaster just visible behind it — and drops it in the Drive folder. Seconds later the upload event has fired, the container Lambda has lifted the candle cleanly off the counter, and the variant step has produced three files: a 1600×1600 web image of the candle floating on white, a 2,000-pixel marketplace square with the product filling most of the frame and no watermark, and a 1080×1080 social square with the shop's logo faint in the corner. Her phone buzzes: "Sea Salt candle: 3 images ready." The whole thing took less time than it took her to put the kettle on.

The next photo is a clear glass vase, and the cutout comes back doubtful — the model has eaten away the see-through middle, leaving a ring. The cleaner doesn't build anything from it. The job stops at *needs review* and she gets the original and the attempted cutout side by side, so she can reshoot the vase against a plain backdrop rather than discover the hole after it's live on three channels.

DESIGN RULES THAT SHAPED EVERY DECISION

- The upload *is* the trigger. A file appearing in S3 starts the pipeline; there's no button to press and nothing to poll.
- One heavy step, kept apart. Only background removal needs a big container and a model; everything else is small and cheap.
- The model produces a mask, nothing more. Cropping, sizing, and watermarking are deterministic image arithmetic.
- It changes pixels, it doesn't describe them. This is a cleaner, not a tagger — there is no caption and no metadata guesswork.
- A doubtful cutout is held, never shipped. Low-confidence results stop at review with the original attached.
- The original is sacred. Every variant is a new object; nothing overwrites the photo you dropped in.

Why this shape

Most small shops handle product photos one of three ways: they pay a studio (slow and expensive for a changing range), they wrestle each shot through a desktop editor by hand (ten minutes a product, and the results drift as whoever's doing it gets bored), or they post the raw snapshot and hope (which is why so many small listings have a kitchen counter in shot). The first doesn't scale to a shop that adds products weekly. The second is the dull, repetitive work that always slips. The third quietly costs sales, because a cluttered photo reads as a cluttered business.

The shape above takes the one snapshot you were always going to take and turns it into the full, consistent set the moment it lands — the same crop, the same backgrounds, the same watermark rule every single time. The heavy bit is isolated in its own container so the rest of the system stays small and almost free. And because a bad cutout is caught and handed back rather than published, the failure mode is “reshoot this one,” not “a vase with a hole in it is now live on Amazon.”

The next four posts walk through each piece in turn: how a raw photo gets ingested, how a background gets removed, how the channel variants get made, and how a cleaned image gets published. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 24, 2026 PART 2 OF 7 · PRODUCT IMAGE CLEANER SERIES ~7 MIN READ

How a raw photo gets ingested

Before anything can be cleaned up, the photo has to arrive and be turned into something predictable to work on. This post is about that first step alone: the lanes that get a snapshot into S3, the S3 upload event that kicks off the work, and the unglamorous normalising — rotate, convert, downscale, record — that every later stage quietly depends on.

KEY TAKEAWAYS

- Three lanes get a photo in — a watched Drive folder, an email-a-snapshot address, and a direct upload — all landing in one S3 `inbox/` prefix.
- The pipeline is started by the S3 upload event itself, routed through EventBridge to a queue. Nothing polls and nothing waits.
- Ingest fixes what phones get wrong: EXIF rotation, HEIC conversion, and a cap on the longest edge so big snaps don't cost extra.
- A job row is opened in DynamoDB the moment a file is accepted, so every photo has a status from its first second.
- Non-images and oddities are rejected at the door — the heavy cutout step never runs on a PDF or a screenshot.

Three ways in, one place they land

The point of the ingest step is to make everything that follows simple, and the way it does that is to collapse every possible way a photo might arrive into one fact: an object now exists under the `inbox/` prefix of the images bucket. How it got there doesn't matter to anything downstream.

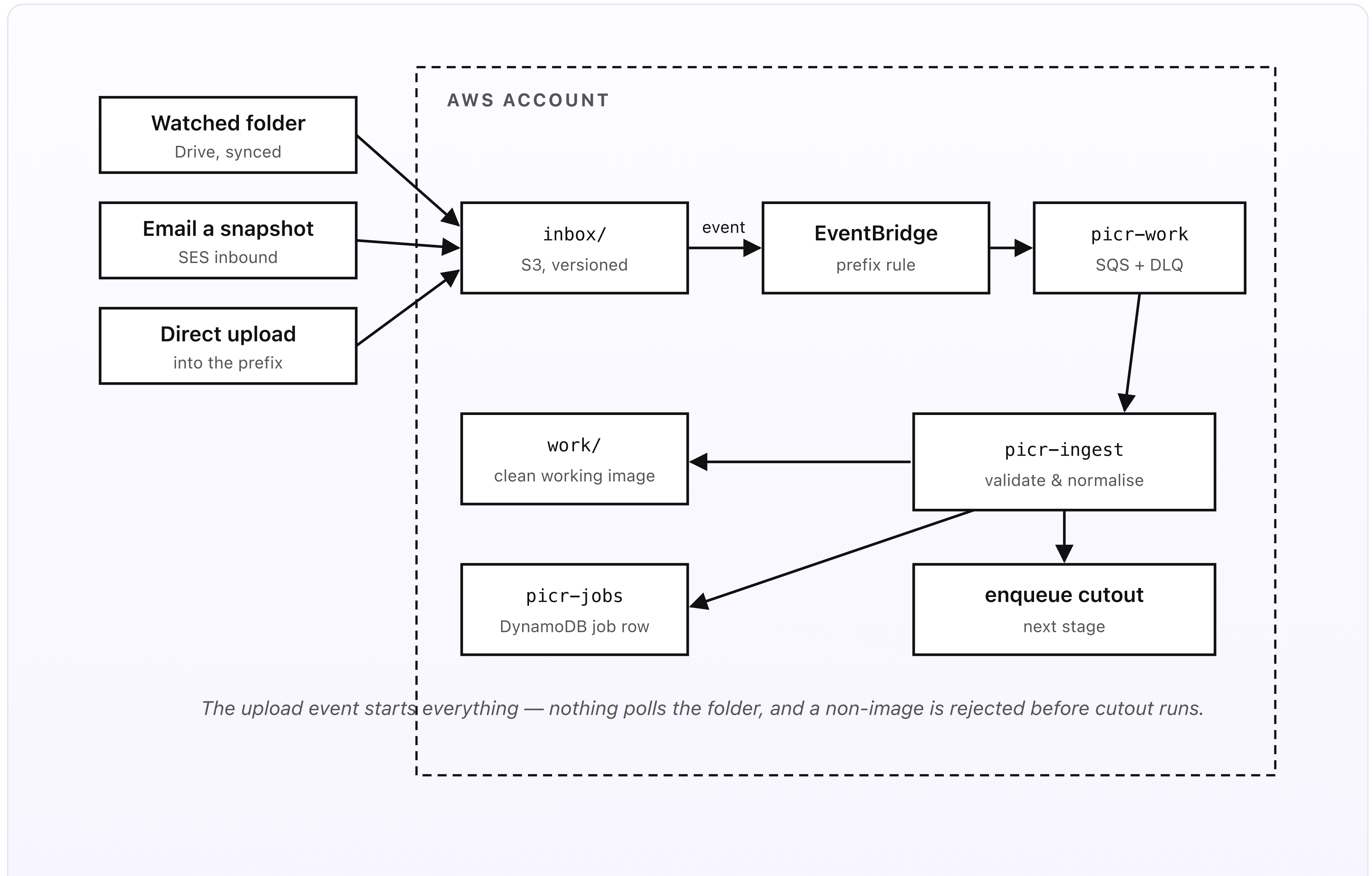
There are three lanes, chosen so the owner can use whichever is least friction in the moment. A **watched folder** — a Google Drive folder that a small scheduled function mirrors into `inbox/` every few minutes — suits a batch of photos taken at a desk. An **email lane** — forward or send a photo to a dedicated address, and

SES writes the attachment straight to `inbox/` — suits a single shot taken on the shop floor, sent before you've put the phone down. And a **direct upload** straight into the prefix covers anyone wiring this into their own tooling. Three doors, one room.

The upload event, not a poll

What turns a dropped file into work is the S3 upload event. The images bucket has event notifications switched on; when an object is created under `inbox/`, S3 emits an `Object Created` event to EventBridge. A single rule matches that prefix and forwards the event to an SQS queue, and the queue triggers the ingest Lambda. There is no schedule checking the folder, no "is anything new?" loop, and therefore no cost or delay while nothing is happening. A photo that lands at 2am is picked up at 2am; a quiet week costs nothing because nothing fires.

The queue between the event and the function is doing real work, not decoration. If someone uploads forty photos from a shoot at once, forty events arrive together; the queue absorbs the burst and lets the functions work through it at their own pace, and a file that fails to process is retried and, after a few attempts, parked in a dead-letter queue rather than lost or retried forever. The upload event is the spark; the queue is what keeps a busy afternoon from becoming a thundering herd.



The upload event starts everything — nothing polls the folder, and a non-image is rejected before cutout runs.

Fig 2. Three lanes converge on one `inbox/` prefix. The S3 upload event flows through EventBridge to a queue and triggers `picr-ingest`, which writes a clean working image, opens a job row, and enqueues the cutout step.

What normalising actually does

Phone photos are messy in predictable ways, and ingest exists to make them boring before any expensive step sees them. It does four things. It reads the EXIF *orientation* tag and physically rotates the pixels the right way up, because a photo that merely *says* it's rotated will be cut out sideways. It converts HEIC — the format iPhones default to — into a format the rest of the pipeline reads without special handling. It *caps the longest edge*, downscaling, say, a 4,032-pixel snap to around 2,000 pixels, which is more than enough for every output variant and roughly a quarter of the work for the container Lambda to chew through. And it strips the rest of the EXIF block, so the customer's GPS location doesn't quietly travel out with a published image.

The output is a single clean working image written to `work/<job_id>/source.jpg` — correctly oriented, a sensible size, no surprises — and a job row in DynamoDB keyed by the job id, opened with status `ingested`. From this point every photo in the system has an identity and a status, which is what makes the review lane and the daily sweep possible later.

What ingest refuses

The cheapest place to stop a mistake is at the door, before the one expensive step runs. Ingest validates that the object really is a raster image it can open — a JPEG,

PNG, HEIC, or WebP — and quietly rejects everything else: a PDF someone dragged in by accident, a screenshot, a zero-byte file from a failed upload, a video. A rejected file gets its job row marked `rejected` with a reason and is moved out of `inbox/` so it isn't picked up twice; the owner sees it on the next notification rather than wondering why nothing happened. Nothing that isn't a photo of a product ever reaches the container Lambda, which is exactly where you don't want to be spending money on a misfire.

DESIGN RULES FOR THE WAY IN

- One landing place. Every lane ends as an object in `inbox/`; nothing downstream knows or cares which lane it was.
- The event is the trigger. S3 → EventBridge → queue → function; no polling, no idle cost.
- A queue absorbs the burst. Forty photos at once are smoothed out, retried on failure, and dead-lettered if they keep failing.
- Normalise early. Rotate, convert, downscale, and strip EXIF before any expensive step looks at the file.
- Reject at the door. Non-images never reach the container Lambda; the costly step only ever sees a real photo.

PART 3 OF 7

JUNE 24, 2026 PART 3 OF 7 · PRODUCT IMAGE CLEANER SERIES ~8 MIN READ

How a background gets removed

This is the one genuinely heavy step in the pipeline, and the one place a model earns its keep. This post is about the cutout: why background removal lives in a container-image Lambda rather than a zip, what the segmentation model produces, and — just as important — how the system knows when a cutout came out badly and refuses to build a catalogue from it.

KEY TAKEAWAYS

- Background removal runs in a container-image Lambda because the model and its weights are far too large for a zip package.
- It runs on arm64 with extra memory — around 6 GB — because more memory also buys proportionally more CPU, and this is the one CPU-bound step.
- The model (rembg with U^2-Net, or a hosted vision model) produces a per-pixel alpha mask, then composites the product onto transparency.
- Each cutout is scored on coverage and edge cleanliness; a result outside the safe band is held for review, never sent onward.
- This is the only step that uses a model, and the only one that costs real compute — everything after it is cheap arithmetic.

Why a container Lambda, and not a zip

Every other function in this system is a small zip package: a few kilobytes of Python plus Pillow. Background removal can't be, and the reason is size. A zip-deployed Lambda is capped at 250 MB unzipped, and a background-removal model doesn't fit — the U^2-Net weights alone are tens of megabytes, the inference runtime that loads them is much larger, and together with their dependencies they blow past the limit comfortably. A container-image Lambda raises that ceiling to 10 GB, which is room to spare. So this one function is built as an image: the model weights are baked into the image at build time, so they're

already on disk when the function starts rather than downloaded on every call, and the whole thing is pushed to a private registry and deployed from there.

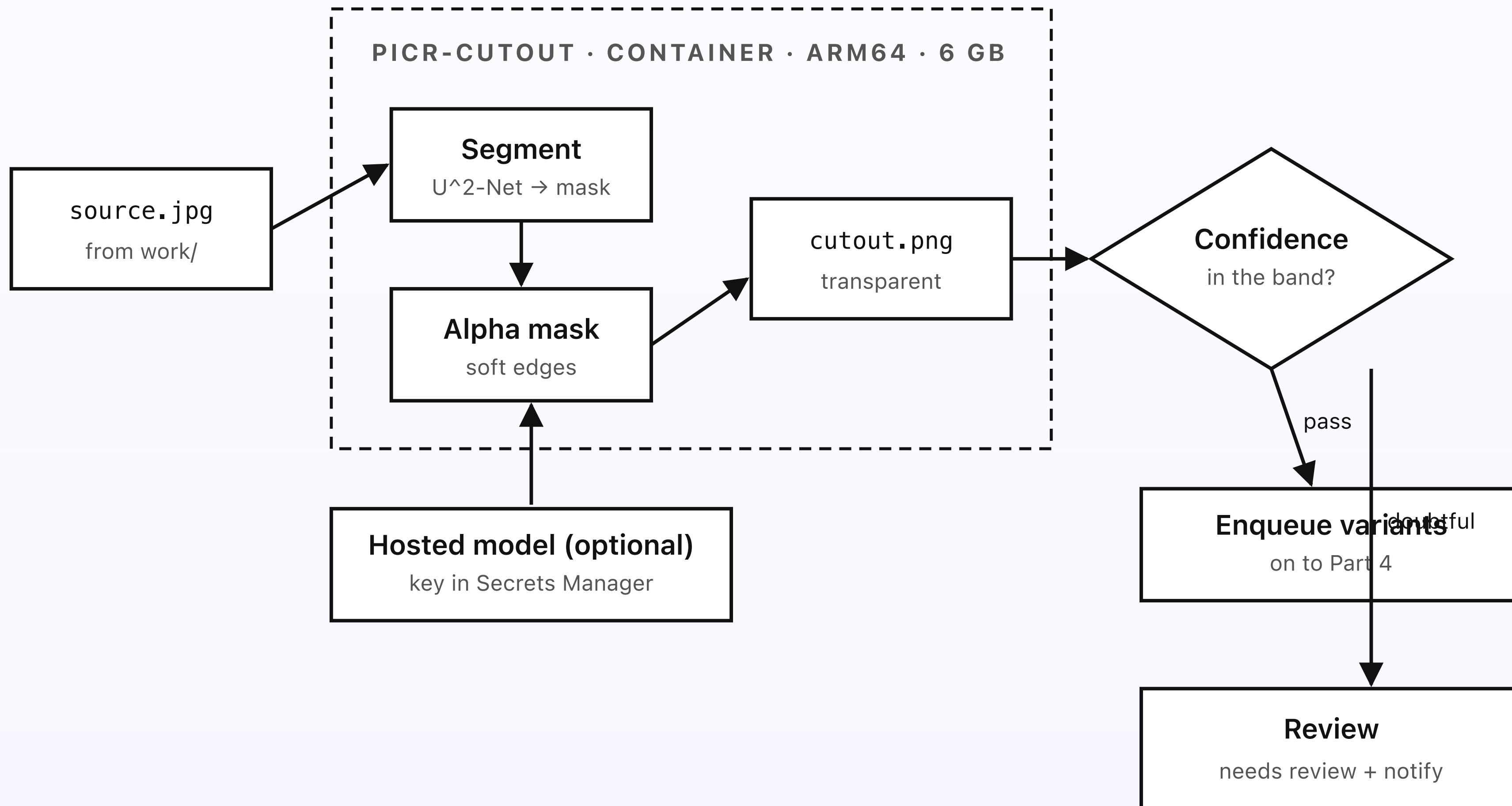
It runs on **arm64**, which is cheaper per unit of compute than x86, and with **more memory than the rest — around 6 GB**. The memory figure isn't about holding the image in RAM; it's that on Lambda, CPU is allocated in proportion to memory. Asking for 6 GB gets you several vCPUs, and since this is the one genuinely CPU-bound step in the pipeline, the extra cores cut the wall-clock time of each cutout, which is what you actually pay for. A bigger, faster function that runs for twenty seconds can cost less than a small, starved one that grinds for two minutes.

What the model actually produces

The function reads the clean working image, hands it to the segmentation model, and gets back an *alpha mask* — a greyscale image the same size as the photo, where each pixel says how much of the product is there: white for "definitely product," black for "definitely background," and the soft greys in between for the fuzzy edges, the wisps of a candle wick, the slight blur where a glass curves away. The default engine is rembg driving U^2-Net, a model built for exactly this — salient-object detection, the "what's the subject of this photo" problem — and an alpha-matting pass cleans up those soft edges so the cutout doesn't end up with a hard, scissored outline.

The mask is then used to composite: the product's pixels are kept, everything the mask calls background is made transparent, and the result is written as a 32-bit PNG — `work/<job_id>/cutout.png` — with a real alpha channel. That transparent PNG is the only thing the next stage needs; the kitchen counter, the

toaster, and the windowsill are gone. Crucially, the model's job ends here. It decides which pixels are product; it does not crop, resize, choose a channel, or know what a watermark is. One job, cleanly bounded.



A low-confidence cutout is never sent onward — it stops here, with the original attached, for a human to judge.

Fig 3. Inside `picr-cutout`: the working image becomes an alpha mask, then a transparent PNG. A confidence gate sends a clean cutout on to the variant step and a doubtful one to review. A hosted model can stand in for the local engine.

Knowing when it went wrong

A segmentation model is confident even when it's wrong, so the function doesn't ask it "are you sure?" — it measures the result instead, on cheap, deterministic signals that don't need a second model. The main one is **coverage**: what fraction of the frame the cutout keeps. A normal product photo, after trimming, leaves something between roughly 8% and 92% of the pixels as product. Near zero means the model decided almost everything was background — it ate the product. Near 100% means it kept almost everything — it failed to find a subject at all, usually a busy or low-contrast photo. Both fall outside the safe band.

The function also looks at how **fragmented** the mask is — a clean cutout is one connected blob, whereas a result speckled into many disconnected islands is the signature of the see-through-vase failure, where the model carves holes through the middle of a transparent object. When coverage is outside the band, or the mask is badly fragmented, the job is marked `needs review`, the variant step is *not* enqueued, and the owner gets a notification with the original photo and the attempted cutout side by side. The channel variants are only ever built from a cutout that passed. That single rule — never build from a cutout you don't trust — is what keeps a bad mask from becoming three published images with a hole in them.

The hosted alternative

The default is the local model, because it has no per-call fee and no third party in the loop — once the image is built, a cutout costs only the Lambda time it runs for. But the function is written so the engine is swappable: point it at a hosted background-removal API instead, with the API key held in Secrets Manager and fetched at call time, and everything around it — the normalisation, the confidence gate, the review lane — stays exactly the same. You'd reach for the hosted option if you wanted a particular vendor's quality on difficult subjects like hair or jewellery and were happy to pay per image for it. The architecture doesn't care which engine produced the mask; it only cares that a mask came back and passed the gate.

DESIGN RULES FOR THE CUTOUT

- A container because it must. The model and weights don't fit a zip; the image bakes them in so they're ready at start.
- Memory buys CPU. More memory means more cores, which means a shorter, cheaper run on the one CPU-bound step.
- The model makes a mask, full stop. No cropping, sizing, or channel logic lives here.
- Measure the result, don't trust the model's confidence. Coverage and fragmentation are cheap, deterministic gates.
- Never build from a doubtful cutout. Out-of-band results stop at review with the original attached.
- The engine is swappable. Local U^2-Net or a hosted API behind the same gate and the same key vault.

PART 4 OF 7

JUNE 24, 2026 PART 4 OF 7 · PRODUCT IMAGE CLEANER SERIES ~7 MIN READ

How channel variants get made

A clean cutout on transparency is not a catalogue. Each sales channel wants its own size, its own background, and its own rules about what's allowed in the frame. This post is about turning one cutout into the full set: the deterministic Pillow steps that crop, pad, resize, and compose every variant — and the watermark rule that exists because marketplaces reject images that carry one.

KEY TAKEAWAYS

- One trusted cutout fans out into a full set of channel variants — web store, marketplace square, and social.
- Every step is plain deterministic Pillow arithmetic: trim to the product, pad to the safe area, resize, composite, watermark.
- Cropping is driven by the alpha channel — the tight bounding box of the actual product, not the original frame.
- The marketplace variant carries no watermark on purpose, because the big marketplaces reject main images that do.
- Same cutout, same config, same output every time — no model decides a crop, a size, or where the logo goes.

From one cutout to many

The cutout that arrives from Part 3 is a single transparent PNG: the product floating on nothing, at whatever size and position it happened to sit in the original photo. That's not a catalogue, because no two channels want the same thing. A web store wants a roomy image with breathing space around the product. A marketplace wants a tight square on pure white with the product filling most of the frame and absolutely no logo. Social wants a square that looks good in a grid, branded with a faint watermark. The variant step takes the one cutout and produces all of them, and it does so with no model at all — every decision is fixed arithmetic against the config you set once.

That determinism is the whole point. The hard, uncertain judgement — what's product and what's background — was made and checked in the cutout step. Everything here is mechanical: given this cutout and this channel spec, there is exactly one correct output, and the same inputs always produce it. Run the same candle through twice and you get byte-for-byte the same web image, which is what makes a catalogue look like a catalogue instead of a scrapbook.

Crop, pad, resize

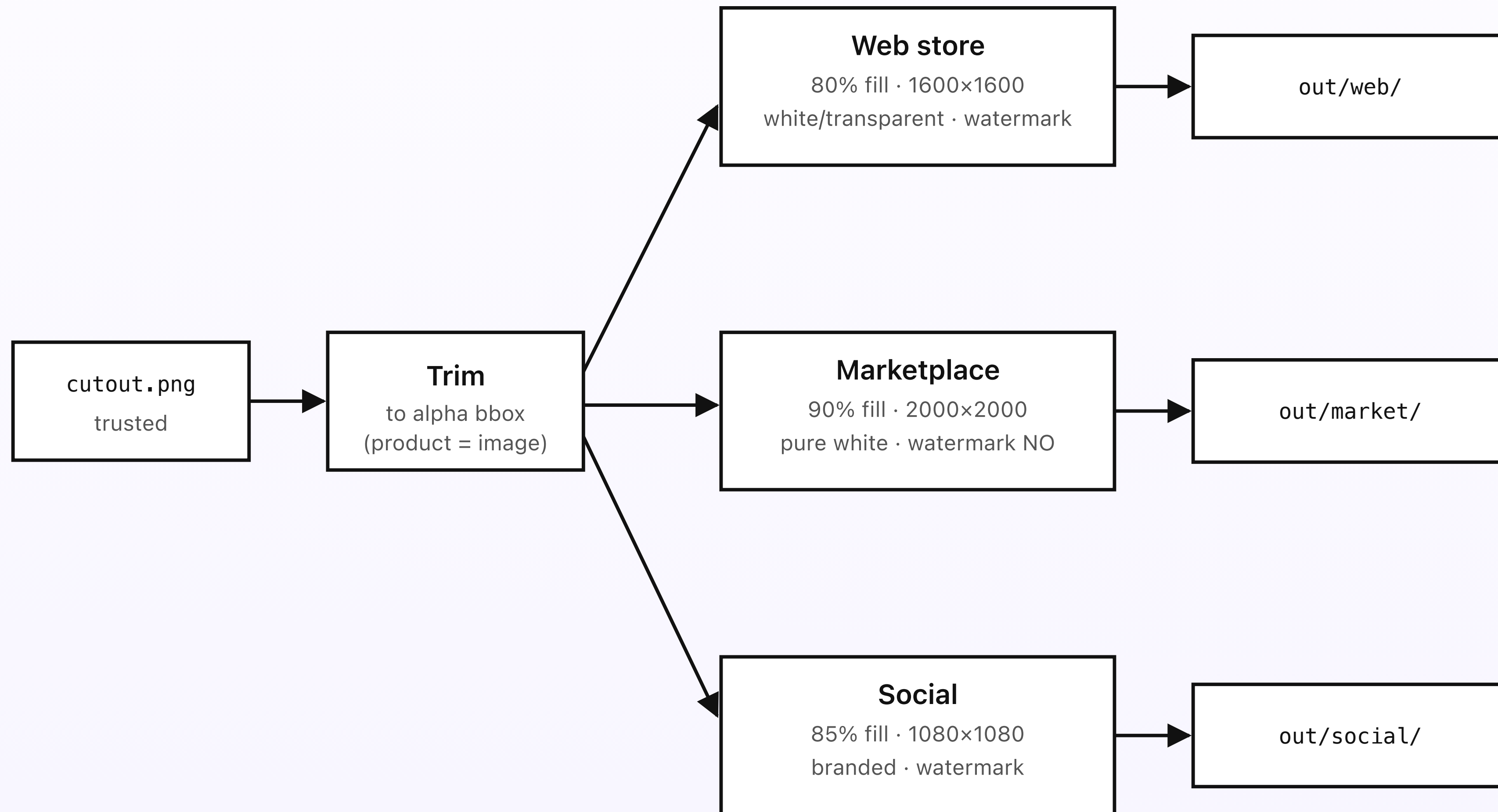
The first move is the same for every channel and it's why the cutout had to come first: **trim to the product**. The alpha channel tells you exactly which pixels are product, so the function computes the tight bounding box of everything non-transparent and discards the empty margins. Now the product is the image, regardless of where it sat in the original snap or how much counter was around it. This is the step that makes a phone photo taken from across the room and one taken up close come out looking identical.

From that trimmed product, each channel applies its own spec, all of it deterministic:

- **Pad to the safe area.** Each channel sets how much breathing space to leave — the product is scaled to fill a target fraction of the frame and centred. The web store leaves a generous margin (the product fills about 80% of the frame); the marketplace square fills more tightly (around 90%, because marketplaces want the product big); social sits in between.
- **Resize to the exact pixels.** The padded canvas is resized to the channel's precise dimensions with a high-quality filter — 1600×1600 for the web store,

2000×2000 for the marketplace, 1080×1080 for social — so every image in a listing is pixel-identical in size and the channel never has to re-scale it.

- **Composite the background.** Where the channel wants white, the product is laid over a pure `#ffffff` canvas and flattened to a JPEG; where it wants transparency, it's kept as a PNG with the alpha intact. White for the marketplace (they require it), transparency offered for the web store so it sits on any page colour.



Same cutout, same config, same output every time — and the marketplace variant never carries a watermark.

Fig 4. One trimmed cutout fans out into three variants. Each branch applies its own fill, size, background, and watermark rule deterministically. The marketplace square is the one with no watermark, by design.

The watermark rule

The watermark is the one place where a small rule prevents a real, recurring mistake. Where it's allowed, it's applied the dull, deterministic way: the logo PNG is scaled to a fixed fraction of the frame, dropped into a chosen corner at low opacity — faint enough to deter casual reuse, light enough not to fight the product — and composited over the finished image. Same logo, same corner, same opacity, every time.

The rule that matters is *where it isn't applied*. The big marketplaces — Amazon in particular — explicitly forbid watermarks, logos, and promotional text on a product's main image, and an image that breaks that rule gets the listing suppressed. So the marketplace variant is configured with watermarking off, full stop, and that isn't a toggle the system is allowed to get wrong: it's baked into the channel spec, not decided per image. The web and social variants carry the mark; the marketplace one never does. It's the same kind of guardrail as the rest of the series — the expensive mistake (a suppressed listing) is made structurally impossible, rather than left to whoever's paying attention that day.

Why keep it deterministic

It would be tempting to let a model "make it look nice" — pick a flattering crop, choose a background, place the logo where it looks best. The cost of that is

consistency: a catalogue where every image was composed slightly differently looks amateurish in a grid, and a process you can't predict is one you can't trust to run unattended. By making every step here pure arithmetic against a fixed config, the output is uniform across a hundred products and reproducible if you ever change a spec and want to re-run the lot. The model already did the one job only a model can do; this stage is deliberately boring, and boring is what scales.

DESIGN RULES FOR THE VARIANTS

- Crop from the alpha, not the frame. The product's bounding box is the image; the original margins are discarded.
- One spec per channel, set once. Fill fraction, pixel size, background, and watermark are config, not per-image choices.
- White when the channel demands white. Marketplace images are flattened onto pure `#ffffff`; the web store can keep transparency.
- No watermark on marketplace mains. The rule is structural, so a suppressed listing can't happen by accident.
- Deterministic on purpose. Same cutout and config always yield the same set, so a catalogue stays uniform and re-runnable.

PART 5 OF 7

JUNE 24, 2026 PART 5 OF 7 · PRODUCT IMAGE CLEANER SERIES ~6 MIN READ

How a cleaned image gets published

The work is done; now it has to land somewhere the owner can actually use it. This post is about the last step: where the finished variants are written, how they get back to the person who dropped the photo in, and the small guarantees — versioning, a manifest, an untouched original — that make the output safe to trust.

KEY TAKEAWAYS

- Each variant is written back to S3 under a per-channel prefix — `out/web/`, `out/market/`, `out/social/` — alongside a manifest.
- The manifest records exactly what was produced: every file, its channel, its dimensions, and the job it came from.
- One notification per job tells the owner the set is ready, with links — or that a cutout needs review, with the original attached.
- The original photo is never touched and never overwritten; every output is a new object, and the bucket is versioned.
- A daily sweep re-surfaces jobs stuck in review so a doubtful cutout doesn't quietly age out and get forgotten.

Where the finished set goes

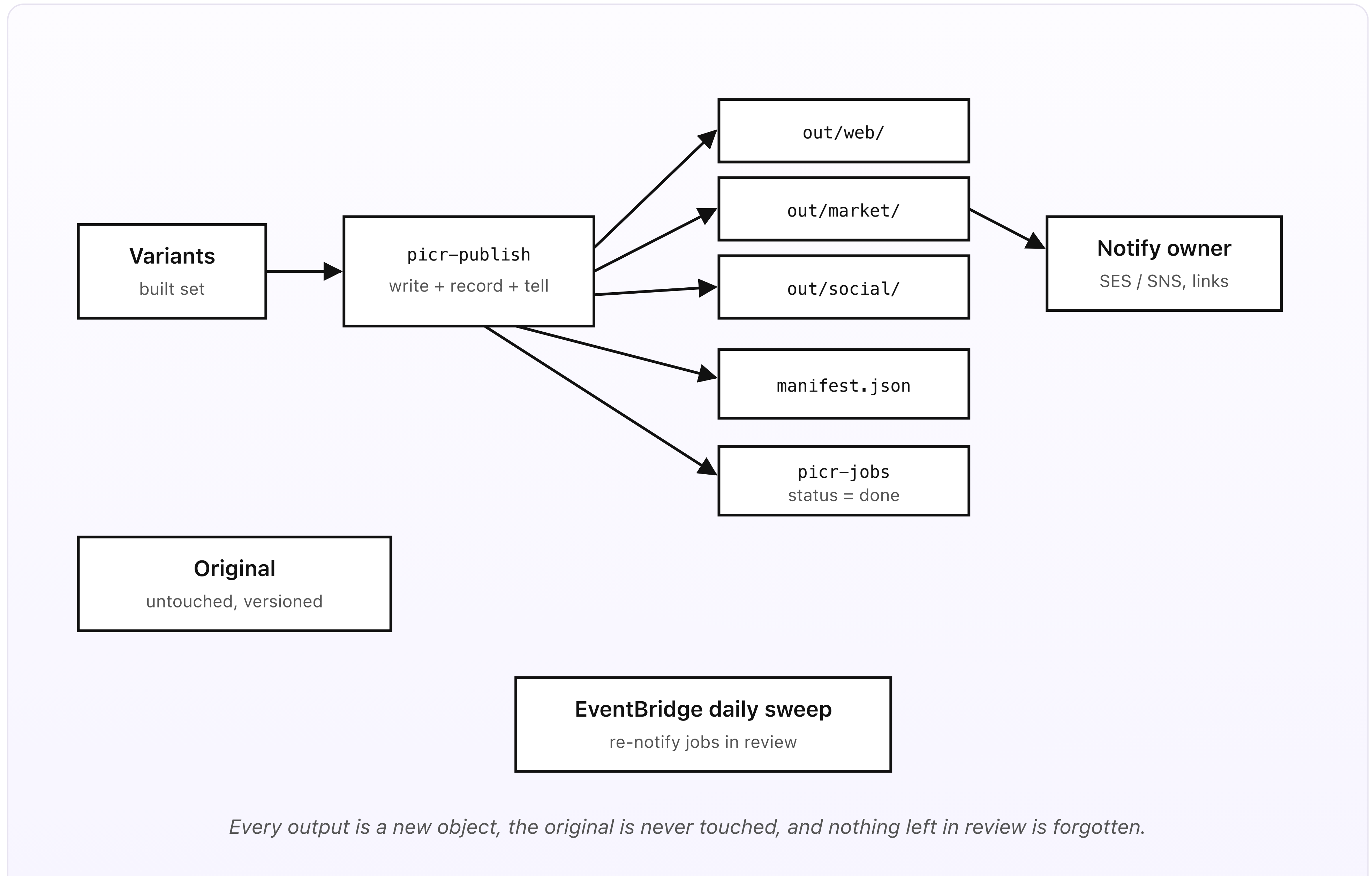
By the time publishing runs, the work is done: a clean cutout passed the gate and the variant step produced the full set. All that's left is to put the images somewhere the owner can actually reach them and to say so. The publish function writes each variant back to the same S3 bucket under a per-channel prefix — the 1600×1600 web image to `out/web/<job_id>.png`, the 2,000-pixel marketplace square to `out/market/<job_id>.jpg`, the social square to `out/social/<job_id>.jpg`. The layout is deliberately predictable: anyone, or anything, that wants the marketplace images for a batch knows to look under one prefix, and the filenames trace straight back to the job that made them.

Because the watched-folder lane runs in both directions, the same prefixes can be mirrored back into a Drive folder, so the finished images land next to where the owner dropped the original — no logging into a console, no downloading from a bucket. The system produces files and puts them where they're useful; it never logs into the store or the marketplace and never publishes a listing. What goes live is always the owner's call.

The manifest

Alongside the images, publish writes a small `manifest.json` for the job: the source filename, the job id, the cutout's confidence score, and one entry per output — its channel, its S3 key, its pixel dimensions, its format, and whether it was watermarked. The manifest is what turns a pile of files into a record. It lets the owner see at a glance that a product really did get all three variants and not two; it lets a re-run detect what already exists; and it's the thing you'd read in six months

to answer “which images did we generate for this product, and from which photo?” The job row in DynamoDB is moved to status `done` at the same moment, so the manifest on S3 and the status in the table always agree.



Every output is a new object, the original is never touched, and nothing left in review is forgotten.

Fig 5. `picr-publish` writes each variant under its channel prefix, drops a manifest, moves the job to `done`, and sends one notification. The original stays untouched in the versioned bucket; a daily sweep re-surfaces anything stuck in review.

One notification per job

The owner gets exactly one message per photo, and it says one of two things. On success: "Sea Salt candle: 3 images ready," with links to the three variants. On a held cutout: "Glass vase: needs review," with the original and the attempted cutout attached so the problem is obvious at a glance. The channel is whatever suits — an email through SES, or a push to an SNS topic the owner's phone or a team channel subscribes to. What matters is that it's one message with a clear outcome, not a stream of per-stage chatter. The owner doesn't need to know the cutout ran or the variants were resized; they need to know the set is ready, or that one needs a second look.

Why nothing overwrites the original

Two rules keep the output safe to trust. First, **the original is sacred**: the photo that landed in `inbox/` is the source of truth and is never modified. Every working file and every variant is a new object under its own key, so a re-run, a config change, or a bug can never corrupt the one thing you can't regenerate — the photo itself. Second, **the bucket is versioned**, so even an output that does get rewritten — you tweak the watermark and re-run a product — keeps its previous version rather than vanishing. If a new run produces something worse, the old version is still there to roll back to.

The last guarantee is about the things that *didn't* finish. A cutout sent to review is a job left deliberately incomplete, and the easiest way for it to go wrong is to be forgotten — the owner misses the notification, gets busy, and the half-done product quietly never makes it online. So an EventBridge daily sweep walks the job table for anything stuck in `needs review` past a day and re-notifies, the same way the other systems in this series re-surface anything left hanging. The pipeline is judged not only by the sets it ships but by its refusal to let a held one disappear.

DESIGN RULES FOR PUBLISHING

- Predictable layout. Per-channel prefixes and job-id filenames make the output easy to find and trace.
- A manifest per job. Every file, channel, size, and the cutout's confidence, recorded as the job's receipt.
- One notification, one outcome. Ready with links, or needs review with the original — never a stream of stage updates.
- The original is never touched. Outputs are new objects; the bucket is versioned so nothing is lost.
- Nothing in review ages out. A daily sweep re-surfaces held jobs until a human deals with them.

PART 6 OF 7

JUNE 24, 2026 PART 6 OF 7 · PRODUCT IMAGE CLEANER SERIES ~6 MIN READ

What the product image cleaner costs

A cleaner that costs more than the time it saves is a toy. This post is the cost breakdown: every AWS service this system touches, what each one adds up to at around 400 images a month, why the background-removal container is the single line that grows with use, and what happens to the total when the volume goes up tenfold.

KEY TAKEAWAYS

- About $3.40/month at roughly 400 images, and almost nothing runs when no photos are coming in.
- The single biggest variable line is the background-removal container Lambda, billed on memory × duration.
- The only real fixed cost is Secrets Manager, at $0.40 per secret per month for the Drive and hosted-model keys.
- Everything after the cutout — cropping, resizing, watermarking, publishing — rounds to cents, because it's plain image arithmetic.
- At ten times the volume (around 4,000 images) the bill lands near $13 — it scales with images processed, not with idle time.

Where the money goes

The cleaner is serverless end to end, so there's no instance idling overnight and no bill while the folder is empty. You pay for a photo only when a photo arrives. At a typical small-shop volume — call it 400 images a month, of which most clear cleanly and a handful go to review — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Lambda — background removal	Container image, arm64, ~6 GB, ~20–30s per image (~400)	$0.90
Secrets Manager	Two secrets — Drive sync key, hosted-model key ($0.40 each)	$0.80
S3 (versioned)	Originals plus the variant set per image, with lifecycle expiry on `work/`	$0.55
Lambda — everything else	Ingest, variants, publish, sweep — small Pillow zip functions	$0.35
CloudWatch Logs	Function logs, 7-day retention	$0.30
DynamoDB (on-demand)	Job-status table — small reads and writes per job	$0.20
SES / SNS	One ready-or-review notification per job	$0.18

AWS service	What it does here	Monthly
SQS + DLQ	Buffering between ingest, cutout, variants, and publish	$0.07
EventBridge	S3 upload events plus the daily review sweep	$0.05
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~400 images/month**	**$3.40**

The shape of that bill is the point. The one step that runs a model and chews real CPU — the cutout container — is the largest usage-priced line, and it's still only a quarter of the total. Everything that does the visible work of making catalogue images — trimming, resizing, compositing, watermarking, publishing — barely registers, because it's deterministic Pillow arithmetic running for a second or two in a small function. The biggest line on the whole table isn't compute at all; it's two secrets sitting in Secrets Manager whether or not a single photo arrives.

The container Lambda is the variable

Background removal is the only line that genuinely moves with how many photos you process, and it's worth understanding why it costs what it does. Lambda bills on gigabyte-seconds: memory allocated multiplied by the time the function runs. At around 6 GB and twenty to thirty seconds per image, each cutout is on the order of 150 GB-seconds, and arm64 prices that at a fraction of a cent — so 400

of them land near $0.90. That's also the lever you'd pull if the bill ever mattered: capping the working image's longest edge in the ingest step (Part 2) directly shortens each cutout's run, and the memory setting trades against duration, since more memory buys more CPU and a shorter run. The other functions don't move the needle — they're measured in single-digit cents no matter how busy you get.

Fig 6. The monthly bill at about 400 images. The background-removal container and two secrets are nearly half of it; everything that builds and ships the catalogue images rounds to cents.

The one real fixed cost

It's worth dwelling on 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 — nearly a quarter of the bill at this volume. One holds the Google Drive key for the watched-folder sync; the other holds the hosted-model key, and if you stick to the local U^2-Net engine you can drop that secret and shave $0.40 off. Everything else on the list is genuinely usage-priced and rounds toward zero at idle, which is exactly what you want from a system that only works when a photo lands.

One cost that is *not* an AWS line: if you choose the hosted background-removal API over the local model, that vendor charges per image, and those fees sit with them, not with AWS. The local engine is the default precisely so that a cutout costs only the Lambda seconds it runs for — no per-image third-party fee — which is what keeps the marginal cost of one more product photo down in the fraction of a cent.

What ten times the volume costs

Push this to a busy shop — 4,000 images a month, ten times the volume — and the bill lands somewhere near $13, not $34. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, EventBridge stays at a few cents, and AWS Budgets stays free. What scales is the genuinely usage-priced work — roughly $9 of cutout compute for ten times the images, a bit more S3 storage, a little more in logs, DynamoDB, and notifications. Even then, the container Lambda

is still the dominant cost, and the thing that does the visible work — the variants — stays close to free.

The honest way to read this: the AWS bill is rounding error against the alternative. Cleaning up 400 product photos a month by hand — cut out, square off, resize three ways, watermark — is a full day of dull editing every month; at 4,000 it's a part-time job nobody wants. $3.40, or even $13, buys those hours back — and the few photos that genuinely came out wrong still get a human's eye, with the original and the failed cutout already in front of them.

DESIGN RULES THAT SHAPED THE COST

- Pay per photo, not per hour. No always-on compute means no idle bill.
- One heavy line, and you can tune it. The cutout's cost is memory × duration; cap the input size to shorten the run.
- Keep the cheap work cheap. Cropping, resizing, and watermarking are deterministic ops that run for a second or two.
- Know your one fixed cost. Secrets Manager is the only line that bills while the folder is empty.
- Prefer the local model. The default cutout has no per-image fee; a hosted API trades cents per image for quality.

PART 7 OF 7

JUNE 24, 2026 PART 7 OF 7 · PRODUCT IMAGE CLEANER SERIES ~8 MIN READ

Engineering reference: the product image cleaner architecture

This is the product image cleaner with the friendly labels removed: the real function names, which one is a container image and why, the bucket prefixes and the event that fires, the job table's keys, and the IAM scope that keeps each function to its own job. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Six Lambda functions: one container image on arm64 with 6 GB for background removal, five small Python 3.14 zip functions for everything else.
- One versioned S3 bucket, `picr-images`, with EventBridge notifications on; the `inbox/` upload event is what starts the pipeline.
- One DynamoDB table, on-demand, keyed by `job_id` with a status-history sort key and a GSI on `status` for the review sweep.
- No Bedrock, no API Gateway, no NAT Gateway. The model runs inside the container (or behind a hosted-API key in Secrets Manager).
- One region, `eu-west-2`, with per-function IAM roles scoped to exactly the prefixes, queues, and secrets each one touches.

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; photos arrive through a watched-folder sync, an SES inbound rule, or a direct upload, all landing under one S3 prefix, and the bucket's upload event — routed through EventBridge to SQS — is the only thing that starts work.

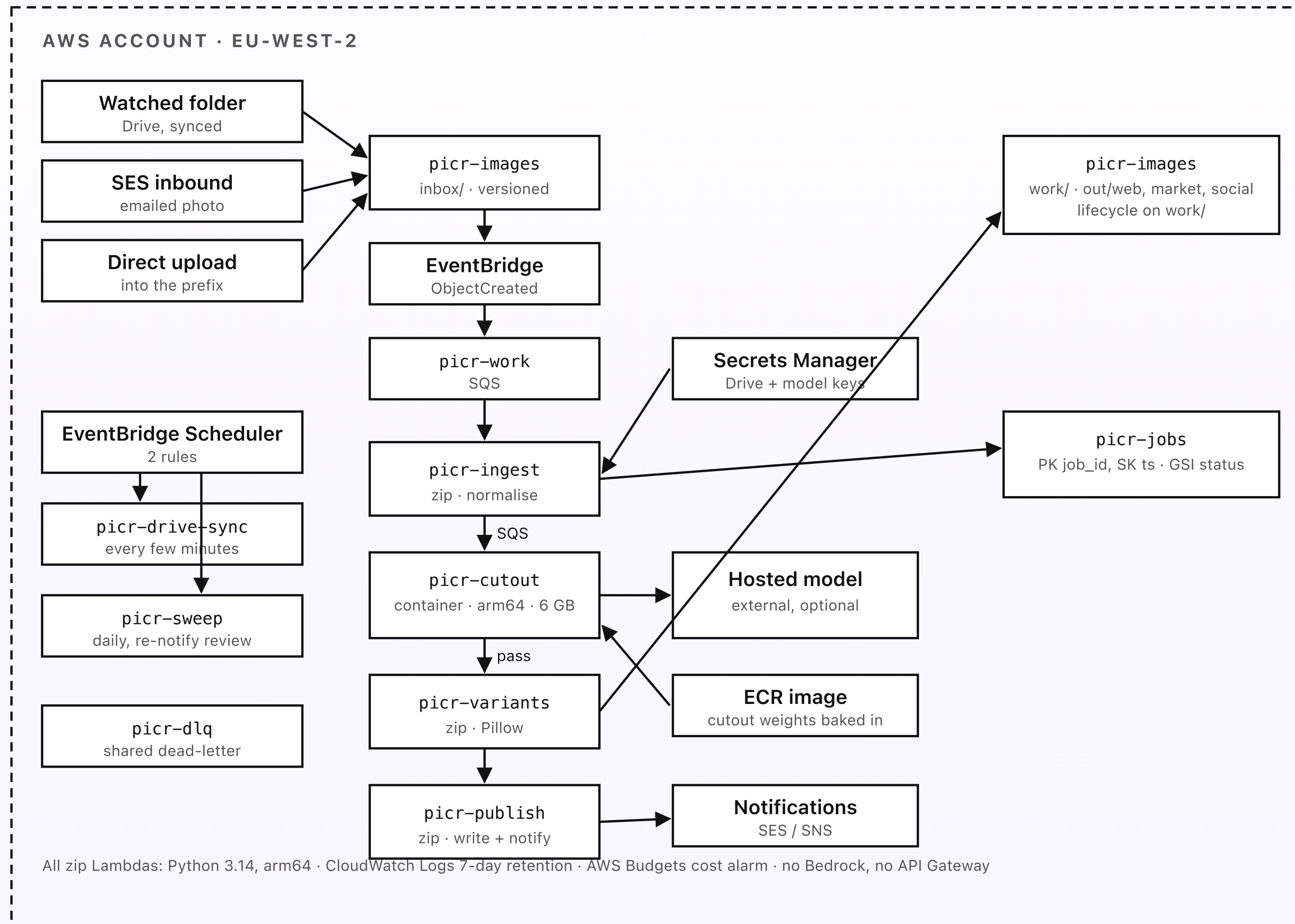

AWS ACCOUNT · EU-WEST-2
Watched folder
Drive, synced
SES inbound
emailed photo
Direct upload
into the prefix
picr-images
inbox/ · versioned
EventBridge
ObjectCreated
picr-work
SQS
picr-ingest
zip · normalise
SQS
picr-cutout
container · arm64 · 6 GB
pass
picr-variants
zip · Pillow
picr-publish
zip · write + notify
picr-images
work/ · out/web, market, social
lifecycle on work/
Secrets Manager
Drive + model keys
picr-jobs
PK job_id, SK ts · GSI status
Hosted model
external, optional
ECR image
cutout weights baked in
Notifications
SES / SNS
EventBridge Scheduler
2 rules
picr-drive-sync
every few minutes
picr-sweep
daily, re-notify review
picr-dlq
shared dead-letter
All zip Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm · no Bedrock, no API Gateway

Fig 7. The product image cleaner drawn for engineers: three inbound lanes into one versioned bucket, an upload event through EventBridge and SQS, a chain of six Lambdas (one a 6 GB container), one job table, and two scheduled jobs. One region, one account.

Lambda functions

Six functions. Five are small Python 3.14 zip packages on arm64; one — the cutout — is a container image, for the reasons in Part 3. Each does one job and hands to the next over SQS, sharing a single dead-letter queue (`picr-dlq`) after five attempts. All carry CloudWatch Logs at 7-day retention.

- `picr-ingest` — zip, arm64, 1024 MB. Triggered by `picr-work` (the upload event). Validates the file is a real image, rotates by EXIF, converts HEIC, caps the longest edge, strips metadata, writes `work/<job_id>/source.jpg`, opens the job row, and enqueues the cutout. Rejects non-images here.
- `picr-cutout` — **container image**, arm64, 6144 MB, ~120s timeout. Runs the segmentation model (rembg / U^2-Net by default), composites the transparent cutout, and scores coverage and fragmentation. On pass, enqueues variants; on fail, marks the job `needs review` and notifies. The only function that may reach an external model and the only one that reads the model secret.
- `picr-variants` — zip, arm64, 2048 MB. Trims to the alpha bounding box, then builds each channel variant (pad, resize, composite onto white, watermark where allowed) with Pillow, writes them under the `out/` prefixes, and enqueues publish.

- `picr-publish` — zip, arm64, 512 MB. Writes `manifest.json`, moves the job to `done`, and sends the single ready notification via SES or SNS.
- `picr-drive-sync` — zip, arm64, 512 MB. Scheduled. Mirrors the Drive drop-folder into `inbox/` and the `out/` prefixes back into Drive; reads the Drive key from Secrets Manager.
- `picr-sweep` — zip, arm64, 256 MB. Scheduled, daily. Queries the `status` GSI for jobs stuck in `needs review` past a day and re-notifies the owner.

Storage, events, and schedules

- **S3.** One bucket, `picr-images`, versioning on, EventBridge notifications enabled. Prefixes: `inbox/` (uploads), `work/<job_id>/` (normalised source and cutout), and `out/web/`, `out/market/`, `out/social/` (finished variants and the manifest). Lifecycle rules expire `inbox/` raw files and the `work/` intermediates after 30 days; the `out/` set is kept.
- **Events.** `ObjectCreated` on the `inbox/` prefix is delivered to EventBridge; a single rule matches the prefix and targets the `picr-work` SQS queue, which triggers `picr-ingest`. The heavier stages are chained by SQS too, with the cutout queue given a visibility timeout longer than the function's 120-second ceiling so a slow cutout isn't redelivered mid-run.
- **DynamoDB (on-demand).** One table, `picr-jobs` — PK `job_id`, SK `ts` for an append-only status history (ingested, cut, varied, published, or needs-review). Attributes: `source_key`, `status`, `coverage`, `confidence`, `variants` (list of keys), and `manifest_key`. A GSI on `status` lets `picr-sweep` find held jobs without a scan.

- **EventBridge Scheduler.** Two rules — `picr-drive-sync` on a few-minute rate, and `picr-sweep` on a daily cron (early morning).
- **Secrets Manager.** Two secrets — the Google Drive key (read by `picr-drive-sync`) and the hosted-model API key (read only by `picr-cutout`). Fetched at call time, never in env vars.
- **ECR.** Holds the `picr-cutout` container image with the model weights baked in, deployed by digest. There is no Bedrock in this system; the only model is the one inside the container or behind the hosted-API key.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `picr-ingest` can read `inbox/`, write `work/`, put to `picr-jobs`, and send to the cutout queue; it cannot read `out/` or any secret. `picr-cutout` can read and write `work/`, read only the hosted-model secret, publish a review notification, update `picr-jobs`, and send to the variants queue; it's the only role allowed to reach the internet for the optional hosted model. `picr-variants` can read `work/`, write `out/`, and enqueue publish — nothing more. `picr-publish` can read `work/` and `out/`, write the manifest, update `picr-jobs`, and send via SES/SNS, but cannot delete from any prefix. The scheduled functions hold only the narrow Drive and table permissions they need and have no inbound trigger surface at all. Everything runs in `eu-west-2`; there is no cross-Region path. An AWS Budgets alarm watches the monthly spend and notifies if it drifts above a few dollars — the early signal that an image is looping the cutout or a lane is misbehaving.

DESIGN RULES THAT SHAPED THE BUILD

- One container, by necessity. Only the cutout needs a model and the memory to run it; everything else stays a small zip.
- The upload is the only trigger. `ObjectCreated` on `inbox/` → EventBridge → SQS → ingest; nothing polls.
- One bucket, clear prefixes. Inbox, work, and per-channel out, with lifecycle on the throwaway prefixes.
- Least privilege, per role. Only the cutout reads the model secret and reaches the internet; only ingest reads `inbox/`.
- State in DynamoDB, pixels in S3. A single job table with a status GSI drives the review sweep; images never live in the table.
- One region, no model service. `eu-west-2` throughout; the model is in the container, not in Bedrock.