7-PART SERIES · FREE COMPANION

Inventory reorder bot

A serverless bot that watches the stock level of every item you sell or use, spots the ones dropping below their reorder point (allowing for how fast each one sells and how long the supplier takes to deliver), and drafts a ready-to-send purchase order to the right supplier for the owner to approve. It never orders on its own — the owner approves, edits, or skips right from the alert. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Inventory reorder bot

PART 1 OF 7

MAY 8, 2026 PART 1 OF 7 · INVENTORY REORDER BOT SERIES ~5 MIN READ

An inventory reorder bot on AWS for a few dollars a month

A small shop sells more things than anyone keeps in their head. The hero product that sells out the same week a big order lands and you have nothing to ship. The packaging boxes that take three weeks to arrive from the supplier. The cleaning supplies, the coffee beans, the filter cartridges, the screws in bin 14, the seasonal item that spikes every Friday. Run out of any of them and you either lose a sale or stop the line. This post walks through the design of a small bot that watches all of it, works out when each item is about to run low, and drafts a ready-to-send purchase order so the owner can approve it in one tap — it never orders on its own.

KEY TAKEAWAYS

- Three sources for stock levels: a Drive stock sheet, an inbox forwarding lane, and a live POS lane.
- Every item ends in one of four moves on each check: stocked, watch, reorder, or urgent reorder.
- Reorder point per item: daily sales rate × supplier lead-time days, plus a small safety buffer.
- Drafts a purchase order for the owner to approve. It never sends an order on its own.
- Designed on AWS for about $2 a month at typical small-business volume.

The whole system on one page

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

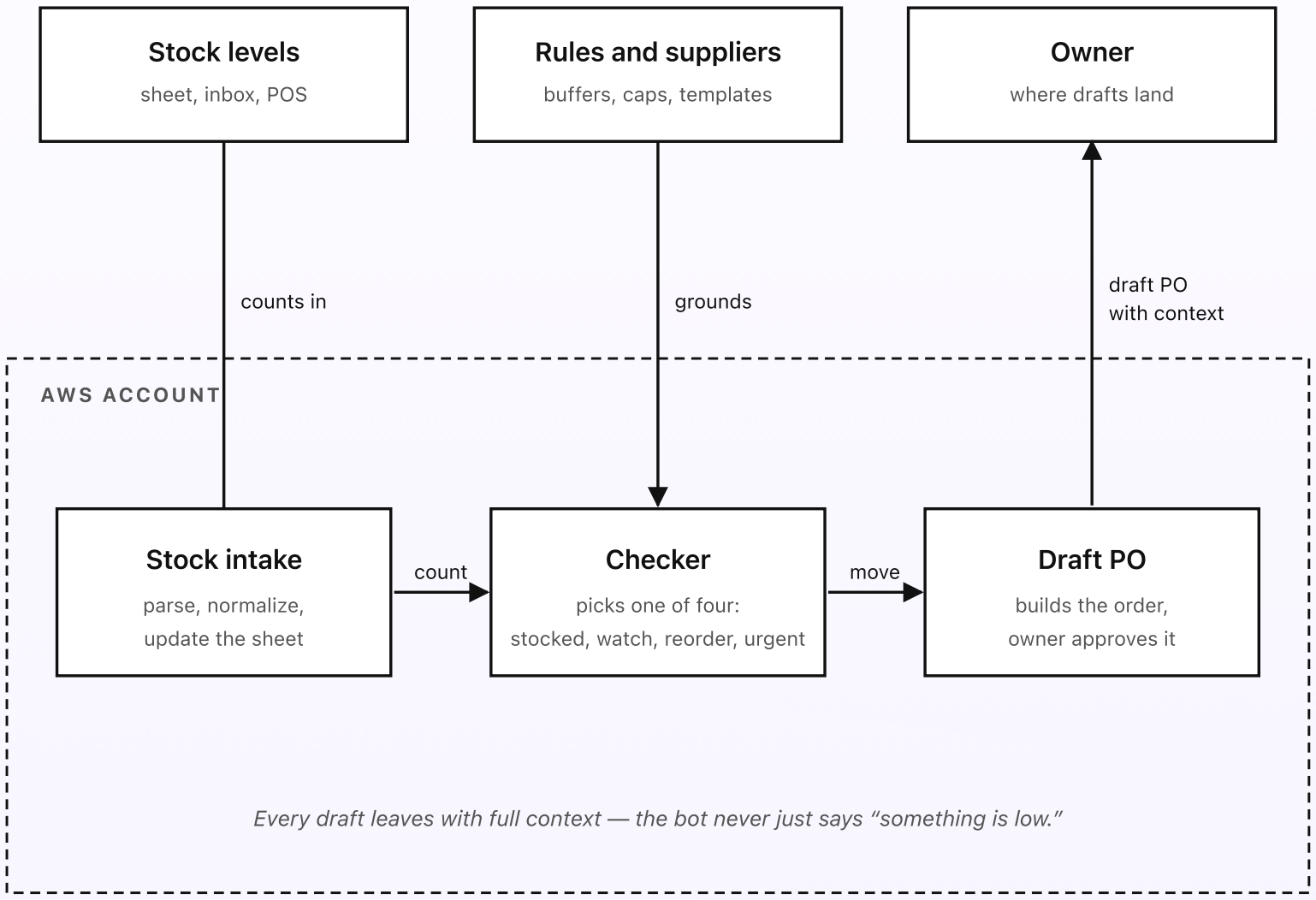

Stock levels
sheet, inbox, POS
Rules and suppliers
buffers, caps, templates
Owner
where drafts land
counts in
grounds
draft PO
with context
AWS ACCOUNT
Stock intake
parse, normalize,
update the sheet
count
Checker
picks one of four:
stocked, watch, reorder, urgent
move
Draft PO
builds the order,
owner approves it
Every draft leaves with full context — the bot never just says “something is low.”

Fig 1. Three sources outside, three pieces inside AWS. Counts flow in from a Drive stock sheet, an inbox forwarding lane, and a live POS lane. The Checker runs daily and picks one of four moves. Draft PO builds the right order and sends it to the owner to approve.

What you set up once (the outside)

- **Stock levels.** A Google Sheet in a Drive folder, one row per item: name, SKU (the short code that identifies the item), supplier, on-hand count, daily sales rate (how many you sell per day on average), supplier lead-time days (how long the supplier takes to deliver), safety buffer, pack size, and unit cost. You can fill it in once and let the other lanes keep it fresh; new counts can also enter via two more lanes covered in Part 2 — an inbox-forwarding lane (forward a stock-count sheet to a dedicated address and the bot proposes updated rows for one-tap approval) and a live point-of-sale lane (your till or online store tells the bot the moment something sells).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the safety buffer for each item or category — the extra cushion you want on hand in case sales spike or the supplier is late — plus an order cap (the most you ever want ordered in one go) and the quiet hours. The *supplier* doc holds each supplier's ordering email, pack size, minimum order, and the wording of the purchase order. Both live in Drive so a staffer can change a buffer or a supplier email without anyone redeploying code.
- **Owner.** The person who approves orders. They have a Slack member ID (so the draft is a private DM, not a public ping) or, if Slack isn't set up, an email address. Drafts land with the item name, the suggested order quantity, the unit cost, the supplier, and three buttons — *Approve*, *Edit*, *Skip*.

What runs on every check (the inside)

- **The stock intake.** Three sources feed the stock sheet. The Drive sheet itself is the canonical store. Counts can also be updated via the inbox forwarding lane (forward a stock-count sheet or a supplier price list to `stock@your-company.com`, the bot uses Textract to read it and Bedrock Haiku 4.5 to pull out item, count, and price, then drops a one-tap approval card in the owner's Slack before any row changes) and the live POS lane (your till or store posts each sale to the bot, which lowers the on-hand count in near real time).
- **The checker.** Runs once a day, early in the morning. Reads the stock list. For each item, works out the reorder point — how low the on-hand count can get before you have to order, allowing for how fast it sells and how long the supplier takes. Picks one of four moves. *Stocked*: plenty on hand — do nothing. *Watch*: getting close to the reorder point — note it for the digest, no order yet. *Reorder*: at or below the reorder point — draft a purchase order for the owner. *Urgent reorder*: already so low you may run out before the next delivery — draft the PO and flag it as urgent. The checker doesn't call a model on the daily check — the move logic is plain Python.
- **Draft PO.** Reads the supplier doc, builds the purchase order for the chosen move, rounds the order quantity up to the supplier's pack size, caps it at the rules-doc limit, and sends it to the owner. Slack DMs go through the Slack API. Email goes through SES outbound. Both honor quiet hours (no pings between 6pm and 8am local by default). Nothing is ever ordered until the owner taps Approve. Every draft writes a row in DynamoDB so the next day's check knows the item is already pending. A weekly digest summarizes what got ordered and what's on watch. A monthly summary writes a board-ready paragraph: spend by supplier, items that hit urgent, stockouts avoided.

In plain words

Your best-selling coffee beans sell about 8 bags a day. The supplier takes 5 days to deliver, and you keep a safety buffer of 20 bags. So the reorder point is 8 × 5 + 20 = 60 bags — once you're down to 60, it's time to order or you'll run dry while the new batch is in transit. The owner is you. Monday morning the bot sees the count has dropped to 58. It drafts a purchase order for 120 bags (a full case, rounded to pack size) at $4.20 a bag to your usual roaster, and DMs you in Slack: "Reorder — House Blend beans, 120 bags at $4.20 = $504, supplier Bayside Roasters — on hand 58, sells ~8/day." with Approve, Edit, Skip buttons. You tap Approve. The PO email goes to the roaster, the item is marked on-order, and the bot won't draft it again until the delivery lands. No stockout, no panic order at a premium, no Saturday spent counting shelves.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is the one weekend you sell out of your hero product, or the packaging that didn't arrive in time, or the rush order you paid double for because nobody noticed the bin was empty.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every draft ships with full context — item, suggested quantity, unit cost, supplier, on-hand count. The owner never has to dig.
- Four moves, always. Stocked, watch, reorder, urgent reorder. There is no fifth.
- Nothing is ordered without a human tap. The bot drafts; the owner approves, edits, or skips.
- Quiet hours are respected. A reorder draft at 11pm helps nobody; it waits for morning.
- The stock list lives in Drive. Adding an item, changing a supplier, or shifting a buffer doesn't need a deploy.
- Every action is logged. Audit an order next year and you can see exactly what was sent and by whom.

Why this shape

Most shops track stock in one of three places: a spreadsheet that nobody updates, a gut feeling about what's "getting low," or the moment a customer asks for something and the shelf is bare. The spreadsheet works until it doesn't — one busy week and the counts go stale. The gut feeling is fine for the three items you think about and useless for the other two hundred. And the empty shelf, of course, is the most expensive way to find out: by then you've already lost the sale.

The setup above moves the source of truth into a sheet the team already edits, but adds a small system that *looks at* that sheet every day and acts only when something needs ordering. Drafts come early enough to order at a normal price from your normal supplier. They include enough context that the owner doesn't have to go check the shelf. And nothing leaves without a human saying yes — because an order placed by mistake costs real money and ties up real cash. The bot is invisible most days; visible only on the days something actually needs reordering.

The next four posts walk through each piece in turn: how a stock level gets read, how the bot spots a low item, how a draft PO reaches the owner, and how a reorder gets approved and the cycle restarts. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 8, 2026 PART 2 OF 7 · INVENTORY REORDER BOT SERIES ~4 MIN READ

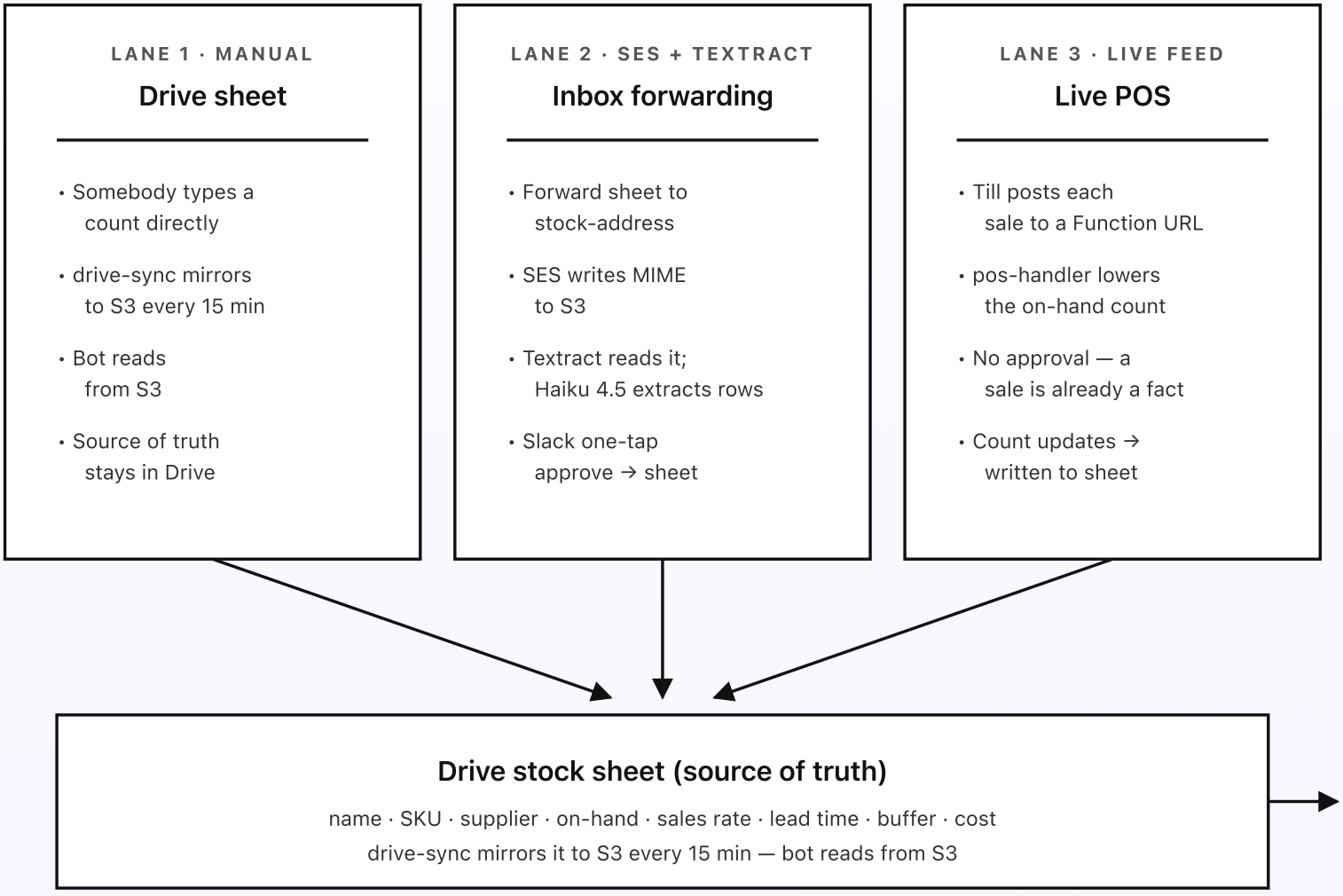
How a stock level gets read

The bot only watches the counts that are in the stock sheet. So the first job is making sure those counts actually reflect what's on the shelf. There are three ways a count gets updated: somebody types it in the Drive sheet, somebody forwards a stock-count sheet to a dedicated address, or your till tells the bot the moment something sells. The first one is obvious. The other two exist because in real life nobody types a new count into a sheet every time a customer buys something.

KEY TAKEAWAYS

- Three intake lanes feed one stock sheet: the Drive sheet, an inbox-forwarding lane, and a live POS lane.
- Inbound stock sheets are parsed by Textract; Bedrock Haiku 4.5 reads the text and proposes updated rows.
- Every parsed row goes to the owner's Slack for one-tap approval before it lands in the sheet.
- The POS lane posts each sale to the bot, which lowers the on-hand count in near real time.
- The stock sheet stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one stock sheet



The Drive sheet stays the source of truth — the other lanes are conveniences that keep its counts fresh.

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the inbox lane proposes rows for human approval and the POS lane keeps counts live. The drive-sync Lambda mirrors the sheet to S3 so the bot can read it without hitting Drive on every check.

Lane 1: the Drive sheet itself

The simplest lane. Open the stock sheet in Drive, edit a count, save. The columns are short: name, SKU, supplier, on-hand count, daily sales rate, supplier lead-time days, safety buffer, pack size, and unit cost. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://ir-stock-source/stock.csv` if the sheet has changed since the last sync. The bot reads from S3, not Drive directly. That keeps Drive API calls predictable and gives you S3 versioning for free, so a bad bulk-edit can be rolled back in one click.

This lane covers the cases where you do a physical count, you know the new number, and you can spend a few seconds typing it in. Most items get set up this way during the initial stocktake, and a monthly count is the natural time to correct any drift.

Lane 2: inbox forwarding (for count sheets and price lists)

Set up a dedicated inbound address — something like `stock@your-company.com` — via Amazon SES. Anyone on the team forwards a stock-count sheet (the printout from a shelf count, or a supplier's packing list) to that address and the bot takes it from there. SES writes the raw MIME to `s3://ir-raw-mime/`. The S3 PUT triggers a parser Lambda. The Lambda walks the MIME tree to the attachment, runs Amazon Textract on it (Textract reads PDF, PNG, JPEG, and TIFF

natively, and pulls tables cleanly; if somebody forwards a spreadsheet, the parser falls back to `openpyxl`), and gets back the extracted text plus any tables.

Then a Bedrock Haiku 4.5 call reads the text and emits structured rows: for each line it can match to a known SKU, the new on-hand count and, if the document is a price list, the new unit cost. The model prompt is short: "Match each line to a SKU in this list. Return JSON only. Mark each field with a confidence score. Do not invent a count that isn't in the text." The output goes to a Slack interactive message that pings the owner: the proposed changes, the confidence per field, and three buttons — *approve*, *edit*, *discard*. On *approve*, a Lambda writes the rows to the Drive sheet via the Sheets API. On *edit*, the owner gets a fillable modal pre-populated with the proposal. On *discard*, the message is logged and the file moved to a discarded prefix in S3 for audit.

The reason every parsed change goes to a human first is simple: a count the model misread is worse than a count that never made it into the sheet at all. The misread one will quietly tell you a bin is full when it's nearly empty, and the first you'll hear of it is the missed reorder.

Lane 3: live POS

Most shops already have a till or an online store that records every sale. Forcing the team to also re-type counts in a sheet is a fight you don't need to have on day one. Lane 3 lets the point-of-sale system keep the counts current on its own.

Your till or store is configured to post each sale to a Function URL — a small web address the bot exposes — the moment it happens. A small `pos-handler` Lambda receives the SKU and quantity sold, lowers the matching on-hand count in the sheet (batched every few minutes so a busy hour doesn't hammer the

Sheets API), and that's it. No approval step, because a recorded sale is already a fact — unlike a forwarded sheet, there's nothing to second-guess. If your POS supports it, the same lane can pick up stock received (a delivery arriving) and add to the count too.

The POS lane is the most hands-off of the three. A shop that can't wire its till to a webhook just leans on Lanes 1 and 2 and loses nothing but immediacy; a shop that can gets counts that are always close to reality.

Why the sheet stays the source of truth

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

Next post: how the bot actually reads the sheet, works out the reorder point per item, and picks one of four moves.

PART 3 OF 7

MAY 8, 2026 PART 3 OF 7 · INVENTORY REORDER BOT SERIES ~5 MIN READ

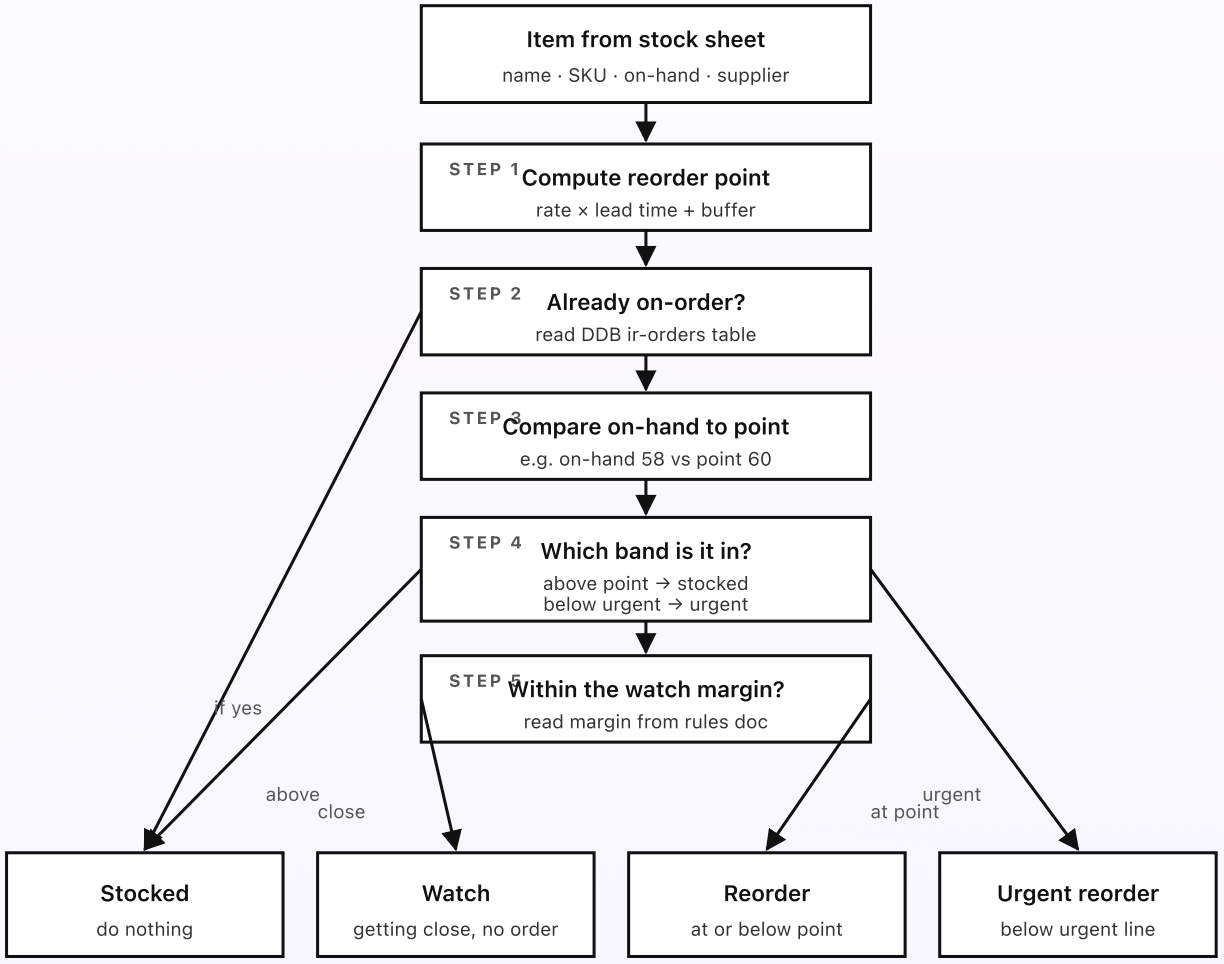
How the bot spots a low item

Once a day, early in the morning, an EventBridge Scheduler rule fires the checker Lambda. The Lambda reads the stock list, looks at one row at a time, works out how low the item is allowed to get before you have to order, and decides whether to do nothing or to draft a reorder — and if so, how urgent. The whole decision is plain Python. No model. No guessing. Every number lives in the stock sheet, where a staffer can edit it without a deploy.

KEY TAKEAWAYS

- The bot runs once a day via EventBridge Scheduler, early in the morning, local time.
- Reorder point per item = daily sales rate × supplier lead-time days + safety buffer. All from the sheet.
- Four moves per item, every check: stocked, watch, reorder, urgent reorder.
- DynamoDB tracks what's already on-order so the bot doesn't draft the same item twice.
- The bot itself never calls a model. The decision is entirely deterministic.

The decision flow, per item



The stock sheet holds every number — change a buffer and tomorrow's check uses the new value.

Fig 3. The bot's decision tree, per item, per daily check. Five steps decide which of four moves applies. The stock sheet holds every number; the bot only enforces them.

The reorder point: rate × lead time + buffer, all from the sheet

The reorder point is the answer to one plain question: how low can the on-hand count get before I have to order, or I'll run out while the new stock is on its way? Work it out from three numbers in the sheet. The *daily sales rate* — how many you sell per day on average. The *supplier lead-time days* — how long the supplier takes to deliver after you place an order. And the *safety buffer* — the cushion you keep in case sales spike or the supplier is late. The point is simply: rate × lead time + buffer.

An example. House Blend beans sell about 8 bags a day. The roaster takes 5 days to deliver. You keep a 20-bag buffer. So the reorder point is 8 × 5 + 20 = 60 bags. The moment the count drops to 60, ordering now means the new batch lands roughly as the buffer runs down — you never hit zero. Below 60 and you're eating into the cushion; the further below, the more urgent. The *urgent line* is the buffer on its own (here, 20 bags): at or below it, you're likely to sell out before any normal order can arrive, so the draft is flagged urgent so the owner sees it first.

Per-item overrides exist too. The sheet has an optional `point_override` column. Type a number there and the bot uses your reorder point for that one row instead of computing it. This is the right escape hatch for the item with seasonal demand

you understand better than any formula, or the one where you simply want a round number.

Four moves, always

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

- **Stocked.** The on-hand count is comfortably above the reorder point, or the item is already on-order. Do nothing. Most items, most days, are stocked.
- **Watch.** The count is getting close to the reorder point but hasn't reached it — within the watch margin set in the rules doc (say, 10% above the point). No order yet, but the item is noted in the weekly digest so the owner can see what's coming.
- **Reorder.** The count is at or below the reorder point but above the urgent line. Draft a purchase order with full context and a sensible order quantity, and send it to the owner to approve. Write a row to the `ir-orders` DynamoDB table marking the draft as pending.
- **Urgent reorder.** The count is at or below the urgent line — you may run out before the next delivery. Draft the same purchase order but flag it urgent so it sorts to the top of the owner's queue, and consider the rules-doc option to suggest the faster (often pricier) supplier if one is configured. Mark it urgent in DynamoDB. This is the one case where the bot is allowed to nag daily until the owner acts.

State that makes the decision deterministic

The checker reads one DynamoDB table every check. `ir-orders` records every draft and every approved order: `(item_id, status, draft_date, qty, supplier)` where status is one of `pending`, `on-order`, or `received`. With that table, the move-decision logic is a few dozen lines of Python and zero magic. A given item with a given on-hand count, a given reorder point, and a given order status always produces the same move. Re-running the check produces no duplicate drafts (because the state in DDB shows what's already pending or in flight).

When a delivery lands and the count goes back up (via Lane 1 or the POS lane), the item's status flips to `received` and it becomes eligible for a fresh draft next time it dips. Part 5 covers the approve-and-order flow in detail.

Why the daily check uses no model

The bot could call a model on the check to forecast demand or write a smarter draft. It doesn't. Two reasons. First, the daily check should be the one part of the system that is utterly predictable — if the sheet says rate 8, lead time 5, buffer 20, the reorder point is 60 and the draft fires at 60. A model in that loop introduces variance the owner can't reason about, and the cost of a wrong reorder is real cash. Second, model calls cost money, and most days most items are stocked, so the call would be wasted nine days out of ten.

Bedrock fires elsewhere — on the inbound parsing lane in Part 2, and on the monthly summary mentioned in Part 6. Not on the daily check. The checker itself is plain Python that reads a sheet and writes events.

Next post: how a draft PO finds the right supplier, how quiet hours are honored, and how the order quantity gets a sensible number before the draft lands in front of the owner.

PART 4 OF 7

MAY 8, 2026 PART 4 OF 7 · INVENTORY REORDER BOT SERIES ~5 MIN READ

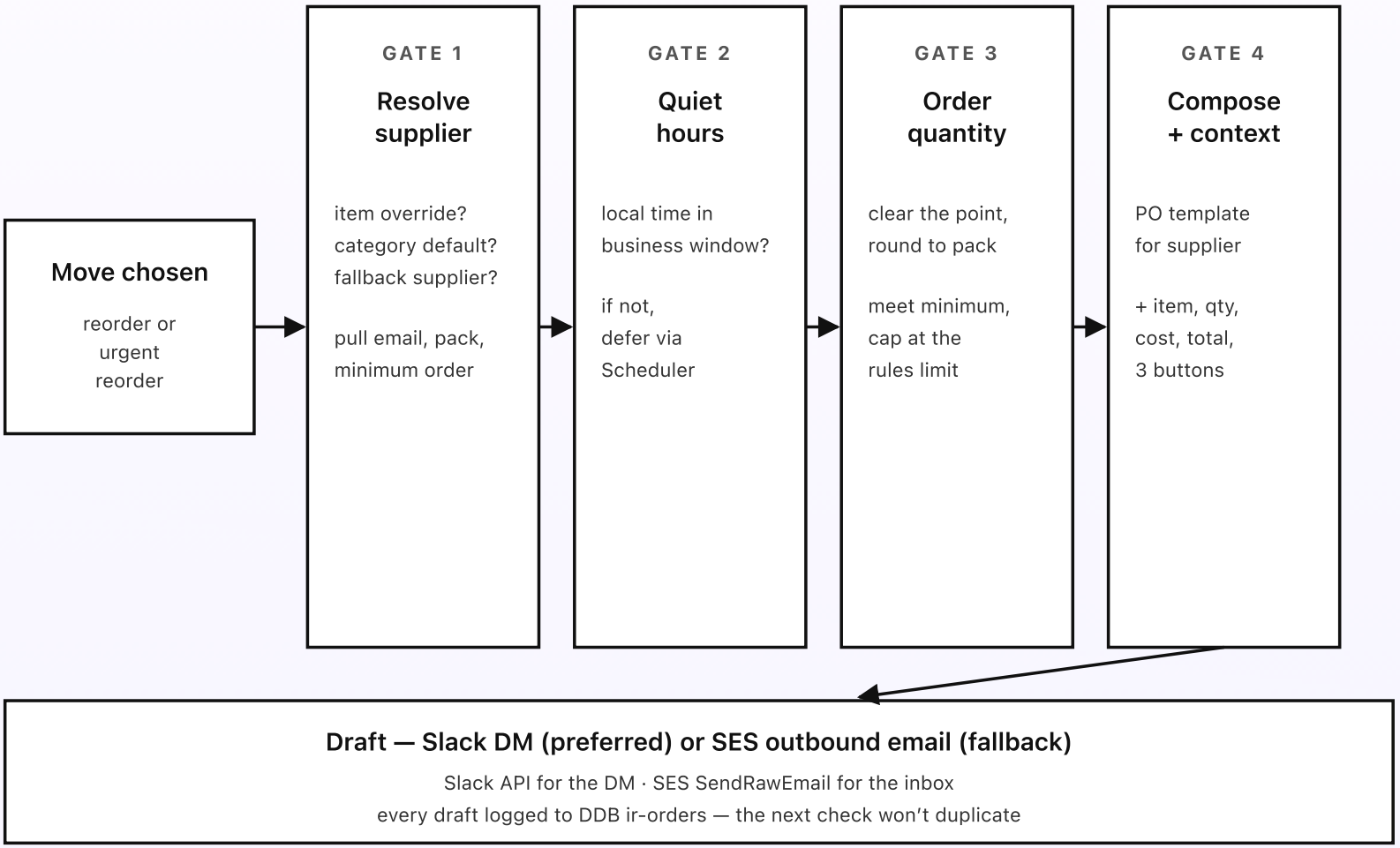
How a draft PO reaches the owner

The bot picked a move — reorder or urgent reorder. Now the draft-PO Lambda has to figure out which supplier to order from, how many to order, when to send the draft, and what context to attach. Get any of those wrong and the draft is worse than no draft: an order for the wrong amount, sent to the wrong supplier, landing at 2am, with no way to see why. Four small guardrails sit between the move and the draft that lands in front of the owner.

KEY TAKEAWAYS

- Supplier resolution: per-item override beats per-category default beats the configured fallback supplier.
- Slack DMs are the default; email is the fallback if no Slack ID is configured for the owner.
- Quiet hours defer a draft to the next business hour so nothing lands overnight.
- The order quantity is rounded up to pack size, meets the minimum, and is capped by the rules doc.
- Every draft ships with the item, suggested quantity, unit cost, supplier, and Approve/Edit/Skip.

Four guardrails on every draft



Every gate is a deterministic check — no model calls, no surprise order on a quiet Tuesday.

Fig 4. Four guardrails between the move and the drafted PO. Resolve the supplier. Honor quiet hours. Work out a sensible quantity. Compose with full context. Then send to the owner via Slack or email and log the draft so the next check doesn't duplicate.

Gate 1: resolve the supplier

Three places the draft-PO Lambda looks for the supplier of an item, in order. First, the stock sheet's per-item `supplier` column — if a row names a specific supplier, that's who the order goes to regardless of any default. Second, the per-category default in the rules doc ("all cleaning supplies default to Acme Wholesale"). Third, the configured fallback supplier — a catch-all so no item is ever stranded without somewhere to order from. The fallback should rarely fire; if it does, the weekly digest names every item that hit it so the supplier doc can be updated.

Once the draft knows the supplier, it pulls that supplier's ordering email, pack size, and minimum order from the supplier doc — the numbers Gate 3 needs to size the order, and the address Approve will use later. It also looks up where to send the draft to the owner: the rules doc maps the owner to a Slack member ID if one is set, otherwise to an email address. Slack is preferred because a DM with action buttons is faster to act on than an email; email is the fallback so nothing falls through the cracks.

Gate 2: quiet hours

The bot runs early in the morning, so the first time a move fires it's already near business hours. But an urgent draft that results from a sudden POS-driven dip can fire later in the day, and one-off deferred drafts can land outside the configured

window. A purchase-order draft at 11pm helps nobody — it just sits unread until morning and risks an approval tap from a half-asleep owner.

Gate 2 reads the rules doc's quiet-hours setting (default 6pm to 8am, configurable per business). If the current local time is in the quiet window, the draft creates a one-off EventBridge Scheduler rule that fires at the next business-hour minute and exits without sending. The Scheduler invokes the same draft-PO Lambda with the same payload at the deferred time, where Gate 2 will let it through. Urgent drafts can be configured to ignore quiet hours if a shop genuinely wants to be woken for a stockout risk — but the default is to wait.

Gate 3: a sensible order quantity

How many to order is its own small decision, and the bot makes it the same way every time. Start from how much you need to comfortably clear the reorder point and cover the next stretch of sales — a target stock level the rules doc can set as a number of days of cover (say, 30 days). Subtract what's on hand. That's the raw quantity. Then three adjustments, all from the supplier doc: round *up* to the supplier's pack size (you can't order half a case), raise to the supplier's minimum order if the raw number is below it, and finally cap the result at the rules-doc order cap so a typo in the sales rate can never produce a five-figure order.

The cap is the important guardrail. Every other gate makes the draft *more* right; the cap is the one that makes a wrong draft *small*. If a sales rate gets fat-fingered to 800 instead of 8, the cap means the worst the bot can propose is the capped quantity — a number the owner will immediately see is off, rather than a

catastrophic auto-order. And because nothing sends without the owner's tap anyway, even a capped-but-wrong draft is caught before any money moves.

Gate 4: compose with full context, then send

The supplier doc has one purchase-order template per supplier: a short order with placeholders for the item name, SKU, quantity, unit cost, line total, and any standing instructions (delivery address, account number, PO prefix). The draft-PO Lambda fills the placeholders, attaches the on-hand count and sales rate so the owner has the "why" at a glance, adds *Approve*, *Edit*, and *Skip* buttons, and sends the message to the owner via the Slack API. The supplier's ordering email isn't touched yet — that only happens on Approve, in Part 5.

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

An urgent draft is composed slightly differently: it carries an "Urgent" tag, sorts to the top of the owner's queue, and includes the days-of-cover-left figure ("~2 days at current sales") so the owner knows exactly how much runway is left. If the rules doc names a faster backup supplier for the item, the urgent draft offers it as a one-tap alternative in the Edit modal.

Every draft — Slack or email, reorder or urgent — writes a `pending` row to `ir-orders` in DynamoDB. The next day's check reads that row and knows not to draft the same item again while it's awaiting a decision.

Why the guardrails exist

None of these gates are exotic. They're the kind of small care a thoughtful buyer would take if they were placing the order themselves — check who we actually buy this from, don't draft at 11pm, order a round case rather than an odd number, and never let a typo turn into a giant order. Putting them in code as four small sequential gates makes them part of the design, not a feature you're trusting the person on shift to remember.

Next post: how a reorder gets approved once the owner has seen the draft — how Approve sends the PO and marks the item on-order, how Edit changes the order first, and how Skip steps back without losing the thread.

PART 5 OF 7

MAY 8, 2026 PART 5 OF 7 · INVENTORY REORDER BOT SERIES ~5 MIN READ

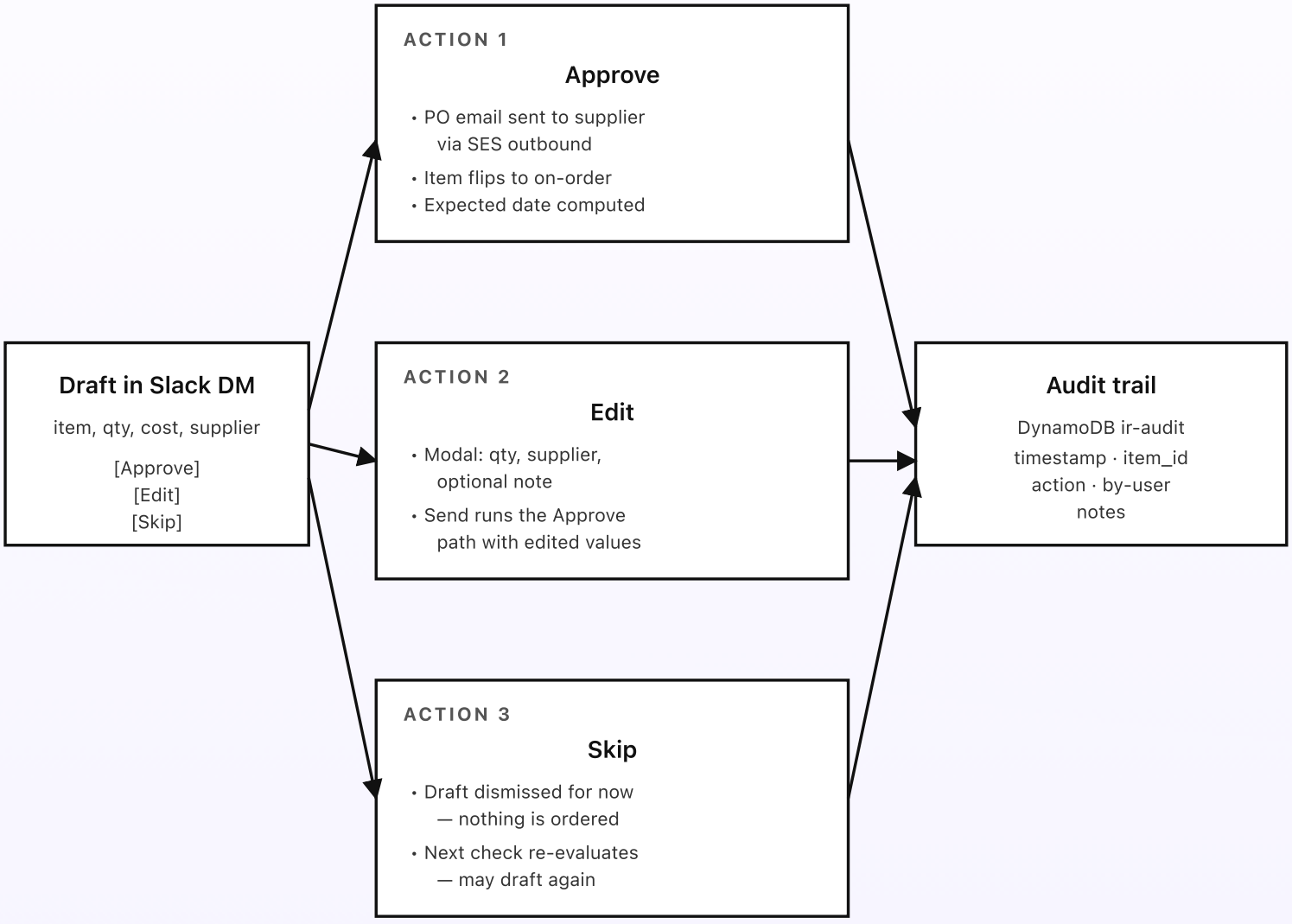
How a reorder gets approved

A draft lands in the owner's Slack DM at 8:03am. House Blend beans are down to 58, and the bot has drafted an order for 120 bags. There are three buttons: Approve, Edit, Skip. What happens when one gets tapped? The honest answer is "it depends which one." This post walks through the three things the owner can do with a draft — approve, edit, skip — and how the order state and the audit trail stay in sync. The one rule that never bends: nothing is ordered without a human tap.

KEY TAKEAWAYS

- Three actions per draft: *approve* (send to the supplier, mark on-order), *edit* (change first), *skip* (do nothing this round).
- Approve sends the PO email to the supplier via SES and flips the item to on-order.
- Edit opens a modal to change quantity, supplier, or note before anything is sent.
- Skip leaves the item alone — the bot re-checks tomorrow and may draft again.
- Nothing is ever ordered without a human tap. Every action writes an audit row.

Three actions on the draft



Nothing is ordered without a human tap — Approve and Edit both send; Skip never does.

Fig 5. Three actions per draft, three different effects. Approve sends the PO and marks the item on-order. Edit changes the order first, then sends. Skip orders nothing and lets the bot re-check tomorrow. Every action writes to the audit trail.

Action 1: approve (the most common)

The owner looks at the draft — 120 bags of House Blend at $4.20 to Bayside Roasters, on hand 58, sells about 8 a day — and it's exactly right. They tap *Approve*.

The button submits to a Function URL Lambda. Three things happen, in order. First, the purchase-order email is sent to the supplier's ordering address via SES outbound, formatted from that supplier's template with the item, quantity, unit cost, line total, and any standing instructions. Second, the item's row in `ir-orders` is flipped from `pending` to `on-order`, with the approved quantity and an expected-delivery date computed from the supplier's lead time (today + 5 days for Bayside). Third, an `action: approved` row is written to `ir-audit` with the user, timestamp, supplier, quantity, and total.

Tomorrow's check reads `ir-orders`, sees the item is on-order, and lands at *stocked* regardless of the on-hand count — no duplicate draft while the delivery is in transit. When the order arrives and the count goes back up (via a physical count or the POS lane), the status flips to `received` and the item rejoins the normal cycle, ready to be drafted again the next time it dips.

Action 2: edit (the small correction)

Sometimes the draft is nearly right but not quite. The owner knows a promotion is coming and wants 200 bags, not 120. Or they want to switch this one order to the backup roaster because Bayside is mid-holiday. Or they want to add a note to the supplier ("please split into two deliveries"). They tap *Edit*.

Edit opens a small Slack modal pre-filled with the quantity, the supplier, and an empty note field. The owner changes whatever they like — the modal still enforces the pack-size rounding and the order cap from Part 4, so an edit can't sneak past the guardrails — and hits Send. From there, the Function URL Lambda runs the exact same path as Approve, but with the edited values: the PO email goes to the (possibly changed) supplier with the (possibly changed) quantity and the note appended, the item flips to on-order, and the audit row records `action: edited` along with what changed from the original draft. Edit is just Approve with a detour through a form — the order that goes out is the one the owner actually wants.

Action 3: skip (the "not now")

Sometimes the right answer is to order nothing. Maybe a big delivery is already on a truck and the sheet hasn't caught up. Maybe the owner is discontinuing the item. Maybe they just counted the back room and there's plenty after all. They tap *Skip*.

Skip writes a row to `ir-orders` marking the draft dismissed and returns the item's status to normal. Nothing is ordered. The next daily check re-evaluates the item from scratch — so if it really is still low tomorrow, a fresh draft goes out, and the owner isn't permanently blinded to a real shortage just because they skipped once. To avoid nagging, the rules doc has a `skip_cooldown_days` setting (default

1): after a skip, the bot waits the cooldown before drafting that item again, so a deliberate skip isn't immediately undone. An urgent item ignores the cooldown — a genuine stockout risk is exactly when you want to be re-prompted.

If the owner skips an item repeatedly, the weekly digest flags it: "skipped 4 times in a row — is the reorder point wrong?" A pattern of skips usually means a number in the sheet (sales rate, buffer, or the point override) needs adjusting, and the digest is where that surfaces.

Every action is logged, every order is traceable

The `ir-audit` table records every approve, edit, and skip with the user who took the action, the timestamp, and a snapshot of the order before and after. If a wrong order goes out (wrong quantity, wrong supplier), the snapshot shows exactly what was sent and by whom, so a follow-up call to the supplier to cancel or amend is backed by a clear record rather than a guess. The audit row is also what the monthly summary reads to report spend by supplier and orders by approver.

This kind of traceability matters because an order is money leaving the business. The next time anyone asks "why did we order 200 bags of beans in May," the answer is one row away: who approved it, when, off which draft, with what numbers in front of them. That's the difference between a system you trust with real purchasing and one you don't.

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

PART 6 OF 7

MAY 8, 2026 PART 6 OF 7 · INVENTORY REORDER BOT SERIES ~3 MIN READ

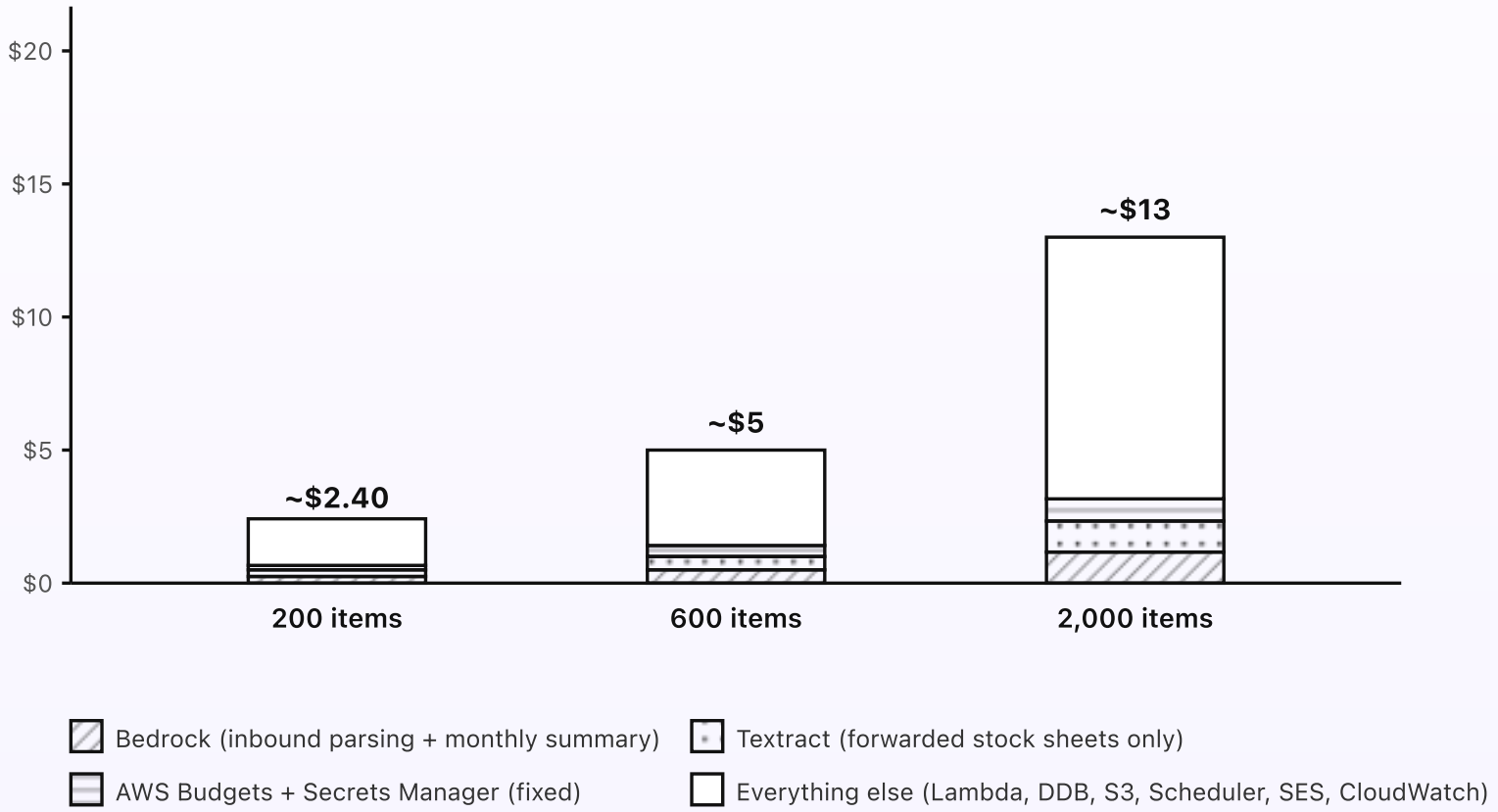
What the inventory reorder bot costs

The bot is one of the cheapest systems in this whole series. The daily check reads a CSV from S3, does some arithmetic on each item, writes a few rows to DynamoDB, and posts a handful of drafts to Slack. It calls no models on the check. Bedrock fires only when somebody forwards a stock-count sheet and once a month for the board summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.40/month at typical SMB volume (around 200 tracked items).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The daily check costs pennies — no model calls.
- Bedrock fires only on inbound stock-sheet parsing (a few times a month) and the monthly summary.
- At 600 tracked items the bill is around $5. At 2,000 items it's around $13.

Cost at three volumes



The daily check is the dominant cost — and even that is fractions of a cent per item per day.

Fig 6. Monthly cost at three tracked-item volumes. Bedrock and Textract are small slivers because they only fire on the inbound parsing lane and the monthly summary. The dominant cost is the everything-else bucket: the daily check reading every item and the POS lane handling sale events.

Where the dollars actually go

Lambda runtime (the bulk). The checker runs once a day. Each check reads the stock CSV from S3, iterates the rows, computes the reorder point for each, and decides on a move. At 200 items, that's a few hundred milliseconds. At 2,000 items it's a couple of seconds. Either way it's pennies a month. Add the draft-PO Lambda firing for each reorder (a handful to a few dozen a month), the Function URL Lambda for approvals, the pos-handler firing on each sale, the calendar-free drive-sync Lambda every fifteen minutes — the Lambda total still lands under a couple of dollars at all three volumes. The POS lane is the one piece that scales with sales rather than item count, so a busy shop sees a touch more here.

DynamoDB on-demand. Three small tables: `ir-orders`, `ir-state`, `ir-audit`. Reads are dominant during the daily check (one read per item per check). Writes are draft events, approvals, and audit rows. Pennies a month at any of these volumes.

S3 + Storage. The mirrored stock CSV plus the archived MIME from any forwarded sheets. A few hundred KB total at SMB volume. Effectively free.

EventBridge Scheduler. The daily check rule plus deferred draft rules from the quiet-hours gate. A few invocations a day. Pennies.

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

Bedrock (only when something fires it). The daily check uses no Bedrock. The inbound parsing lane fires Haiku 4.5 once per forwarded sheet: a few thousand

input tokens (the Textract output) and a few hundred output tokens (the proposed rows), so a fraction of a cent per parse. At a few forwarded sheets a month, Bedrock costs cents. The monthly summary is one larger call: write a paragraph that summarizes the month's orders, spend, and stockouts avoided; a couple of cents.

Textract (only on forwarded sheets). Per-page pricing; a typical count sheet or price list is one to a few pages. A few cents per parse. At a few sheets a month, Textract is a few cents to a dollar. At 2,000 tracked items with twenty sheets forwarded per month, it lands around a couple of dollars.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the approve endpoint and the POS webhook.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The checker sleeps 23.99 hours a day.
- **A Knowledge Base.** The stock list is structured rows, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the check.** The daily decision is plain Python. Bedrock fires only on the inbound parsing lane and the monthly summary.

How the cost scales

Lambda runtime grows roughly linearly with item count, because every item is evaluated on every check. DynamoDB grows linearly too. The POS lane grows with sales volume rather than item count. Bedrock and Textract are uncorrelated with item count — they only fire when somebody forwards a sheet or it's the first of the month. So the bill at 5,000 tracked items is around $32; at 10,000 it's around $62. Past those volumes the daily-check model probably stops being right (you'd switch to a partial-check that only evaluates items near their reorder point, or react to POS dips instead of polling), but those are optimizations for very large catalogs — not redesigns.

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

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

PART 7 OF 7

MAY 8, 2026 PART 7 OF 7 · INVENTORY REORDER BOT SERIES ~8 MIN READ

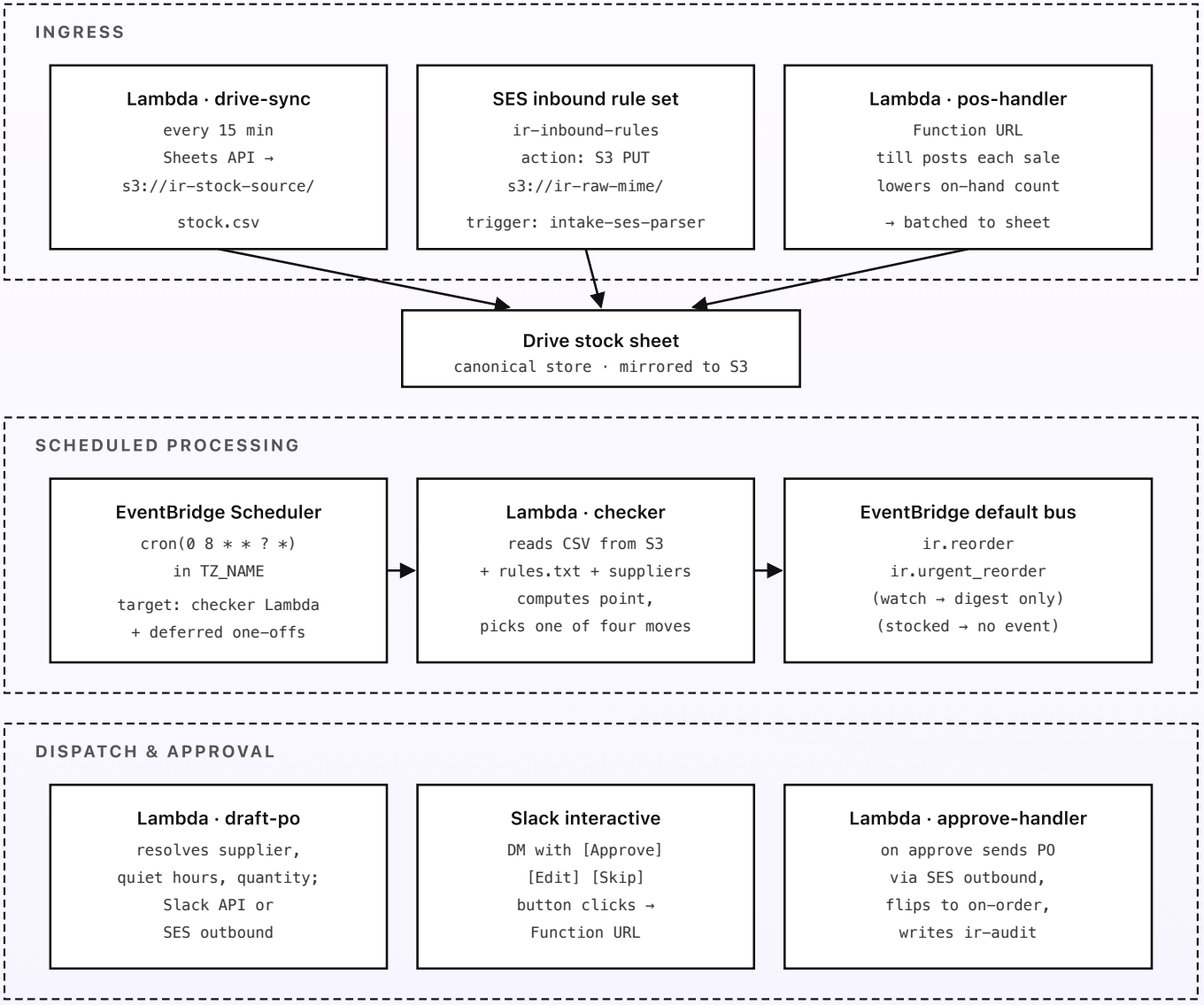
Engineering reference: the inventory reorder bot architecture

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

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound, Bedrock Global cross-Region inference, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is a missed reorder, not a regional outage. One AWS account dedicated to the bot (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Nothing is ordered without a human tap — and every interaction is logged to ir-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the stock sheet), scheduled processing (the daily check emitting events), dispatch and approval (the draft ships and the owner's decision is recorded). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

All Lambdas use the `arm64` architecture, the smallest memory size that meets latency targets (typically 256 MB), Python 3.14 runtime, and CloudWatch Logs at 7-day retention. Each function has its own least-privilege IAM role. None run inside a VPC.

- **`drive-sync`** — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `ir/drive/sa`) to export the stock sheet as CSV and write to `s3://ir-stock-source/stock.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and supplier docs to `s3://ir-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`pos-handler`** — Lambda Function URL, `AuthType: NONE`, verifies an HMAC signature on the request body using a shared secret with the POS. Receives sale (and optionally received-stock) events, validates the SKU against the current stock CSV, and applies the count delta. To avoid hammering the Sheets API during a busy hour, deltas are accumulated in `ir-state` and flushed to the Drive sheet on a short timer (a `rate(5 minutes)` Scheduler target that batches pending deltas). Memory: 256 MB. Timeout: 15 s.

- **`intake-ses-parser`** — S3 PUT trigger on `s3://ir-raw-mime/`. Parses MIME, extracts the attachment, runs Textract via `StartDocumentTextDetection` + `StartDocumentAnalysis` (asynchronously to handle multi-page sheets, with table extraction on). On Textract completion (via SNS notification), reads the structured text and calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to match lines to known SKUs and propose updated rows. Posts the proposal to Slack with Approve/Edit/Discard buttons. For XLSX/CSV attachments (Textract isn't needed), parses directly with `openpyxl` or the stdlib `csv` module. Both are stable and widely used in 2026; `openpyxl`'s maintenance velocity is light, which is acceptable for a parsing path that runs a few times a month. Memory: 512 MB. Timeout: 60 s.
- **`checker`** — EventBridge Scheduler target, daily at 8am local time (the schedule expression runs in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). Reads `s3://ir-stock-source/stock.csv` and the rules and supplier docs. For each row, computes the reorder point (`rate × lead_time + buffer`, honoring any `point_override`), reads order state from `ir-orders`, and decides on a move. Emits one event per row that needs an order: `ir.reorder` or `ir.urgent_reorder`, with the item context as the event payload. *Watch* items are accumulated for the digest; *stocked* items emit nothing. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`draft-po`** — EventBridge rule on the two move events. Resolves supplier, checks quiet hours, computes the order quantity (clear-the-point target minus on-hand, rounded up to pack size, raised to the supplier minimum, capped at the rules-doc cap), formats the draft from the supplier template, and sends via the Slack `chat.postMessage` Web API (`ir/slack/bot-token` in Secrets

Manager) or SES `SendRawEmail`. On a quiet-hours defer, creates a one-off EventBridge Scheduler rule that re-invokes `draft-po` at the next available business minute. Writes a `pending` row to `ir-orders` after a successful send. Memory: 256 MB. Timeout: 30 s. *Never contacts the supplier — only the owner.*

- **`approve-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive button clicks (Approve/Edit/Skip) and by email-link clicks. On *approve* or *edit*, sends the PO email to the resolved supplier via SES `SendRawEmail`, flips the item in `ir-orders` to `on-order` with the approved quantity and an expected-delivery date, and writes to `ir-audit`. On *skip*, marks the draft dismissed and applies the `skip_cooldown_days`. Memory: 256 MB. Timeout: 15 s.
- **`digest`** — EventBridge Scheduler target, weekly Sunday 6pm. Reads `ir-orders` for the past week and the stock sheet; sends a digest message to a configured Slack channel summarizing orders placed, items on watch, and any item skipped repeatedly. No Bedrock; the message is a plain summary table. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `ir-orders` and `ir-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph board narrative (spend by supplier, items that hit urgent, stockouts avoided); emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB · `ir-orders`** — one row per draft/order. PK `item_id`; sort key `order_id`; attributes: `status` (pending/on-order/received/skipped), `qty`,

`supplier`, `unit_cost`, `draft_date`, `expected_date`. On-demand. No TTL.

- **DynamoDB · `ir-state`** — per-item scratch state. PK `item_id`; attributes: `pending_delta` (un-flushed POS deltas), `skip_until`, `last_move`, `last_checked`. On-demand.
- **DynamoDB · `ir-audit`** — one row per write action of any kind. PK `(item_id, ts)`; attributes: `action` (approved/edited/skipped/parsed), `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **S3 · `ir-stock-source`** — mirrored CSV from the Drive stock sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3 · `ir-rules-source`** — mirrored rules and supplier docs as plain text. Versioning enabled.
- **S3 · `ir-raw-mime`** — raw inbound MIME from forwarded sheets. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3 · `ir-source-docs`** — the parsed source sheets and price lists after the inbound parser handles them, kept for reference and audit.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `intake-ses-parser` for the inbound stock-sheet parsing, and `summary` for the monthly board narrative. Claude Sonnet 4.6 (`anthropic.claude-sonnet-4-6-20250930-v1:0`) is available as a config-flag fallback if a particularly messy supplier price list needs heavier reasoning, but Haiku 4.5 handles the normal case.

- **Embeddings.** Not used. The stock list is structured rows; deterministic lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The checker itself doesn't call Bedrock; the parsing lane fires a few times a month at most.

EventBridge Scheduler config

- `ir-daily-check` — `cron(0 8 * * ? *)` in the SMB's timezone. Target: `checker` Lambda.
- `ir-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `ir-pos-flush` — `rate(5 minutes)`. Target: a flush handler that writes accumulated POS deltas from `ir-state` to the Drive sheet.
- `ir-weekly-digest` — `cron(0 18 ? * SUN *)` in TZ. Target: `digest` Lambda.
- `ir-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `draft-po` when a quiet-hours defer is needed. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `stock.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.

- SES inbound rule set `ir-inbound-rules`: one rule with recipient `stock@your-company.com` → spam scan → S3 PUT to `s3://ir-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the PO emails and email-fallback drafts: verify a sender identity at `orders@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request. Supplier PO emails are sent only from `approve-handler`, never from `draft-po`.

IAM (least privilege per Lambda)

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

- **checker role:** `s3:GetObject` on the stock, rules, and supplier keys; `dynamodb:Query` + `GetItem` on `ir-orders`, `ir-state`; `events:PutEvents` on the default bus. *No* `bedrock:*` and *no* `ses:*`.
- **draft-po role:** `scheduler:CreateSchedule` for the deferred one-offs; `secretsmanager:GetSecretValue` on the Slack bot token; `dynamodb:PutItem` on `ir-orders`; outbound network access to `slack.com`. *No* `ses:SendRawEmail` to a supplier — only the owner-facing email path.
- **approve-handler role:** `ses:SendRawEmail` from the verified sender identity (the only role that can email a supplier); `dynamodb:PutItem` + `UpdateItem` on `ir-orders` and `ir-audit`; `secretsmanager:GetSecretValue` on the Slack signing secret and the Sheets-API secret; `dynamodb:Query` for order-state lookup.
- **intake-ses-parser role:** `s3:GetObject` on `ir-raw-mime`; `textract:StartDocumentTextDetection` + `StartDocumentAnalysis`;

`bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token.

- **pos-handler and drive-sync roles:** `secretsmanager:GetSecretValue` on the relevant secret (POS HMAC key; Google service-account); `s3:GetObject` / `PutObject` on the stock and rules buckets; `dynamodb:UpdateItem` on `ir-state` (pos-handler); outbound network to `www.googleapis.com` (drive-sync).

Slack interactive flow

The Slack incoming webhook is the simplest delivery surface but doesn't support interactive button responses. So the draft messages are posted via the `chat.postMessage` Web API instead, with Block Kit blocks containing the action buttons. Button clicks are sent by Slack to the configured Interactivity request URL, which is the `approve-handler` Function URL. `approve-handler` verifies the Slack signing secret on the inbound request, parses the action_id (`approve`, `edit`, `skip`), opens a modal if needed (Edit opens a modal; Approve and Skip are one-tap), and processes the response when the modal is submitted.

The Slack app needs `chat:write`, `im:write`, and the Interactivity URL configured. The bot token lives in Secrets Manager under `ir/slack/bot-token`. The signing secret is `ir/slack/signing-secret`.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch

metric for alerting.

- **Alarms:** checker Lambda failures > 0 in a day (the daily check is the one piece that has to run); draft-po failure rate > 1% in 24h; approve-handler signature-verification failures > 5/hour (might mean the Slack secret rotated); any supplier-email send failure (an order that didn't actually reach the supplier).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `ir-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for the Drive and Sheets APIs live in Secrets Manager under `ir/drive/sa`. The Slack bot token and signing secret are under `ir/slack/*`. The POS webhook HMAC key is under `ir/pos/hmac`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, quiet-hours window, watch margin, days-of-cover target, order cap, skip cooldown, and fallback supplier all live in Parameter Store under `/ir/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) and AWS SAM for the stack. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `ir-stock-source` and `ir-rules-source` so a bad Drive edit can be rolled back in one click, and

version the EventBridge Scheduler timezone setting so you don't accidentally start running the daily check in UTC after a CI rotation. Keep the supplier-email permission isolated to `approve-handler` so no scheduled path can ever place an order. Total deployable surface: around eight Lambdas, three DDB tables, four S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SES rule set, and one Budgets alarm.

That's the full system. Six narrative posts and this engineering reference. If you want to talk about adapting it for your business, see Work with me.