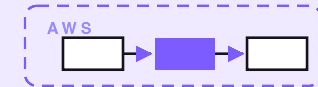
7-PART SERIES · FREE COMPANION



Quote drafter

A serverless drafter that catches every incoming RFQ from your website form, sales inbox, or direct file uploads; extracts the line items against your catalog; prices each one against your rules; drafts a complete quote with a cover paragraph in your voice; and parks it in front of a rep for one-tap approval. Drafts never auto-send. 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/quote-drafter

CONTENTS

Quote drafter

01 A quote drafter on AWS for a few dollars a month

02 How an RFQ reaches the drafter

03 How the drafter reads an RFQ

04 How a quote gets priced

05 How a draft stays honest

06 What the quote drafter costs

07 Engineering reference: the quote drafter architecture

PART 1 OF 7

MAY 4, 2026 PART 1 OF 7 · QUOTE DRAFTER SERIES ~5 MIN READ

A quote drafter on AWS for a few dollars a month

A prospect emails on Tuesday: “Need pricing on 240 of part A-12, 60 of A-12L, delivered to our Chicago site by the 30th. Can you send a quote?” By Friday afternoon nobody has replied because Sara is at a trade show, the website RFQ form went to a shared inbox nobody really owns, and the pricing sheet that has all the volume breaks lives in a folder six clicks deep. This post walks through the design of a small drafter that catches every RFQ the moment it arrives, extracts the line items, prices each one against your catalog and rules, drafts a complete quote, and parks it in front of a rep for one-tap review — never auto-sending.

KEY TAKEAWAYS

- Three outside surfaces, three AWS pieces. The drafter turns three RFQ sources into one queue.
- Every RFQ ends in one of four moves: auto-draft, clarify, out of scope, reject.
- The drafter prices only from your catalog and rules files. It never invents a SKU, a price, or a discount.
- A quote draft never auto-sends. It always waits for a rep's one-tap review.
- Designed on AWS for about $4/month at typical small-business RFQ volume.

The whole system on one page

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

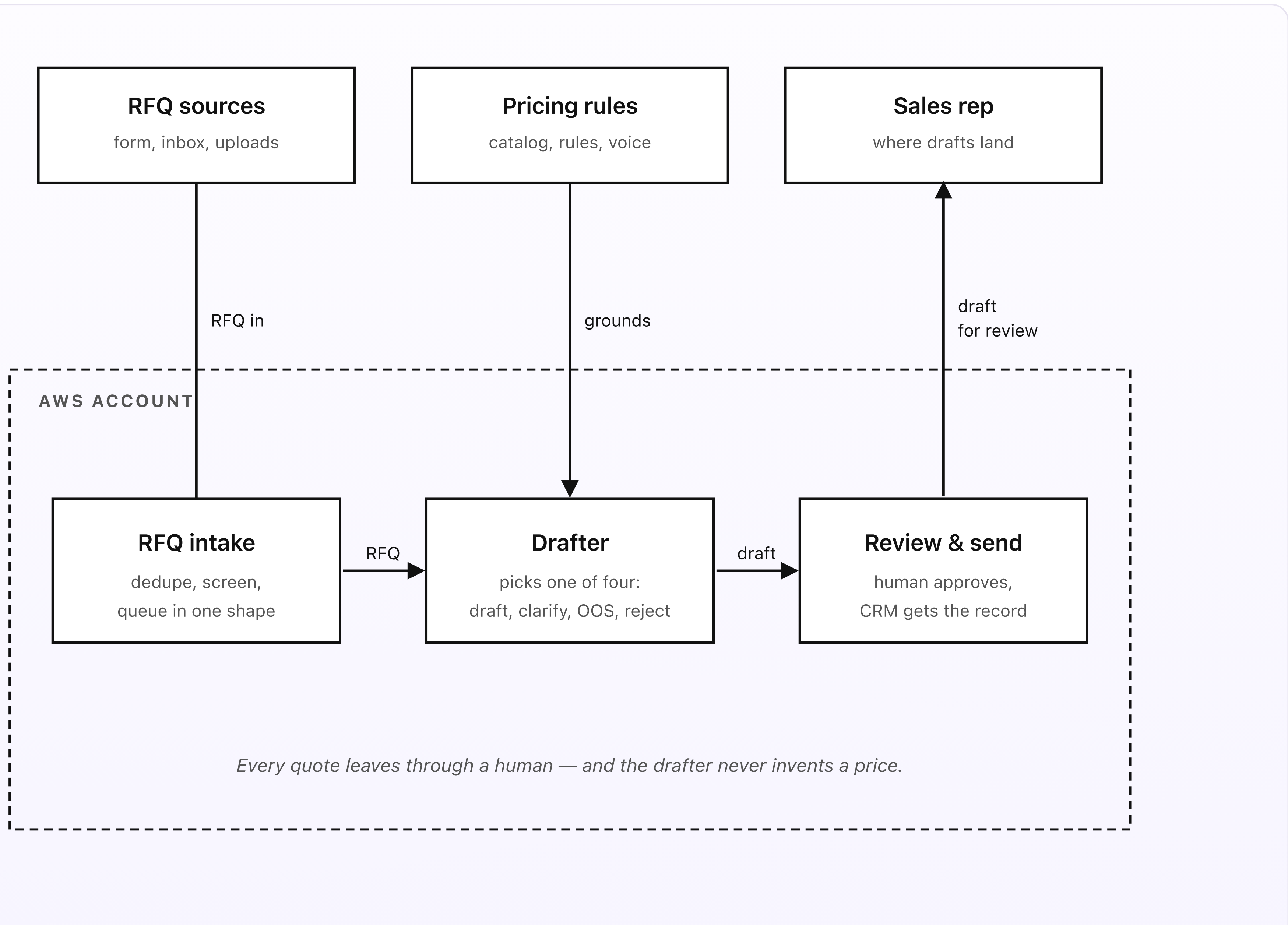

RFQ sources
form, inbox, uploads
Pricing rules
catalog, rules, voice
Sales rep
where drafts land
RFQ in
grounds
draft
for review
AWS ACCOUNT
RFQ intake
dedupe, screen,
queue in one shape
RFQ
Drafter
picks one of four:
draft, clarify, OOS, reject
draft
Review & send
human approves,
CRM gets the record
Every quote leaves through a human — and the drafter never invents a price.

Fig 1. Three outside surfaces, three pieces inside AWS. RFQs flow in from forms, the inbox, and direct uploads; the Drafter picks one of four moves; and the rep sees a complete draft within seconds — with the priced line items, the citation back to the catalog and rules, and a one-tap approve-and-send.

What you set up once (the outside)

- **Your RFQ sources.** The "Request a quote" form on your website, the shared sales inbox, and a small upload portal for prospects sending a spec sheet or tender document. The drafter takes webhooks from the form, reads inbound mail through SES, and accepts presigned uploads to S3. You can disconnect any source in one click if a campaign goes sideways.
- **A rules folder.** Three short Google Docs in a Drive folder. The *catalog* doc lists your SKUs, the plain-English names customers actually use for each one, base prices, units, minimum order sizes, and lead times. The *rules* doc covers tier discounts, volume breaks (the per-unit price drops at certain quantities), regional pricing, payment terms, the biggest discount a rep can approve without a manager, and what your team is allowed (and not allowed) to promise in a first quote. The *voice* doc has a few cover-paragraph templates, your signature, and your brand's tone. Edit any of these docs and the drafter picks up the change within minutes. No code change, no deploy.
- **A rep destination.** The Slack channel or shared review queue your team already watches. Every draft pings here with the original RFQ, the extracted line items with confidence scores, the priced quote with citations, the suggested cover paragraph, and a one-tap approve-and-send button. Out-of-scope and reject moves don't ping; they show up in the morning digest, where they belong.

What runs on every RFQ (the inside)

- **The RFQ intake.** Receives or polls each source. Drops duplicates by message-id and email-plus-subject hash, so the same prospect resending a thread twice doesn't become two drafts. Screens out the obvious junk: banned-domain spam, competitor email domains, banned-phrase floods, and submissions missing required fields. Writes the cleaned RFQ to a single queue. By the time the drafter sees an RFQ, it's in one shape regardless of source.
- **The drafter.** For every RFQ in the queue, the drafter reads the message, the form fields, and any attachments. It pulls out three things in parallel: the *line items* (what they're asking for, with quantities and variants), the *constraints* (deadline, delivery address, any payment terms they proposed), and the *context* (industry, company size, project notes). It looks up each line item in the catalog — by SKU code, but also by the plain-English name customers use. Then it picks one of four moves. With *auto-draft*, every line maps to a catalog row and the constraints are clear, so the drafter builds the quote. With *clarify*, the extraction worked but something important is ambiguous — a missing size, a wide quantity range, no deadline. The drafter writes one specific question for the customer instead of guessing. With *out of scope*, at least one line item didn't match the catalog. The drafter tags the RFQ for manual handling and pings the sales lead with what matched and what didn't. With *reject*, the message is spam, competitor fishing, or a vendor pitch. The drafter archives it with a reason. When the drafter does build a quote, every price cites the catalog row and rules passage that produced it. It never invents a SKU, a price, or a discount.
- **Review and send.** An *auto-draft* pings the on-call rep in Slack with the full package: the original RFQ, the line items, the priced quote, the cover

paragraph, and a button to approve, edit, or reject. The PDF only renders when the rep opens the draft — nothing renders for drafts that get rejected. A *clarify* writes one short question for the customer (never two) and parks the RFQ as "awaiting reply." The rep one-taps it to send. An *out-of-scope* RFQ skips the auto-draft entirely; it tags the contact in the CRM and pings the sales lead with what matched and what didn't. A *reject* archives the message with a reason. Every approved quote — regardless of size — gets written to the CRM with the conversation thread, the priced quote, and the cover paragraph. A draft that sits unactioned pings the rep again at 24 hours and escalates to the sales lead at 48 hours.

In plain words

An email lands in your shared sales inbox at 11:14am. Subject: "Quote request — 240 of A-12 plus 60 of A-12L for our Chicago facility, deadline May 30." The cloud reads it within a few seconds. The two SKUs are real; they're in the catalog. The quantities are above your tier-2 volume break, so the rules doc applies a 4% per-unit discount on A-12 and 6% on A-12L. The Chicago address is in your Midwest region, so standard shipping applies. The deadline is sixteen business days out, comfortably above your stated lead time. The cloud builds the quote: two line items with unit prices, line totals, the discount notation citing "rules.md § volume breaks," subtotal, freight, total, and a 30-day validity. It writes a one-paragraph cover note in your voice. It pings the on-call rep in Slack with the full package. The rep sees it on their phone, glances at the discount line, taps approve. The PDF goes out. The whole loop takes under two minutes. The RFQ arrived at 11:14am

Tuesday and would otherwise have sat in the shared inbox until Sara got back from the trade show on Thursday.

Total cost runs in coffee-money territory at small-business volume. Pennies per RFQ, dominated by a few model calls per draft.

DESIGN RULES THAT SHAPED EVERY DECISION

- The drafter prices only from your catalog file and your rules file. It never invents a SKU, a price, or a discount.
- Four moves, always. Auto-draft, clarify, out of scope, reject. There is no fifth.
- A draft never auto-sends. The rep's approval is the only path to the customer.
- Every priced line cites the catalog and rules passage that produced it. The rep can audit a number in one click.
- Configuration lives in Drive. Editing your prices, your discounts, or your terms doesn't need a deploy.
- Every RFQ is written to the CRM, regardless of move. Nothing falls through.

Why this shape

Most “AI quote” tools fall into one of two traps. The first kind generates beautiful-looking quotes that quietly make up prices — a 7% “loyalty discount” that doesn't

exist in your rules, or a per-unit price that's 2% off because the model rounded a number it shouldn't have. You don't notice until the customer points it out, and now you're stuck honoring a quote you didn't mean to send. The second kind needs a strict, form-based input that customers will never produce in a real RFQ. They'll always write "240 of A-12 by the 30th" in a sentence, not in a form. Neither is what you want when a real prospect writes you on Tuesday and is comparing you to two competitors who'll quote by Wednesday.

The setup above splits the difference. Messy emails, scattered specs, mixed SKU and plain-English names — all of it turns into clean, priced drafts within seconds. Every price cites the catalog row and rules passage that produced it, so a rep can check any number in one click. Ambiguous RFQs become one specific question for the customer, not a guess. Off-catalog and competitor RFQs never auto-draft; they get tagged and parked. The rep never sees a draft built from numbers nobody can trace.

The next four posts walk through each piece in turn: how an RFQ reaches the drafter, how the drafter reads it, how a quote gets priced against your rules, and how a draft stays honest on its way to the rep. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 4, 2026 PART 2 OF 7 · QUOTE DRAFTER SERIES ~4 MIN READ

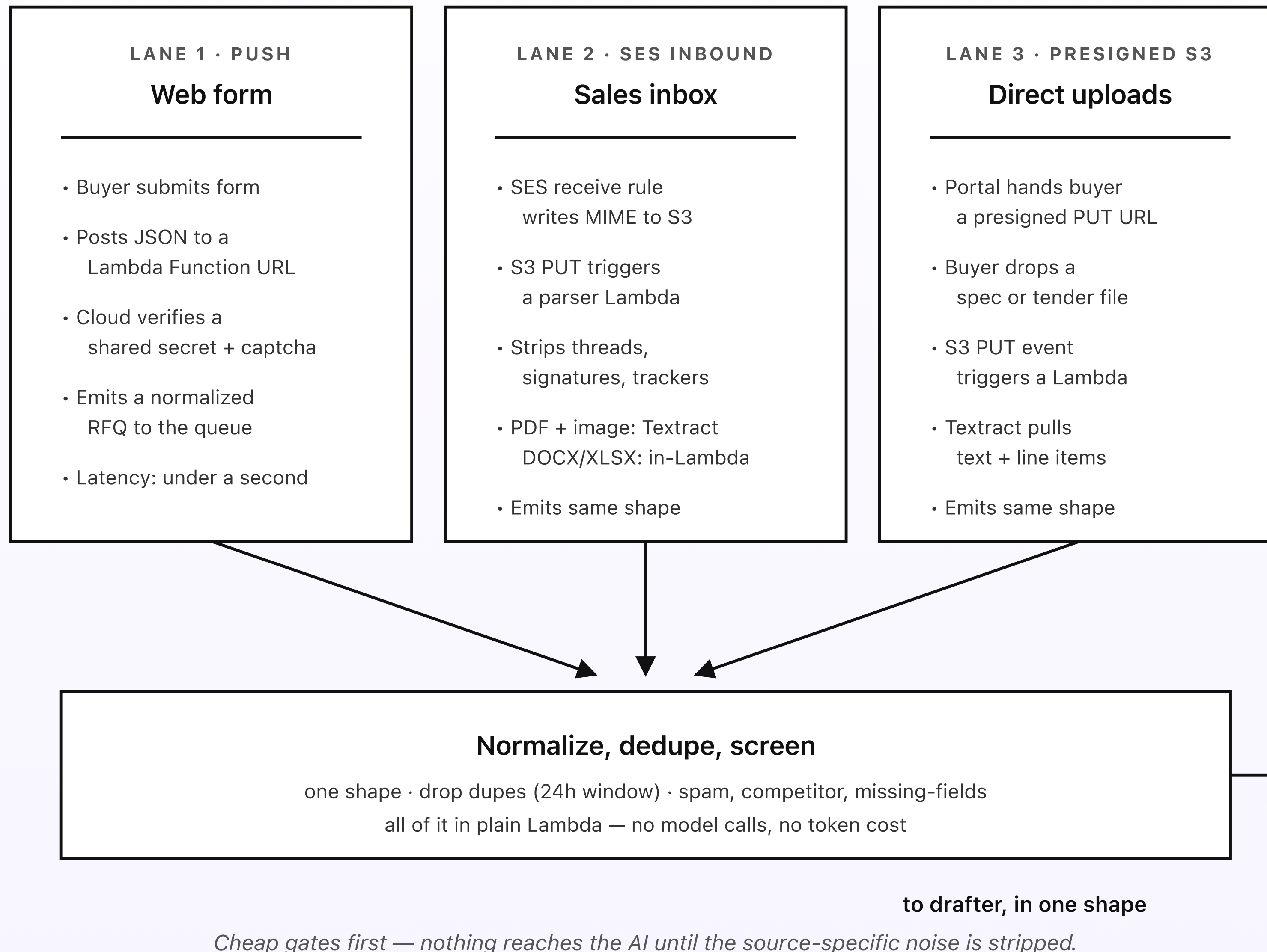
How an RFQ reaches the drafter

RFQs don’t arrive at your business through one door. The website “Request a quote” form posts to an HTTPS endpoint. The shared sales inbox sees free-text emails with the request buried in a paragraph or pasted from a spreadsheet. The biggest customers attach a tender document or a spec PDF and expect you to read it. The intake’s job is to fold all three mechanisms into one queue, drop duplicates, and screen out junk before any AI sees a single field.

KEY TAKEAWAYS

- Three intake lanes feed one queue: website RFQ form, sales inbox via SES, and direct file uploads through a presigned S3 portal.
- Push when the customer can push; presign-and-park when they have a file. Each lane has its own current-2026 reality.
- Dedupe drops the same RFQ if it arrives via two channels (email + form) within a short window.
- Screen kills banned-domain spam, competitor emails, and missing-required-field junk.
- Both filters run in plain Lambda code — no AI, no token cost.

Three lanes at the door



Cheap gates first — nothing reaches the AI until the source-specific noise is stripped.

Fig 2. Three lanes funnel into one queue. The web form pushes JSON, the inbox parses MIME via SES, and direct uploads ride a presigned S3 URL. Normalize, dedupe, and screen run in plain Lambda before any model sees a single field.

Lane 1: the web form fast path

The fastest, cheapest, cleanest lane. Your “Request a quote” form posts JSON over HTTPS straight to a Lambda Function URL — no API Gateway in the path. The Lambda checks two things before doing any real work: a per-form shared secret in the body, and a captcha token (hCaptcha or Cloudflare Turnstile, whichever you already use). If either check fails, it returns a 401 and stops. If both pass, the Lambda turns the form payload into the common RFQ shape, writes a row to DynamoDB for the audit log, and pushes a message onto SQS for the drafter. Total time: under a second from the buyer’s click to the message hitting the queue.

The form itself stays simple. Required fields: name, business email, company, line items as free text, and a deadline if known. Optional: phone, delivery address, attachments. The cloud doesn’t need a structured line-item input — the drafter handles “240 of A-12 plus 60 of A-12L” in a sentence just fine.

Lane 2: the sales inbox via SES

The lane most real RFQs actually arrive through. A buyer’s purchasing manager forwards a request from a chain of three other people. The thread is six replies deep. There’s a signature with a logo, a quoted block from the original RFP, and a PDF attachment with the actual specs. Most tools choke on a message like this. SES handles it directly.

Set the MX record on a dedicated subdomain (`quotes.your-company.com`) to AWS SES. Configure a receiving rule set with one rule: write the raw MIME to an S3 bucket and stop. The S3 PUT triggers a parser Lambda. The Lambda walks the MIME tree, picks the latest reply (stripping quoted threads and signatures using a small library — `mail-parser-reply` works well), reads any structured fields the buyer pasted in (a small block of `Field: Value` lines if your team trained customers to use them), and pulls the body text out of attachments. Textract handles PDF and image attachments; for DOCX or XLSX, the parser reads them in-Lambda using `python-docx` and `openpyxl`, since Textract only accepts PDF and image formats. Tables matter — SMB tender docs love them. The cleaned message + extracted attachment text become one normalized RFQ in the same shape as the form lane.

Latency on the inbox lane is a few seconds end-to-end; SES inbound usually delivers within five seconds, plus a Textract round trip if there's an attachment. Fast enough that a buyer who fires off a quick "Hey, can you quote 240 of A-12" gets a draft in front of a rep before they've closed their email.

Lane 3: direct uploads via a presigned S3 portal

The lane the biggest customers want. They have a tender document, a CAD-driven spec sheet, or a long Excel of part numbers. They don't want to paste it into a form and they don't want to email it because it's 4 MB and includes embedded drawings. They want a link to drop the file at.

The portal is a one-page static site. The buyer types their name and email, accepts the terms, and clicks "Get upload link." A small Lambda mints a presigned S3 PUT URL with a 30-minute TTL and a fixed key prefix per session. The buyer

drops their file straight to S3 from the browser — no file passes through Lambda or your servers. The S3 PUT event fires a parser Lambda. Textract pulls text and any structured tables; if the file matches a known template (an industry-standard tender form, say) the parser maps fields directly. The output is the same normalized RFQ shape as the other two lanes.

This lane is also where the largest RFQs come in — multi-page bids worth real money. The drafter handles them the same way it handles a one-line email; the size of the source doesn't change the shape of the queue.

Dedupe and screen, before any AI runs

The first thing the queue consumer does is run two free filters — both in plain Python, neither costs a Bedrock token.

Dedupe. The same buyer sometimes hits two lanes within a few minutes. They email the inbox, then fill out the web form because they got distracted halfway through and started over. The drafter shouldn't process that as two RFQs and produce two drafts. The intake keeps a 24-hour rolling window in DynamoDB keyed by `(email, hashed subject or line items)` and drops the second arrival as a duplicate. If a buyer sends a real follow-up — new line items, larger quantities — the hash changes and the second one passes through. The audit log records both, so a rep can find either if needed.

Screen. Before dedupe, the intake runs a quick free filter. It checks four things: a banned-domain list (throwaway addresses, vendor sales lists, your own staff testing the form), competitor email domains (you have a list, the drafter doesn't need to read the message to know), banned-phrase floods (template spam from

RFP-generator services that hit a thousand vendors at once), and submissions missing required fields (no email, no line items, no quantities). Anything caught here is archived with a reason and never reaches the drafter. The team only sees real RFQs.

Why this shape

The temptation is to build one “smart inbox” that uses an LLM to figure out, from any incoming message, whether it’s an RFQ, a follow-up, a complaint, or spam. That works on a demo and falls apart in week three. Each lane is shaped differently — a webhook is not an email is not an S3 upload — and merging them too early loses signal you can’t recover. By splitting at the door and merging at the queue, the source-specific work (signature checks, MIME parsing, attachment OCR) stays in the lane that knows how to do it. The drafter sees a clean, uniform RFQ shape and spends its model tokens on the part that’s actually hard: pulling out the line items.

Next post: how the drafter reads that uniform RFQ — the three extractors, the catalog lookup, the confidence scoring, and the move-picker that turns all of it into one of four moves.

PART 3 OF 7

MAY 4, 2026 PART 3 OF 7 · QUOTE DRAFTER SERIES ~5 MIN READ

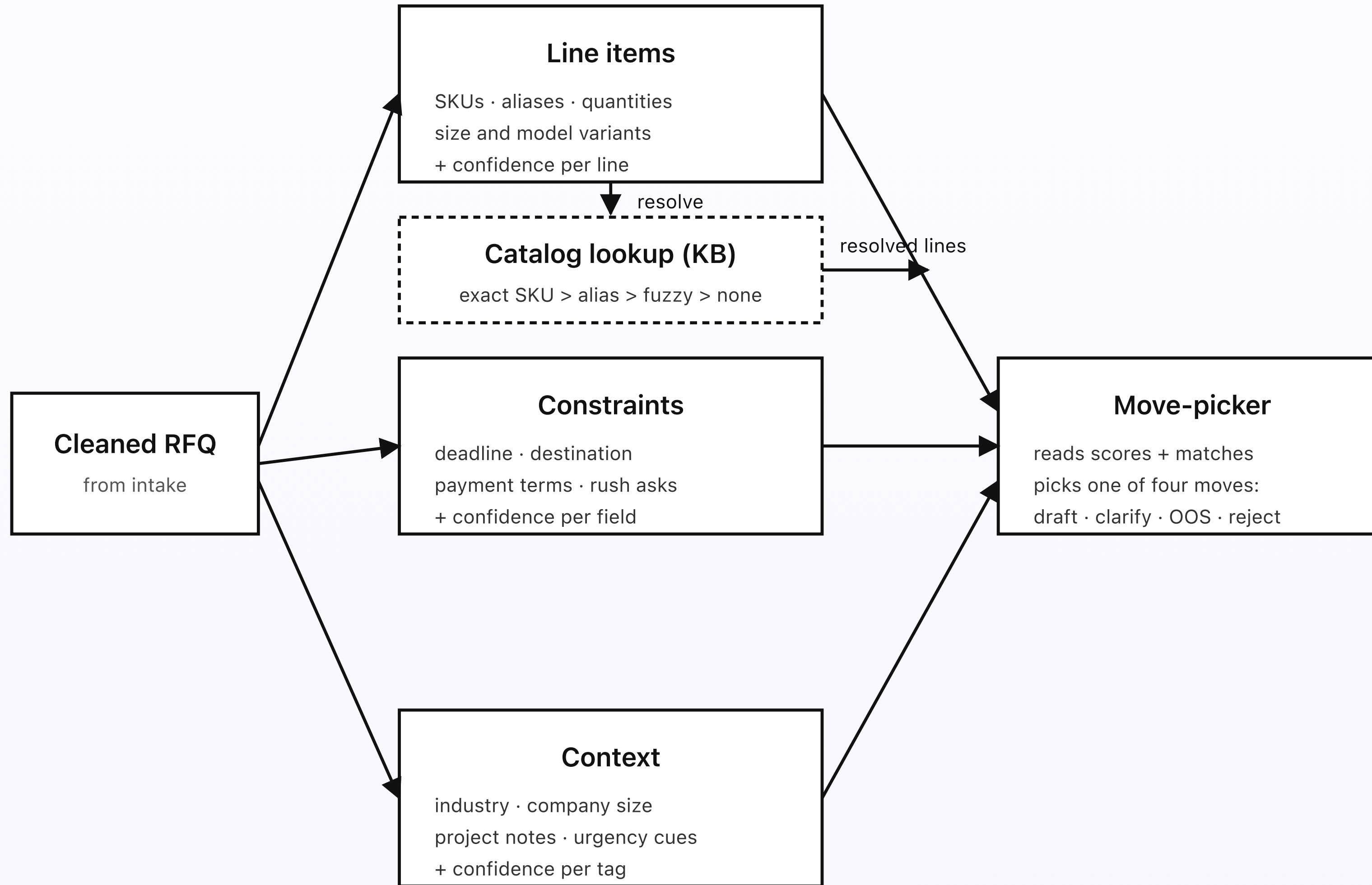
How the drafter reads an RFQ

By the time the drafter sees an RFQ it's in one shape, regardless of source. Inside that shape, though, lives the actual hard problem: turning "240 of A-12 plus 60 of A-12L for our Chicago site by the 30th, plus a couple of the long brackets if you have them in stock" into a clean list of catalog SKUs with quantities, a delivery destination, and a deadline. Three small extractors run in parallel, a catalog lookup that knows your aliases resolves each line item, and a move-picker reads it all to choose one of four moves.

KEY TAKEAWAYS

- Three extractors run in parallel against every RFQ: line items, constraints, context.
- The catalog lookup is grounded in a Bedrock Knowledge Base. Exact SKU beats alias beats fuzzy beats nothing.
- Each extracted field carries a confidence score the move-picker reads before anything else.
- Four moves, one pick per RFQ: auto-draft, clarify, out of scope, reject.
- The drafter never invents a SKU. A line item without a catalog match becomes the out-of-scope move.

The three extractors



Confidence scores travel with every field — the move-picker never reads a value without seeing how sure the extractor was.

Fig 3. Three extractors and a catalog lookup feed the move-picker. Each extractor emits its fields with confidence scores; the catalog lookup resolves line items via the Bedrock Knowledge Base; the move-picker reads the lot and chooses one of four moves.

Line items: what they're actually asking for

The hardest of the three. Customers don't write line items the way your catalog does. Your catalog has SKU `A-12`, "Bracket, 12-inch, standard." The buyer writes "240 of the standard bracket" or "the 12-inch ones" or just "A-12 x 240." Sometimes a customer who has bought from you for years writes the SKU correctly but with a typo (`A12` without the dash). Sometimes they write a quantity range ("maybe 200, definitely 150"). Sometimes they bundle ("the usual order plus 60 of the long ones").

The extractor uses Claude Haiku 4.5 with a short system prompt: "Pull out a list of `{quantity, name-or-SKU, variant-notes}` rows. One row per visible request. Mark the confidence of each based on how clearly the buyer said it. Do not invent anything that isn't in the message." The output is a list of candidate lines. Quantity ranges get a `range` field that the move-picker will treat as ambiguous. Bundles get split into their parts. The extractor never tries to match a name to a catalog row — that's the catalog lookup's job.

Constraints: the parts that aren't negotiable

Deadline, delivery address, payment terms the buyer proposed ("net 30," "COD," "PO attached"), and any custom asks: rush shipping, partial deliveries, a certificate of analysis, a special invoicing format. Each is its own typed field with a confidence. Missing fields are fine. The move-picker handles "no deadline given"

differently than "deadline next Tuesday" differently than "deadline ASAP." What the extractor never does is fill in a missing field with a guess. If the RFQ doesn't mention a destination, the extractor returns `destination: null, confidence: 1.0` — sure that nothing was said.

Context: the bits that flavor the cover paragraph

Industry, company-size cues (from the email domain plus anything in the signature or project description), urgency hints (named events like "before our trade show on the 15th" or "ahead of the audit"), and project notes (one sentence on what the order is for, when the buyer mentions it). The drafter doesn't price differently based on context. Pricing comes only from the catalog and rules. But the cover-paragraph composer (Part 5) reads the context tags to write a slightly different opening sentence for an RFQ headed to an aerospace audit versus a hardware store opening their second location.

The catalog lookup: where RAG actually earns its keep

Each candidate line item from the line-items extractor goes to the catalog lookup. The catalog lives in a Drive doc — a flat list, one row per SKU, with columns: SKU code, plain-English aliases (comma-separated, the names customers actually use), base price, unit, MOQ, lead time, and notes. A small sync Lambda copies the doc to an S3 bucket every few minutes; a Bedrock Knowledge Base indexes that bucket using Titan Text Embeddings v2 over Amazon S3 Vectors. (Bedrock KB doesn't ship a native Google Drive connector, so the one-hop-through-S3 design is intentional — and the side benefit is that S3 versioning gives you point-in-time

history of every catalog edit.) The lookup runs each candidate line item as a vector retrieval against the Knowledge Base.

The retrieval returns the top few rows. Then a strict matcher in plain Python — no model — decides what counts as a match. The order is fixed. An exact SKU match in the buyer's text wins, even if the embedding score is mediocre. A name match wins next: the buyer wrote "the standard bracket" and your catalog lists "standard bracket" as a name for A-12. A close-but-not-exact SKU match (a typo, a missing dash) wins next, but only if it's close enough by a threshold you set. Below that, the lookup marks the line as unmatched. It never invents a SKU. An unmatched line doesn't become a quote line. It becomes a flag for the move-picker.

The move-picker: four moves, always

With confidence scores from three extractors and match strengths from the catalog lookup, the move-picker decides what happens next. It's a small set of hand-written rules — not a model call. The rules are written out so the team can check any decision later.

- **Auto-draft.** Every line resolved to a catalog row at high match strength. The constraints are clear: deadline present, destination present, no unsupported custom asks. Context is fine. Move on to pricing.
- **Clarify.** Most of the RFQ looks fine but something important is unclear. A quantity range too wide to price ("200 to 500" spans two volume-break tiers, so the per-unit price changes). A SKU that resolves to two catalog rows differing only in size, with no size given. A missing deadline on a request marked "rush." The drafter writes one specific question for the customer —

never two — and parks the RFQ as "awaiting reply." If the buyer answers, the RFQ goes back through the drafter from the top.

- **Out of scope.** At least one line item didn't resolve. Or the buyer asked for something the rules doc says you don't do — "net 90" on a first order, delivery outside your service area. The drafter doesn't auto-draft. It tags the contact in the CRM, pings the sales lead in Slack with what matched and what didn't, and stops there. A human picks it up.
- **Reject.** Spam signals that slipped past the screen step, vendor pitches phrased as RFQs (the screen catches most; the move-picker catches the rest by reading the message), and competitor fishing. Archived with a reason. The team never sees it.

Next post: how an auto-draft RFQ becomes a priced quote — how the catalog and rules docs combine into one number per line, and how every applied rule cites the passage that produced it.

PART 4 OF 7

MAY 4, 2026 PART 4 OF 7 · QUOTE DRAFTER SERIES ~5 MIN READ

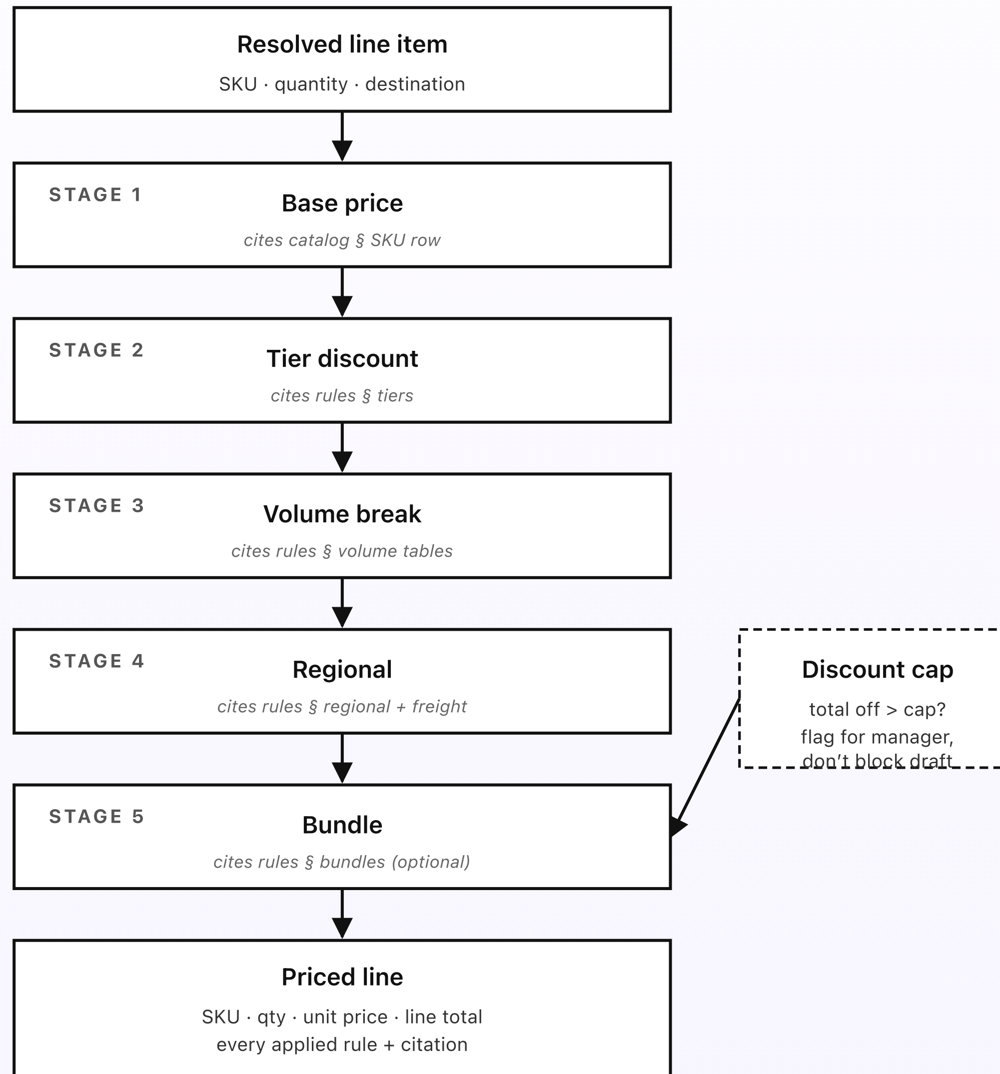
How a quote gets priced

An auto-draft RFQ has resolved line items: catalog SKUs, quantities, a destination, a deadline. Pricing is the part that's temptingly easy for an LLM to do badly. The setup here doesn't use a model for any of it. Every line runs through a five-stage pipeline of plain Python, each stage reading the catalog or the rules doc, each stage citing the passage that produced its number. By the time the cover-paragraph composer sees the priced quote, the prices are already settled and traceable.

KEY TAKEAWAYS

- Five stages run per line, in order: base price, tier discount, volume break, regional, bundle.
- None of the stages call a model. All of them read the catalog or the rules doc.
- Every applied rule writes back the section that produced it. The cover paragraph cites them.
- Discounts above a configurable cap raise a flag for the rep but don't block the draft.
- The engine never invents a number. A missing rule means no rule applied, never a guess.

The pricing pipeline, per line



Every number on the quote ties back to one passage in the catalog or rules doc — auditable in one click.

Fig 4. Five stages run per line, in order. Each reads the catalog or the rules doc, applies its rule, and writes back the section that produced its number. The discount-cap watcher flags lines for manager review without blocking the draft.

Stage 1: base price

The line arrives with a resolved SKU. The engine reads that SKU's row from a small in-memory index of the catalog — a direct lookup, no model and no vector search. The index refreshes whenever the rules folder changes. Base price comes from the row's `price` column, in the unit listed there. The citation is "catalog § *SKU*." If the row is missing a price, the engine doesn't guess. The line is flagged for manual handling and stops there. This shouldn't happen in a healthy catalog. When it does, it's usually a row someone added without filling in the price. The flag lands in the rep's queue with a message naming the row.

Stage 2: tier discount

The engine looks up the customer's tier from your CRM, keyed on the buyer's email domain. Tiers are configured in the rules doc — typical setups have three or four (new prospect, standard, preferred, contract). If the email domain doesn't match a known customer, the engine defaults to `new prospect`. The rules doc says, in plain prose, what each tier gets: "preferred customers receive a 5% discount on all standard SKUs." The engine applies that discount and writes the citation: "rules § tiers, preferred."

If the buyer's email domain is in your CRM but no tier is set on the contact, the engine treats them as standard — not as preferred. A missing tier is different from

a tier of “preferred not yet set.” Defaulting to standard keeps the engine from accidentally giving away a discount because somebody forgot to update a contact record.

Stage 3: volume break

The rules doc has a small table for each SKU that ships in tiered quantities: “A-12: 50–199 units, list price; 200–499, 4% off; 500+, 7% off.” The engine reads the table for the line’s quantity and applies the deepest break that qualifies. Whether volume discounts *stack* with the tier discount is a rules-doc decision, not an engine decision. The default in the rules doc is yes, so a preferred customer ordering 250 units gets 5% (tier) plus 4% (volume) — 9% off list in total. The citation is “rules § volume, A-12 row 2.”

If the line came through with a quantity range from the extractor (the buyer wrote “200 to 500”), the engine doesn’t price it. A range that spans two volume-break tiers is exactly the kind of thing the move-picker should have caught and turned into the clarify move. If the engine still sees a range here, it’s a bug. The line is flagged and parked.

Stage 4: regional pricing and freight

The destination decides two things: any regional adjustment to the unit price (some products cost more in regions that require compliance certification) and the freight estimate. Both come from the rules doc. The regional adjustment is a percentage on the running unit price. The freight estimate is a per-line number plus a per-shipment fixed amount. Regional rules are strict: a destination either

matches a defined region or it doesn’t. If it doesn’t, the engine keeps the standard unit price and marks freight as “needs rep confirmation” rather than guessing the cost of shipping somewhere the rules doc doesn’t know about.

Stage 5: bundles (when they apply)

Most RFQs don’t hit this stage. A few do. The rules doc’s bundles section defines combinations: “A-12 + A-12L + A-mount-bracket, any quantities ≥ 100 of A-12, the set is priced at $X per A-12 with the others included.” If the RFQ has line items that together match a bundle definition, the engine replaces those lines with one bundle row. The citation: “rules § bundles, A-12 + A-12L + A-mount-bracket.” The rep sees both the bundle row and a small note showing what individual lines went into it.

Bundles are a common source of customer confusion (“why didn’t I get the bundle price?”), so the engine writes a small note when a bundle *didn’t* apply because the quantity threshold wasn’t met — not on the customer-facing quote, but in the audit row the rep sees. If the buyer would have qualified by ordering thirty more of A-12, the rep can mention it on follow-up.

The discount-cap watcher

After all five stages, the engine adds up the total discount across all applied rules and compares it to a cap you set (often 15%, sometimes 20% for preferred customers, occasionally 0% for new prospects on first orders). If the total goes over the cap, the line gets a `requires_manager_approval = true` flag. The line still drafts. The quote still renders. The rep still sees it. But the flag is visible at the

top of the draft and on every flagged line, and the “Approve and send” button on the rep’s screen is replaced by “Send to manager for approval.” The customer never sees a quote until somebody with authority signs off.

Next post: how the draft stays honest on its way to the rep — the four guardrails that sit between the model output and the queue, and why a draft never auto-sends.

PART 5 OF 7

MAY 4, 2026 PART 5 OF 7 · QUOTE DRAFTER SERIES ~5 MIN READ

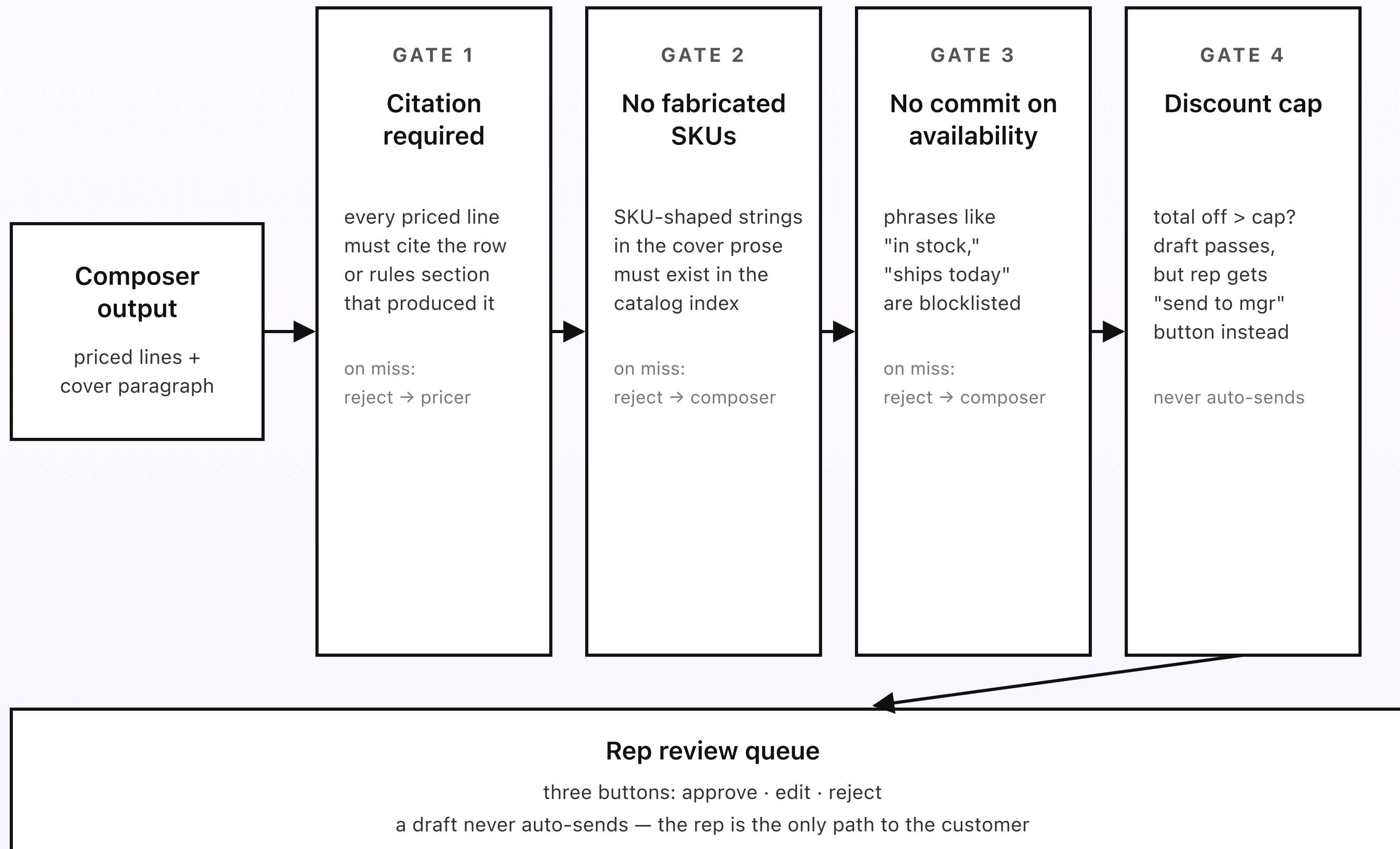
How a draft stays honest

Pricing is settled. The catalog and rules docs grounded every number. The cover paragraph is composed in your voice. Now the drafter is one model call away from rendering a PDF that will end up in front of a customer. This is the part where small errors become expensive: a fabricated SKU in the cover sentence, a casual “in stock” in a paragraph that wasn’t fact-checked, a discount that’s allowed by the rules but shouldn’t be auto-applied without a manager seeing it. Four guardrails sit between the model and the rep’s review queue. None of them are in the model. All of them run as plain code.

KEY TAKEAWAYS

- Four guardrails between the model output and the rep's queue: citation required, no fabricated SKUs, no commit on availability, discount cap.
- None of the guardrails are themselves model calls. All four run as cheap deterministic checks.
- A draft never auto-sends. The rep's approval is the only path to the customer.
- Drafts unactioned trigger a 24-hour reminder; 48-hour escalation.
- Every approval, edit, and rejection is written to the audit log for later review.

Four guardrails in a row



Every guardrail is a deterministic check, not a model call — cheap, fast, traceable.

Fig 5. Four guardrails between the composer's output and the rep's review queue. Each gate is a plain check; failures route back upstream rather than landing in front of a human. The fourth gate doesn't reject — it just changes which button the rep sees.

Gate 1: citation required on every priced line

Pricing wrote a citation for each applied rule, line by line. The first guardrail checks that work. For every line on the draft, the gate walks the applied-rules list and confirms each entry has a citation pointing into the catalog or the rules doc. If any line is missing a citation — this shouldn't happen in a healthy run, but it can if the pricer hit a bug or stopped halfway — the draft is rejected back to the pricer for a redo, with the offending line flagged. The rep is never paged on a draft with even one uncited number. Being able to check every number is what makes the rest of the system trustworthy. One uncited line breaks that trust for the whole quote.

Gate 2: no fabricated SKUs in free-text

The cover paragraph is the riskiest part of the draft. The pricer's output is structured and grounded; the cover prose is the model's creative space. Most of the time the model writes something safe: "Thanks for the request — here's our quote for the brackets you outlined; pricing reflects our preferred-customer tier." Sometimes the model gets clever and writes a SKU it shouldn't. Maybe the buyer's message mentioned a competitor's SKU and the model echoed it. Maybe the model made up something that looks like one of yours.

Gate 2 runs a strict regex against the cover prose to find anything that looks like a SKU. Your SKU pattern is small and well-defined — a letter prefix, a dash, two-to-

four digits, sometimes a letter at the end — so the regex is short. Every match is checked against the catalog index. If the model wrote “A-15” and you don’t make an A-15, the draft is rejected back to the composer with the offending text flagged. The composer rewrites the cover with explicit instruction to avoid that string. After two failed attempts, the gate gives up and falls back to a plain cover paragraph from the voice doc — safe, simple, and human-edited.

Gate 3: no commit on availability

The drafter does not check inventory. Inventory lives in the customer’s ERP or warehouse system and is its own integration. This drafter sits upstream of that. So the cover paragraph must never claim that an item is in stock, that it ships today, or any other availability promise. The voice doc tells the model not to write phrases like that. Gate 3 is the enforcement. A small block-list of phrases — “in stock,” “ships today,” “available now,” “on hand,” “ready to ship,” and a few others, all set in the rules doc — is matched against the cover prose. Any hit rejects the draft back to the composer with instruction to rewrite without that claim. The lead-time field on each line is fine; that’s a planning estimate straight from the catalog. The rejection is on customer-facing language that promises something the drafter can’t actually check.

Gate 4: discount cap (passive)

Gate 4 doesn’t reject anything. It reads the `requires_manager_approval` flags the pricer set on lines whose total discount exceeded the cap. If any line carries the flag, the draft is allowed through to the queue, but the rep’s review surface

displays it differently: a red strip across the top of the draft (“manager approval required”), a per-line discount summary, and the “Approve and send” button replaced with “Send to manager for approval.” A manager who clicks that button gets the same review surface and can either approve-and-send or reject with a note. Either action is logged.

The rep can also still edit the line, change the quantity (which might drop it below the discount cap), and re-render the draft — the pipeline runs again from Stage 4 onward and Gate 4 re-evaluates.

The send loop: a draft never auto-sends

Once the four gates pass, the draft lands in the rep’s review queue with three buttons. *Approve* renders the PDF, sends it to the customer, and writes the conversation thread plus a link to the PDF into the CRM. *Edit* opens the draft for line-level changes; saving runs the gates again. *Reject* archives the draft with a reason and an optional note for the audit log. On approve, the customer gets the quote within a couple of seconds — the PDF is rendered on demand by a small Lambda when the rep clicks, not pre-rendered for every draft. Reject is for drafts the rep doesn’t want to send: the buyer changed their mind in a follow-up, the RFQ turned out to be from a customer the rep wants to call directly, or the line items are right but the rep wants to write a different cover paragraph.

If a draft sits without action for 24 hours, the system pings the rep again with a reminder. At 48 hours it escalates to the sales lead. Weekends still count; the timing is forgiving but the alert chain matters because RFQs are time-sensitive. A

draft that nobody reviews for three days is a draft the customer probably gave up on.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume, and Part 6 explains exactly where the dollars go.

PART 6 OF 7

MAY 4, 2026 PART 6 OF 7 · QUOTE DRAFTER SERIES ~3 MIN READ

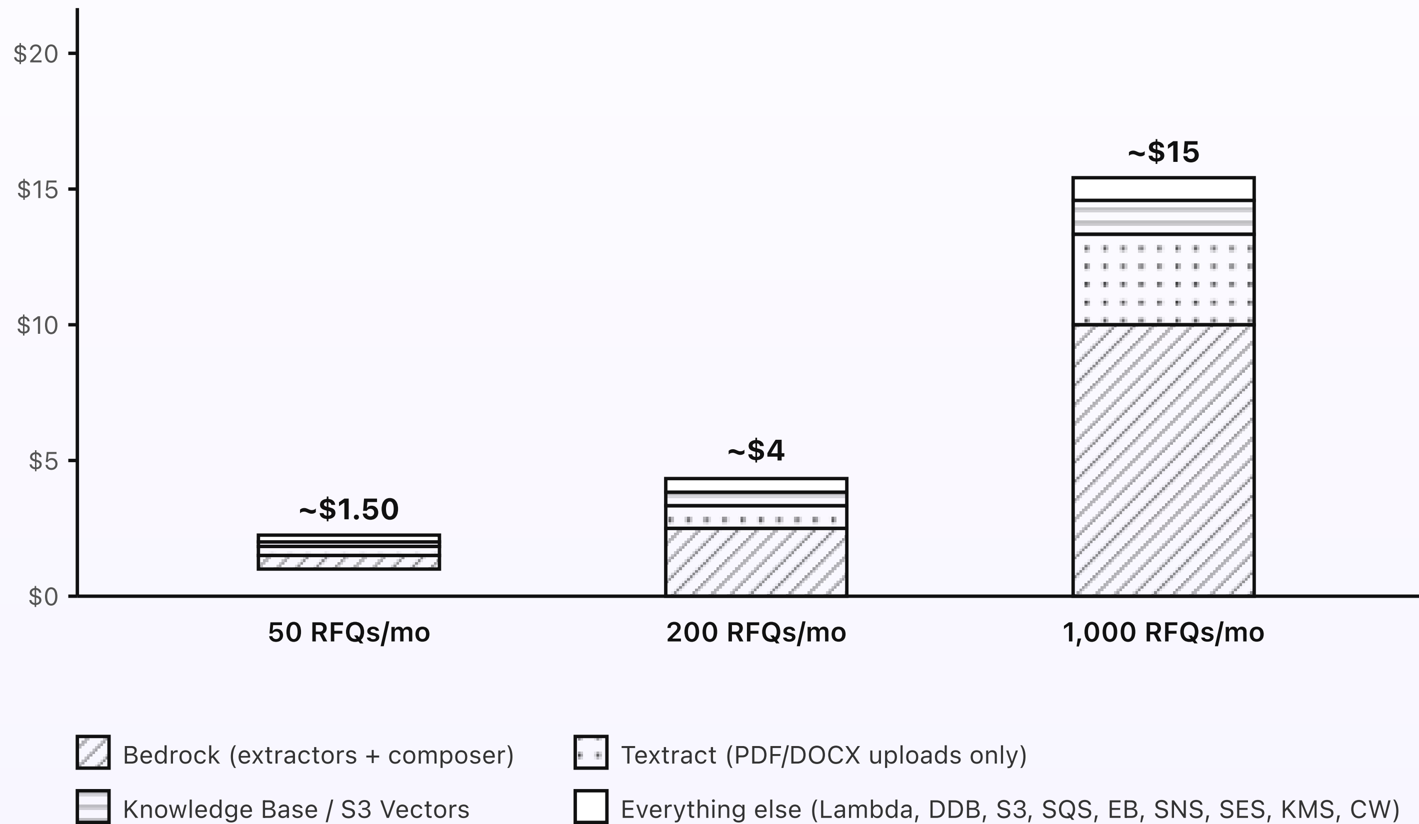
What the quote drafter costs

The whole pipeline runs in coffee-money territory at SMB volume. The fixed cost of having it sitting idle is essentially zero — quiet weeks bill nothing. The variable cost is pennies per RFQ, dominated by Bedrock tokens for the extractors and the cover-paragraph composer. PDF rendering only happens when the rep opens a draft. Three numbers below; the shape of the bill is the same at every volume.

KEY TAKEAWAYS

- Around $4/month at typical SMB RFQ volume (around 200/month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- Variable cost is pennies per RFQ — mostly Bedrock tokens.
- Textract on uploaded PDFs is the second-biggest line item; only the upload lane triggers it.
- At 1,000 RFQs/month the bill is around $15. The shape doesn't break.

Cost at three volumes



Fixed cost is essentially zero — the chart shows the full bill, not just variable.

Fig 6. Monthly cost at three RFQ volumes. Bedrock dominates at every volume; Textract is the second line item, but only fires on PDF and image uploads. The shape is linear — doubling RFQs roughly doubles the bill.

Where the dollars actually go

Bedrock (the bulk). Each RFQ runs three small extractor calls (line items, constraints, context) and one composer call (the cover paragraph). On Claude Haiku 4.5 via Global cross-Region inference, each call uses roughly a few thousand input tokens (the RFQ + a small system prompt + retrieved catalog rows for the line items extractor) and emits a few hundred output tokens. At Haiku's pricing, that's roughly a penny per RFQ on average. Volume scales linearly: 50 RFQs ≈ $0.50 in Bedrock alone, 200 ≈ $2, 1,000 ≈ $10. The clarify and out-of-scope moves use fewer model calls and cost less.

Textract (only when an upload lane fires). The web form and inbox lanes don't pay Textract unless an inbound email has a PDF or image attached. Textract reads PDF, PNG, JPEG, and TIFF; DOCX and XLSX are read in-Lambda with `python-docx` and `openpyxl`, which don't cost extra. Textract's pricing is per-page; a typical RFQ attachment is one to three pages, so each upload-lane RFQ costs a few cents. At 200 RFQs/month with maybe a quarter coming through uploads, Textract is around $0.80 to $1. At 1,000 RFQs the same share lands around $4. If your customer base skews heavily toward big tenders with multi-page spec docs, Textract becomes a bigger slice.

Knowledge Base on S3 Vectors. Storage is cheap; query fees scale with retrieval count. The line-items extractor queries the catalog Knowledge Base once per RFQ (with a small batch of candidate line items in the same query). At 200 RFQs/month, query costs are well under $0.50. Embedding ingestion only happens when the catalog or rules doc changes; for a stable catalog, that's near-free.

Everything else. Lambda runtime: a few hundred milliseconds per RFQ across the intake, drafter, pricer, composer, and dispatch functions. With Lambda Function URLs in front (no API Gateway), there's no per-request fee on the webhook layer. DynamoDB on-demand for the audit and dedupe tables: pennies a month at any of these volumes. SES inbound: $0.10 per thousand received emails. SQS, EventBridge, SNS, Secrets Manager: in total under a dollar a month. CloudWatch Logs at 7-day retention is the largest piece in this bucket, and even it lands well under a dollar at 200 RFQs/month.

What doesn't cost money

The pieces a more conventional setup would charge for — that this design avoids:

- **API Gateway.** Replaced by Lambda Function URLs. The webhook layer doesn't have a per-request fee.
- **NAT Gateway.** Nothing in the system needs outbound internet from a VPC, so no NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate task running 24/7. Quiet weekends bill nothing.
- **Pre-rendered PDFs.** Drafts that nobody opens never render. PDF generation is on-demand in Lambda when the rep clicks.
- **A scheduler.** The 24-hour reminder and 48-hour escalation run via EventBridge Scheduler one-off rules, billed per invocation — cents at SMB volume.

How the cost scales

The variable line items (Bedrock, Textract) grow with RFQ count. The fixed line items (KB storage, CloudWatch) stay nearly flat. Doubling RFQ volume roughly doubles the bill. Cutting volume in half nearly halves it. The system has no "break" volume where costs jump — you don't suddenly pay for a load balancer or a bigger instance class. The bill at 5,000 RFQs/month is around $70; at 10,000 it's around $140. Past that volume, the conversation changes — you might fine-tune a smaller model to bring per-RFQ cost down — but that's a tuning step, not a redesign.

Set an AWS Budgets alarm at $25/month so anything unusual (a runaway loop, a stuck escalator, a flood of regenerated drafts) pages you before the bill matters. The drafter's normal-volume bill stays well under that ceiling.

Last post in the series: the engineering reference. Same system, drawn purely for engineers — service names, IAM roles, Bedrock model IDs, the SES receiving rule set, the presigned-upload flow, and where each Lambda lives.

PART 7 OF 7

MAY 4, 2026 PART 7 OF 7 · QUOTE DRAFTER SERIES ~9 MIN READ

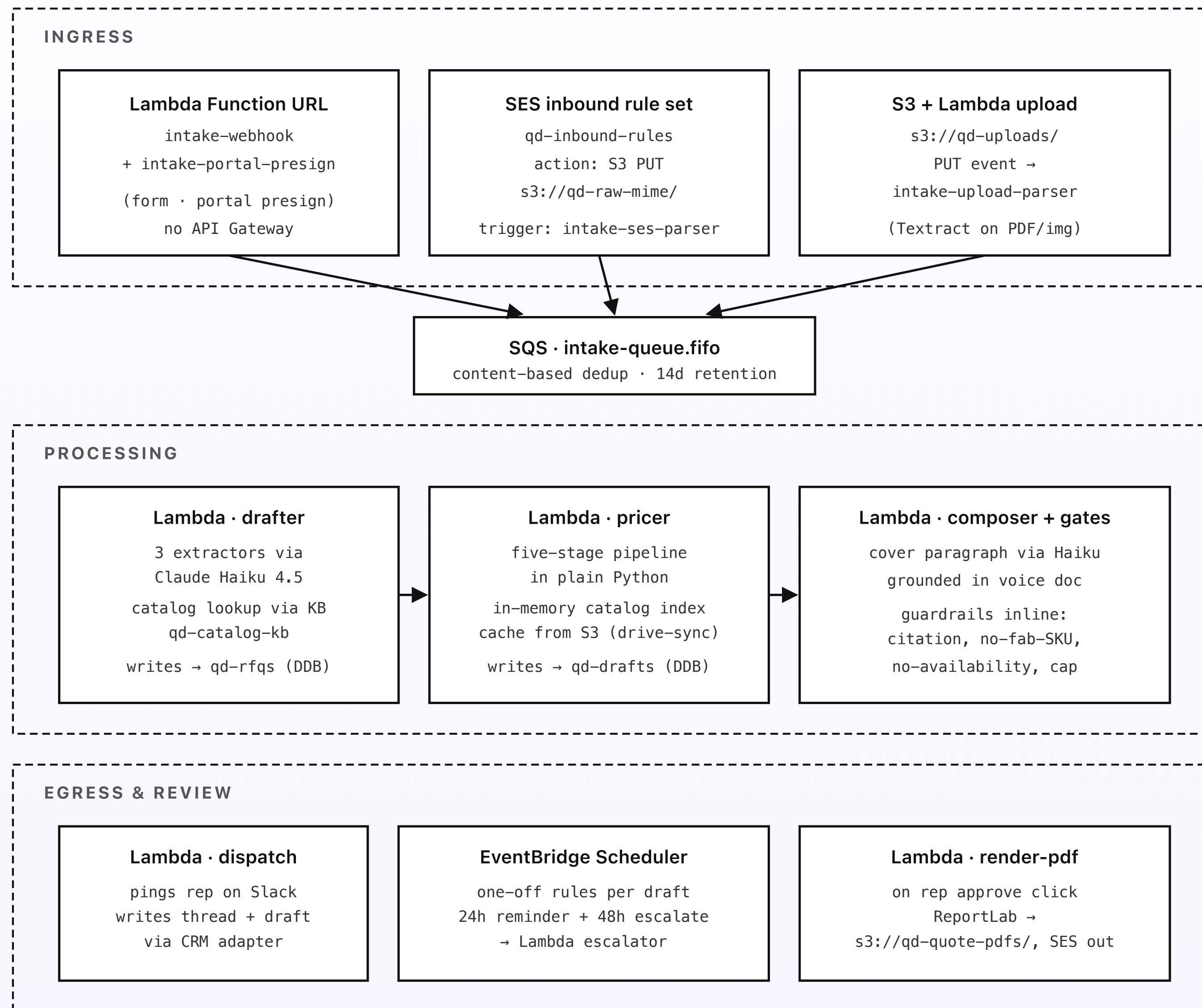
Engineering reference: the quote drafter architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Knowledge Base wiring, IAM scopes, the SES inbound rule set, the presigned-upload portal flow, the PDF rendering Lambda, and the CRM adapters. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **us-east-1**. SES inbound, Bedrock cross-Region inference, and S3 Vectors are all available there with current SLAs and full feature support. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the real failure mode for an SMB is the rep missing a draft, not a regional outage. One AWS account dedicated to the drafter (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every Lambda is invoked from a queue, an event, or a click — no synchronous chains.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into one queue), processing (drafter, pricer, composer), egress and review (dispatch, scheduler, PDF render). All Lambdas are queue- or event-driven; nothing is synchronous-chained.

Lambda functions

All Lambdas use the `arm64` architecture, the smallest memory size that meets latency targets (typically 256–512 MB), Python 3.14 runtime, and CloudWatch Logs at 7-day retention. Each function has its own least-privilege IAM role. None run inside a VPC, so there's no NAT Gateway and no cold-start ENI provisioning.

- **`intake-webhook`** — Lambda Function URL with `AuthType: NONE`. Verifies a per-form shared secret stored in Secrets Manager (`qd/forms/<form-id>/secret`) and a captcha token (hCaptcha `siteverify` or Cloudflare Turnstile `siteverify`). On success, writes a DynamoDB row to `qd-audit` and pushes a normalized message to `intake-queue.fifo`. Memory: 256 MB. Timeout: 5 s.
- **`intake-portal-presign`** — Lambda Function URL. Mints a presigned `s3:PutObject` URL into `s3://qd-uploads/<session>/<original-name>` with a 30-minute TTL and a 25 MB content-length range. Stores session metadata (buyer name, email, terms-accept timestamp) in `qd-sessions` (DDB, TTL = 1 hour).
- **`intake-ses-parser`** — S3 PUT trigger on `s3://qd-raw-mime/`. Parses MIME, walks the tree to the latest reply, strips signatures and quoted threads using `mail-parser-reply`. For attachments: Textract handles PDF, PNG, JPEG, and TIFF via `StartDocumentTextDetection` + `StartDocumentAnalysis`

(asynchronously to handle multi-page docs). Textract doesn't accept DOCX, so DOCX attachments are read with `python-docx` in the Lambda; XLSX attachments use `openpyxl`. Emits a normalized message to `intake-queue.fifo`. Memory: 512 MB. Timeout: 60 s (the wait for Textract is via SNS notification, not blocking).

- **`intake-upload-parser`** — S3 PUT trigger on `s3://qd-uploads/`. Same shape as the SES parser: Textract for non-text formats, normalized output to the queue.
- **`drafter`** — SQS event source on `intake-queue.fifo`, batch size 1. Calls Bedrock `InvokeModel` three times in parallel (`asyncio.gather`) for the line-items, constraints, and context extractors using `anthropic.claude-haiku-4-5-20251001-v1:0` via Global cross-Region inference. Calls `RetrieveAndGenerate` on Bedrock Knowledge Base `qd-catalog-kb` for the line-items resolution. Decides the move and writes to `qd-rfqs`. Pushes auto-draft moves to `draft-pricer-queue`; clarify, OOS, reject moves go straight to dispatch via the `qd-events` EventBridge bus. Memory: 1024 MB. Timeout: 90 s.
- **`pricer`** — SQS event source on `draft-pricer-queue`, batch size 1. Reads the in-memory catalog index (lazy-loaded from `s3://qd-catalog-source/catalog.txt` on cold start; cache invalidated on the next invocation when the S3 ETag changes, so a Drive edit propagates after the next 5-minute sync). Runs the five-stage pricing pipeline; writes to `qd-drafts`; pushes to `draft-composer-queue`. Memory: 512 MB. Timeout: 30 s. No model calls.
- **`composer`** — SQS event source on `draft-composer-queue`, batch size 1. One Bedrock `InvokeModel` call to Haiku 4.5 with the priced lines and the voice doc

passages as context. Runs the four guardrails inline (Gate 1: citation check; Gate 2: SKU regex + catalog lookup; Gate 3: block-list match; Gate 4: cap flag check). On any rejection, retries up to twice; on third failure, falls back to the templated cover paragraph from the voice doc. Updates `qd-drafts`; emits `draft.ready` on EventBridge. Memory: 1024 MB. Timeout: 60 s.

- **`dispatch`** — EventBridge rule on `draft.ready`, plus other move events (`rfq.clarify`, `rfq.oos`, `rfq.reject`). Pings the on-call rep in Slack via the Slack incoming webhook stored in Secrets Manager (`qd/slack/webhook`). Writes the conversation thread + draft to the configured CRM adapter. Memory: 256 MB. Timeout: 30 s.
- **`escalator`** — EventBridge Scheduler target. Runs at the 24-hour and 48-hour points after `draft.ready` if the draft hasn't been actioned. 24h: re-pings the same rep. 48h: pages the sales lead. Memory: 256 MB. Timeout: 15 s.
- **`render-pdf`** — Lambda Function URL invoked when the rep clicks *approve*. Reads the draft from `qd-drafts`, generates a PDF via `reportlab`, writes to `s3://qd-quote-pdfs/<rfq-id>.pdf`, sends to the customer via SES outbound, writes the final state to the CRM. Memory: 512 MB. Timeout: 30 s.

Storage

- **DynamoDB** · **`qd-audit`** — one row per intake event. PK `rfq_id` (UUIDv7); attributes: source lane, raw payload S3 key, dedupe hash, screen result. On-demand. TTL = 90 days.
- **DynamoDB** · **`qd-rfqs`** — one row per RFQ post-extraction. PK `rfq_id`; attributes: extracted line items (with confidence + KB match), constraints, context, chosen move, drafter version. On-demand. No TTL.

- **DynamoDB · `qd-drafts`** — one row per priced + composed draft. PK `rfq_id`; attributes: priced lines (with citations), cover paragraph, gate results, manager-approval flag, current state (`queued`, `approved`, `edited`, `rejected`, `expired`). On-demand. No TTL.
- **DynamoDB · `qd-sessions`** — presigned-upload sessions. PK `session_id`; TTL = 1 hour. On-demand.
- **S3 · `qd-raw-mime`** — raw inbound MIME. Lifecycle to Glacier at 30 days; expiry at 365 days.
- **S3 · `qd-uploads`** — buyer-uploaded spec docs. Same lifecycle.
- **S3 · `qd-quote-pdfs`** — rendered customer-facing PDFs. Lifecycle to Glacier at 30 days; expiry at 7 years (or your retention policy).
- **S3 Vectors · `qd-kb-vectors`** — the Bedrock Knowledge Base vector store backing `qd-catalog-kb`.
- **SQS · `intake-queue.fifo`** — FIFO with content-based deduplication (5-minute window). 14-day retention. DLQ `intake-queue-dlq.fifo` after 3 failures.
- **SQS · `draft-pricer-queue`, `draft-composer-queue`** — standard queues. 14-day retention. DLQs after 3 failures.

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`. The drafter and composer use the same model with different

system prompts; consolidating on one model keeps cost and quota management simple.

- **Embeddings.** `amazon.titan-embed-text-v2:0`, output dimension 1024, normalized. Used by the Knowledge Base for the catalog and rules docs.
- **Knowledge Base.** `qd-catalog-kb`, vector store on Amazon S3 Vectors at `s3://qd-kb-vectors/`, embedding model Titan v2. Data source: `s3://qd-catalog-source/`, populated by the `drive-sync` Lambda described below. Sync schedule: every 15 minutes via EventBridge Scheduler invoking `StartIngestionJob` on the data source. Bedrock KB doesn't ship a native Google Drive connector, so the Drive folder lives one hop away through the sync Lambda; this also means a versioned S3 bucket gives you point-in-time history of every catalog change for free.
- **Lambda ·** **`drive-sync`** — EventBridge Scheduler target, fires every 5 minutes. Uses the Google Drive API (service-account credentials in Secrets Manager under `qd/drive/sa`) to export `catalog.gdoc`, `rules.gdoc`, and `voice.gdoc` as plain text and write them to `s3://qd-catalog-source/<name>.txt` if the Drive `modifiedTime` is newer than the S3 `LastModified`. After each successful sync, calls `StartIngestionJob` on `qd-catalog-kb` only if any file actually changed. Memory: 256 MB. Timeout: 30 s.
- **Quotas.** Default account quotas are sufficient at SMB volume. Request a quota increase on Haiku TPS if you anticipate burst-mode RFQ volume above ~5/second.

IAM (least privilege per Lambda)

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

- **drafter role:** `bedrock:InvokeModel` on the Haiku ARN; `bedrock:Retrieve` + `bedrock:RetrieveAndGenerate` on `qd-catalog-kb`; `sqs:ReceiveMessage` + `DeleteMessage` on `intake-queue.fifo`; `dynamodb:PutItem` on `qd-rfqs`; `events:PutEvents` on the `qd-events` bus; `sqs:SendMessage` on `draft-pricer-queue`.
- **pricer role:** `sqs:ReceiveMessage` + `DeleteMessage` on `draft-pricer-queue`; `dynamodb:GetItem` on `qd-rfqs`; `dynamodb:PutItem` on `qd-drafts`; `s3:GetObject` on the catalog cache bucket; `sqs:SendMessage` on `draft-composer-queue`. *No* `bedrock:*`.
- **render-pdf role:** `dynamodb:GetItem` + `UpdateItem` on `qd-drafts`; `s3:PutObject` on `qd-quote-pdfs`; `ses:SendRawEmail` from the verified sender identity; CRM-adapter outbound network access via the Lambda's default outbound-internet (no VPC).
- **intake-portal-presign role:** `s3:PutObject` + `s3:GetObject` presign permission on `qd-uploads` only; `dynamodb:PutItem` on `qd-sessions`. Importantly, the role can *generate* presigned URLs that allow PUT, but the role itself never PUTs the content; the buyer's browser does, using the presigned URL.

SES inbound and domains

- Set the MX record on a dedicated subdomain (e.g. `quotes.your-company.com`) to `inbound-smtp.us-east-1.amazonaws.com`.
- Configure the SES inbound rule set `qd-inbound-rules` with one active rule. Conditions: recipient ends with `@quotes.your-company.com`. Actions, in order:

scan for spam (built-in), write to `s3://qd-raw-mime/<message-id>`, stop. The S3 PUT triggers `intake-ses-parser`.

- For SES outbound (sending the customer-facing quote PDFs), verify a separate sender identity at `quotes@your-company.com` and configure DKIM and SPF on the parent domain. SES is in production-mode (out of sandbox) by request.

Presigned-upload portal flow

1. Buyer opens the static portal at `https://upload.your-company.com/` (CloudFront in front of an S3 bucket; static HTML/JS only).
2. Buyer types name + email and accepts terms. Browser POSTs to the `intake-portal-presign` Function URL.
3. Lambda mints an `s3:PutObject` presigned URL into `s3://qd-uploads/<session-id>/<sanitized-filename>` with 30-minute TTL, 25 MB max content-length, and the `Content-Disposition` + `Content-Type` conditions baked into the signature. Lambda writes the session row to `qd-sessions` (TTL 1 hour) and returns the URL.
4. Browser does an `S3.PutObject` directly with the signed URL. No content passes through Lambda.
5. S3 PUT event triggers `intake-upload-parser`; from there it's the same path as the other lanes.

PDF rendering

The render-pdf Lambda uses `reportlab` packaged in a Lambda layer. The PDF template lives in the deployment artifact (not in S3) so render-time is deterministic; the layout includes the company logo, a fixed header, the priced lines as a table, the cover paragraph, and a footer with the quote validity and a unique reference number tied to `rfq_id`. Rendering is on-demand: the template is small enough to render in a few hundred milliseconds, and rendering only when the rep approves means drafts that get edited or rejected never burn the cycles.

CRM adapters

A single `crm-adapter` Lambda layer with one module per CRM, switched at runtime via an environment variable (`CRM_ADAPTER=hubspot|salesforce|pipedrive|drive-sheet`). Each adapter implements four operations: `upsert_contact(email, name, company, domain)`, `create_deal(rfq_id, contact_id, line_items, total)`, `attach_file(deal_id, s3_key)`, `add_note(deal_id, text)`. The Drive Sheet adapter is the fallback for the smallest setups; it appends to a Google Sheet via the Sheets API with the same schema as the other adapters' deal table.

OAuth credentials per CRM live in Secrets Manager under `qd/crm/<adapter>/oauth`. Refresh tokens are rotated by an EventBridge Scheduler rule firing a `refresh-crm-tokens` Lambda once a day.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. One log stream per Lambda invocation. Subscription filter on the keywords `"error"`,

`"throttle"`, `"timeout"` to a CloudWatch metric for alerting.

- **Alarms:** queue depth on `intake-queue.fifo` > 50 for 5 min (someone's posting fast and the drafter can't keep up); DLQ depth > 0 (something failed three times); Lambda error rate per function > 1% over 5 min.
- **X-Ray:** off by default. The pipeline is short and the queues handle correlation; X-Ray cost isn't worth it at SMB volume.
- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `qd-cost-alarm` which subscribes the on-call rep's email and Slack.

Config and secrets

Per-form shared secrets, the captcha key, the Slack webhook, the CRM OAuth credentials, and the SES sender identity all live in Secrets Manager under the prefix `qd/`. Application configuration (Bedrock model IDs, the discount cap, the block-list phrases for Gate 3, the Drive folder ID) lives in a single Parameter Store hierarchy `/qd/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment; Secrets Manager values are fetched per-invocation only when the secret is actually needed.

Deploy

Whichever IaC you prefer. The only opinionated bits are: deploy Function URLs separately from API Gateway (since there isn't one), configure the SES rule set as a separate stack since rule-set changes can affect mail flow, and turn on S3 versioning for `qd-catalog-source` so a bad Drive edit can be rolled back in one click. CDK with a Python stack file works well; SAM also fits. Total deployable

surface: around thirteen Lambdas, four DDB tables, five S3 buckets, three SQS queues, one EventBridge bus, one Knowledge Base, one SES rule set, and one Budgets alarm.

That's the full system. Six narrative posts and this engineering reference. The repo template (or a deployable starter) lives where the rest of my AWS scaffolding does — if you want to talk about adapting it for your business, see Work with me.