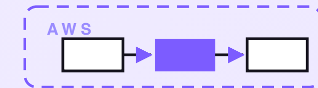
7-PART SERIES · FREE COMPANION



Expense approver

A serverless system that takes each staff expense claim and its receipt, reads the receipt, checks the claim against your written expense policy, auto-clears the small in-policy ones for a manager's one-tap confirm, and sends anything over a limit or out of policy to the right person with the reason. A human approves every payment — nothing is reimbursed automatically. 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/expense-approver

CONTENTS

Expense approver

01 An expense approver on AWS for a few dollars a month

02 How an expense claim gets submitted

03 How an expense claim gets checked against policy

04 How an expense claim finds its approver

05 How an expense claim gets paid

06 What the expense approver costs

07 Engineering reference: the expense approver architecture

PART 1 OF 7

MAY 28, 2026 PART 1 OF 7 · EXPENSE APPROVER SERIES ~5 MIN READ

An expense approver on AWS for a few dollars a month

A small business runs on a steady drip of small spends. The $12 client coffee. The $30 airport taxi. The $400 software seat somebody bought because the trial ran out on a Friday. Each one comes back as a claim with a blurry receipt, and somewhere a manager is supposed to look at it, decide if it's fair, and approve it for payment. In practice the manager rubber-stamps the pile once a week, or the pile sits for a month and the team grumbles. This post walks through the design of a small system that reads each claim, checks it against your policy, clears the easy ones for a one-tap confirm, and sends the rest to the right person with the reason — while a human still approves every single payment.

KEY TAKEAWAYS

- Three ways a claim gets in: a web form, a forwarded receipt email, and a chat upload.
- Every claim ends in one of four outcomes: clear, confirm, review, or reject.
- Per-category limits live in a policy doc: meals $40/day, taxis $30/trip, software needs sign-off.
- A human approves every payment. Nothing is reimbursed automatically — ever.
- Designed on AWS for about $2.40/month at typical small-business volume.

The whole system on one page

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

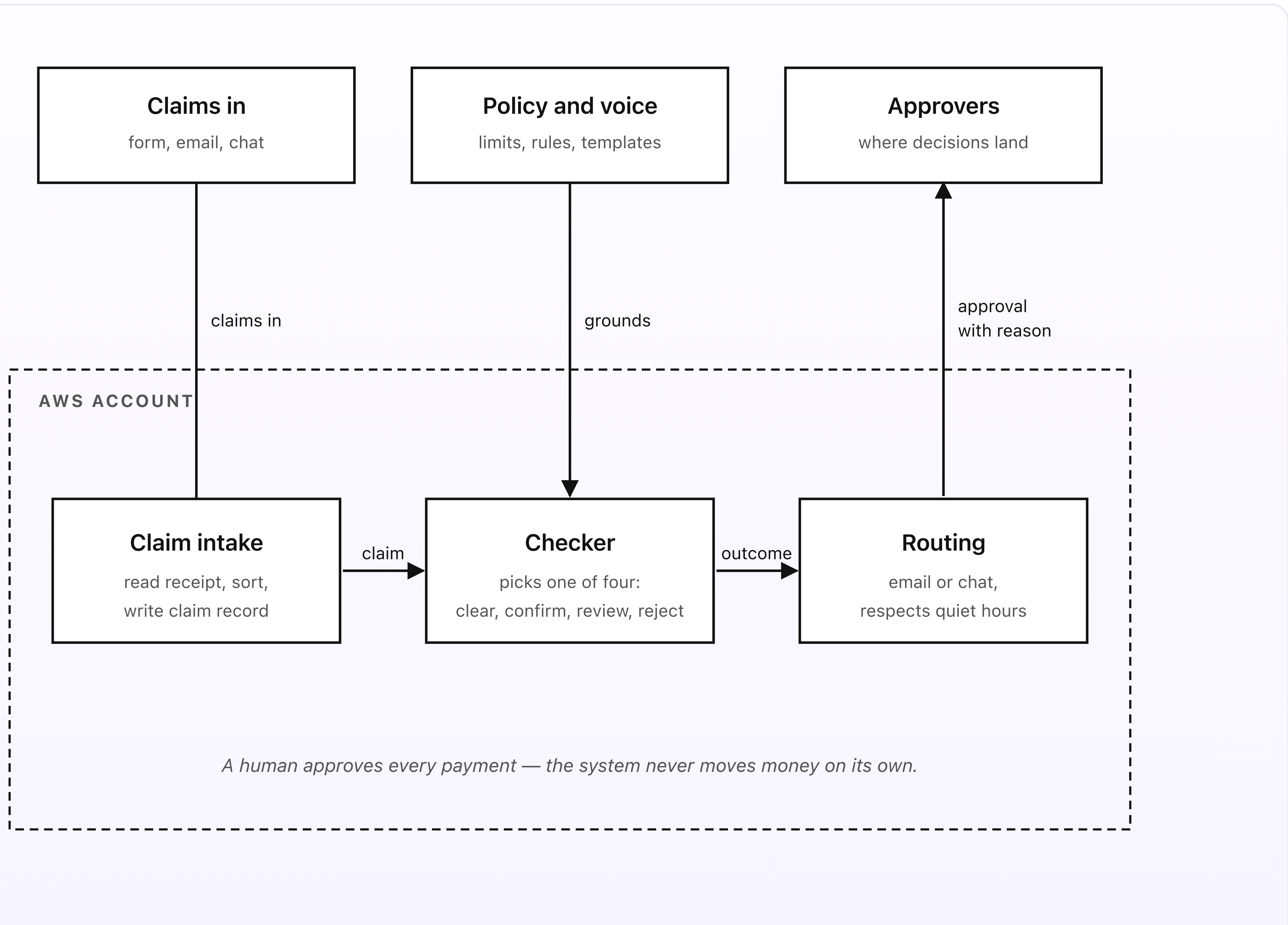

Claims in
form, email, chat
Policy and voice
limits, rules, templates
Approvers
where decisions land
claims in
grounds
approval
with reason
AWS ACCOUNT
Claim intake
read receipt, sort,
write claim record
claim
Checker
picks one of four:
clear, confirm, review, reject
outcome
Routing
email or chat,
respects quiet hours
A human approves every payment — the system never moves money on its own.

Fig 1. Three sources outside, three pieces inside AWS. Claims flow in from a web form, a forwarded email, and a chat upload. The Checker compares each against policy and picks one of four outcomes. Routing sends the right request to the right person, who makes the call.

What you set up once (the outside)

- **Claims in.** Three ways a staff member submits a claim. A short web form (pick a category, type an amount, snap a photo of the receipt). A forwarded receipt email (forward the vendor's emailed receipt to a dedicated address and the system takes it from there). And a chat upload (drop a receipt photo into a team chat channel). All three land as one claim record, covered in Part 2. Every claim carries an amount, a date, a category, the claimant, and a receipt image.
- **A policy folder.** Two short Google Docs in a Drive folder. The *policy* doc covers the per-category limits in plain prose — "meals up to $40 a day, taxis up to $30 a trip, software of any amount needs a manager's sign-off, anything over $250 goes to finance." It also lists which categories are allowed, when a receipt is required, and who approves what. The *voice* doc holds the message templates — what the approval card and the reject note actually say.
- **Approvers.** The people who decide. A manager for the everyday in-policy claims, a finance lead for the larger ones. Each approver has a chat ID (so the request is a private message) or, if chat isn't set up, an email address. The approval card lands with the claim amount, the category, whether it's in or out of policy and why, a link to the receipt, and Approve, Reject, and Ask buttons.

What runs on every claim (the inside)

- **The claim intake.** Each new claim, however it arrived, is normalized into one record. Textract reads the receipt image and pulls out the amount, the date, and the vendor. A small Bedrock Haiku 4.5 call sorts the receipt into a category ("this looks like meals," "this looks like a taxi") so the claimant doesn't have to get the category perfectly right. The cleaned claim is written to DynamoDB and the receipt is stored in S3.
- **The checker.** Reads the claim. Compares the amount against the per-category limit in the policy doc. Picks one of four outcomes. *Clear*: in policy and small — the manager gets a one-tap confirm card. *Confirm*: in policy but worth a quick look (close to the limit, or the category that always wants eyes) — same one-tap confirm, with the detail shown. *Review*: over a limit or out of policy — full approval card to the right person, with the reason spelled out. *Reject candidate*: clearly outside policy (a banned category, no receipt where one is required) — the system proposes a reject for a human to confirm. The checker doesn't call a model on the limit comparison; the math is plain Python.
- **Routing.** Reads the voice doc, formats the approval card for the outcome and category, and sends it to the resolved approver. Chat messages go through a chat webhook; email goes through SES outbound. Both honor quiet hours so a request doesn't land at 2am. Every send and every decision writes a row to DynamoDB so the trail is complete. A weekly digest summarizes what was approved and what's still waiting. A monthly summary writes a board-ready paragraph: spend by category, top claimants, anything that needed a second look.

In plain words

Sam took a client to lunch and spent $36. He snaps the receipt into the web form and picks “meals.” The intake reads the receipt: $36.00, today, “Olive & Vine.” The category sorts to meals. The checker sees meals has a $40/day limit and $36 is under it, with a receipt attached — that’s a *clear*. His manager Dana gets a one-tap card in chat: “Sam — meals $36.00, in policy (limit $40), receipt attached. [Approve] [Reject] [Ask].” Dana taps Approve. The claim is written to the payable sheet your bookkeeper reads, and Sam gets a note that it’s approved. The whole thing took Dana three seconds.

Now Sam buys a $400 software seat. The checker sees software needs a sign-off regardless of amount — that’s a *review*. The request goes to the finance lead, not Dana, with the reason: “software, $400, policy requires finance sign-off.” The cost of running all this is about $2.40 a month at SMB volume. The cost of *not* running it is the rubber-stamped pile where the one claim that should have been questioned sailed through with the rest.

DESIGN RULES THAT SHAPED EVERY DECISION

- A human approves every payment. The system reads, checks, and routes — it never moves money.
- Four outcomes, always. Clear, confirm, review, reject. There is no fifth.
- Every approval card ships with the reason — in policy or not, and why. The approver never has to dig.
- Quiet hours are respected. An approval request that lands at 2am is a worse request.
- The policy lives in Drive. Changing a limit or an approver doesn't need a deploy.
- Every claim and every decision is logged. Audit a reimbursement next year and the trail is there.

Why this shape

Most small teams handle expenses in one of three ways: a shoebox of receipts reconciled monthly, a spreadsheet nobody enjoys filling in, or a manager's memory of who spends what. The shoebox is slow and the team waits weeks to get paid back. The spreadsheet drifts the moment someone forgets a row. And the memory fails the day the claim is bigger than usual and nobody questions it because they're busy.

The setup above keeps the policy in a doc the team already edits, but adds a small system that *reads* each claim against that policy the moment it arrives and does

the easy work for you. The in-policy lunch is a one-tap confirm, not a meeting. The over-limit software buy is routed to the person who should actually see it, with the reason attached. And nothing is ever paid without a human tapping Approve — the system makes the decision fast, it doesn’t make the decision for you.

The next four posts walk through each piece in turn: how a claim gets submitted, how it gets checked against policy, how it finds its approver, and how it gets paid once approved. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 28, 2026 PART 2 OF 7 · EXPENSE APPROVER SERIES ~4 MIN READ

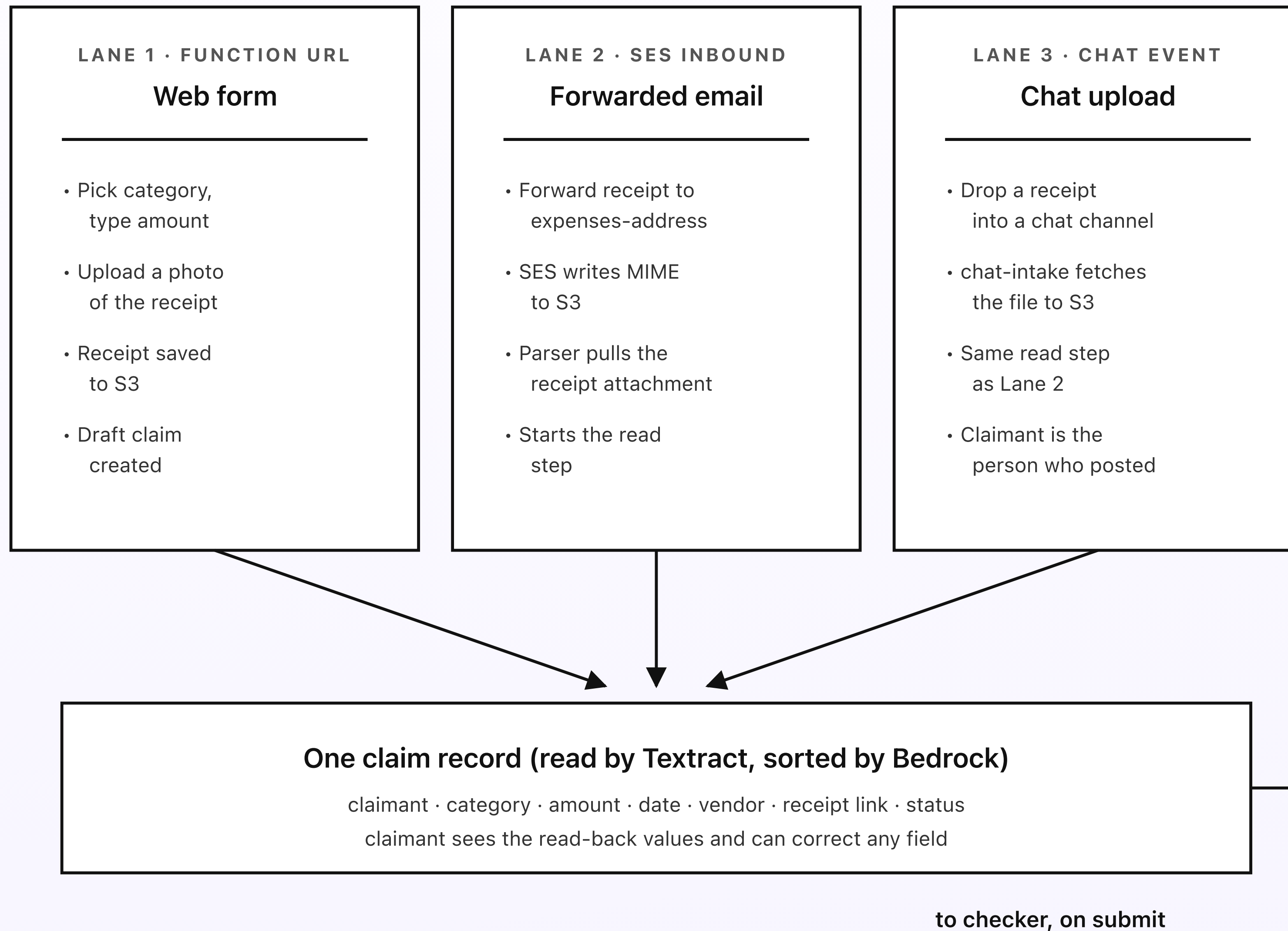
How an expense claim gets submitted

The approver can only check what it's been given. So the first job is making it dead simple to hand it a claim, because the easier the submission, the fewer receipts end up lost in a coat pocket. There are three ways a claim gets in: somebody fills a short web form, somebody forwards a receipt email, or somebody drops a photo into a team chat. The point of having three is that people submit differently, and a claim that never gets submitted is the only one the system can't help with.

KEY TAKEAWAYS

- Three intake lanes feed one claim record: a web form, a forwarded email, and a chat upload.
- Each receipt is read by Textract into amount, date, and vendor.
- Bedrock Haiku 4.5 sorts the receipt into a category so the claimant doesn't have to be exact.
- The claimant sees the read-back values and can correct them before the claim is filed.
- However it arrived, every claim becomes one record in the same shape.

Three lanes into one claim



However a claim arrives, it becomes one record in the same shape — the lanes are just doors into the same room.

Fig 2. Three lanes converge on one claim record. The web form, the forwarded email, and the chat upload all end with Textract reading the receipt and Bedrock sorting the category. The claimant gets to fix any misread value before the claim goes to the checker.

Lane 1: the web form

The simplest lane, and the one most people use. A short form sits behind a Lambda Function URL — no app to install, just a link the team bookmarks. Pick a category from a short list, type the amount, snap or upload a photo of the receipt, hit submit. The form Lambda writes the receipt image to `s3://ea-receipts/` and creates a draft claim in DynamoDB with the claimant's identity (they signed in), the category they picked, and a pointer to the receipt.

The form is deliberately tiny. Three fields and a photo. Anything the team has to think about is a thing that gets done wrong or skipped, so the form asks for the minimum and lets the read step in a moment fill in the rest.

Lane 2: forwarded receipt email

Half of business receipts arrive as email in the first place — the ride app, the SaaS invoice, the online order confirmation. Forcing the team to download those and re-upload them to a form is busywork. So there's a dedicated inbound address — something like `expenses@your-company.com` — set up via Amazon SES. Forward the receipt email to it and the system takes over.

SES writes the raw email to `s3://ea-raw-mime/`. The S3 PUT triggers a parser Lambda that walks the email, finds the receipt (a PDF attachment, an image, or even the receipt text in the body), and starts the same read step the form uses.

The claimant is matched from the "From" address of the forward, so the system knows whose claim it is. If the match is unclear, the claim is held and the system asks the sender to confirm before it goes anywhere.

Lane 3: chat upload

Some teams live in chat. The receipt photo gets dropped into a channel with a one-line "client lunch" before the person even thinks of it as an expense. Lane 3 catches those. A chat-intake Lambda listens for file uploads in a configured expenses channel, fetches the image, writes it to S3, and starts the read step. The person who posted is the claimant.

This lane is the most opt-in of the three. A team that doesn't use chat for this loses nothing; a team that does gets to submit a claim without leaving the tool they're already in.

Reading the receipt, once, the same way

Whatever lane a claim came in through, it hits the same read step. Amazon Textract reads the receipt image and returns the printed values — most importantly the total amount, the date, and the vendor name. Textract handles photos, PDFs, and scans natively, so a phone snap of a crumpled receipt works fine. Then a short Bedrock Haiku 4.5 call reads the vendor and line items and sorts the receipt into one of your policy categories: "this looks like meals," "this looks like a taxi," "this looks like software." The category the claimant picked is treated as a hint, not the final word — if the receipt clearly says otherwise, the system flags the mismatch.

The claimant always sees the read-back before the claim is filed: “We read $36.00, today, Olive & Vine, category meals — correct?” They can fix any field with a tap. This matters because a misread amount is the one error you really don’t want flowing into an approval: too low and the claimant is shorted, too high and the business overpays. A human confirming the read keeps both honest.

Why everything becomes one record

Three lanes in, but only one shape of claim record the rest of the system ever sees. That’s deliberate. The checker in the next post, the routing after it, and the audit trail all work on a single claim shape: claimant, category, amount, date, vendor, receipt link, status. The lanes are just convenient doors; once a claim is through any of them, it’s indistinguishable from a claim that came through the others. One shape means one set of rules to reason about and one place to look when somebody asks “what happened to my claim?”

Next post: how the checker reads a claim, sorts it against policy, and picks one of four outcomes.

PART 3 OF 7

MAY 28, 2026 PART 3 OF 7 · EXPENSE APPROVER SERIES ~5 MIN READ

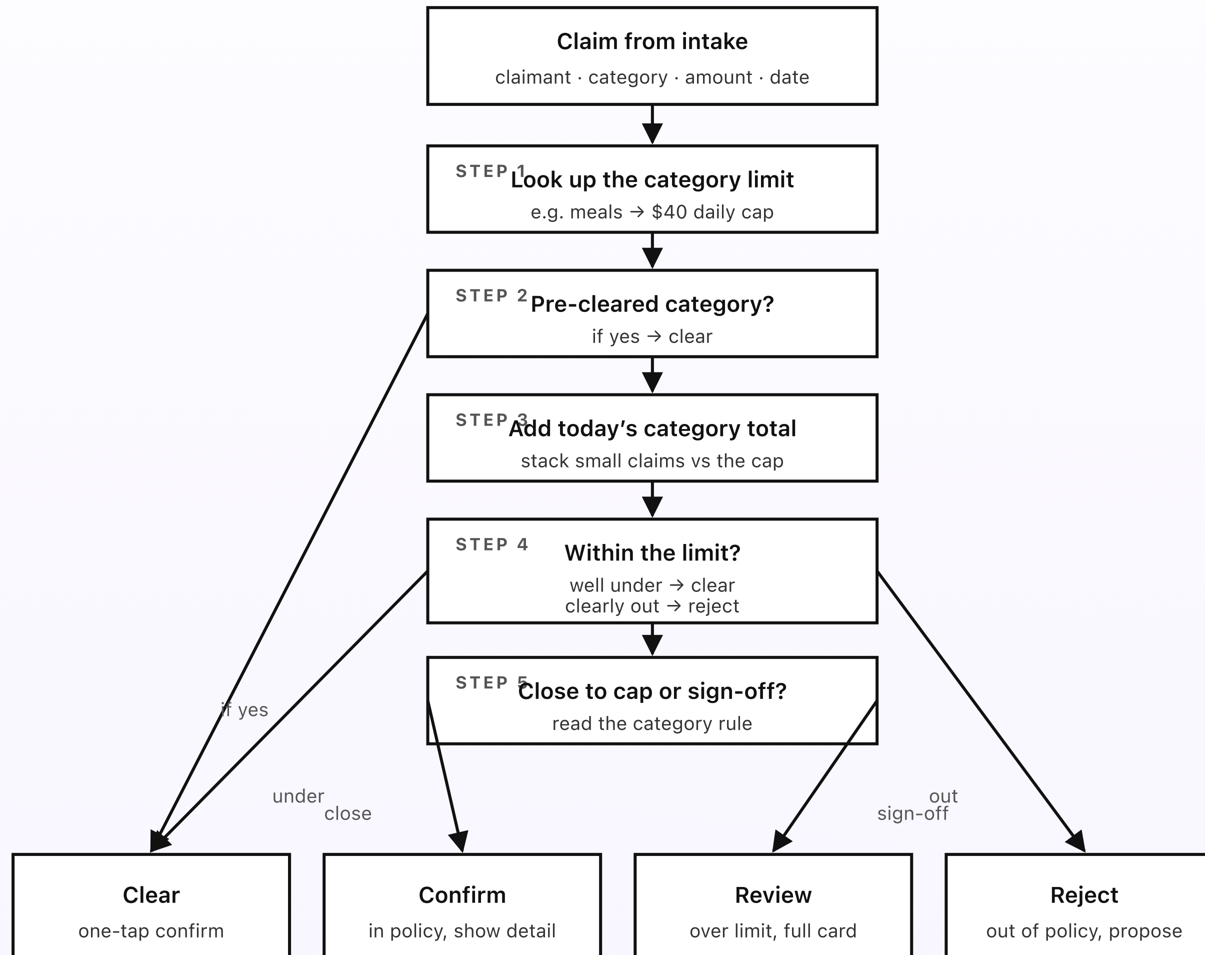
How an expense claim gets checked against policy

A claim arrives, read and sorted. Now the checker has to decide what to do with it. It reads the claim, looks at the category, compares the amount against the limit in the policy doc, and decides whether to clear it, ask for a quick confirm, send it for a full review, or propose a reject. The whole decision is plain Python. No model. The policy doc holds every limit, where an owner can edit it without a deploy.

KEY TAKEAWAYS

- The checker runs the moment a claim is submitted — it's event-driven, not batched.
- Per-category limits live in the policy doc — meals $40/day, taxis $30/trip, software needs sign-off.
- Four outcomes per claim: clear, confirm, review, reject.
- DynamoDB tracks the running daily total per claimant so a stack of small meals can't slip the cap.
- The checker never calls a model on the limit comparison. That part is entirely deterministic.

The decision flow, per claim



The policy doc holds every limit — change one and the next claim uses the new value.

Fig 3. The checker's decision tree, per claim. Five steps decide which of four outcomes applies. The policy doc holds every limit; the checker only enforces them.

$40 meals isn't magic, it's in the doc

The policy doc has one short section per category. Each section names the rule in plain prose: "Meals: up to $40 per person per day, receipt required. Taxis and rideshare: up to $30 per trip, receipt required. Software and subscriptions: any amount, but needs a manager's sign-off. Client entertainment: up to $150, needs finance sign-off above $80. Anything over $250 in any category goes to finance." The numbers are the caps. The phrase "needs sign-off" is what tips a claim from a quick confirm into a full review.

The limits exist for a reason. The $40 meal cap is roughly a fair lunch and keeps the everyday claims flowing without a meeting. The software sign-off catches the seat that quietly renews at $400 a year. The $250 ceiling is the line above which a second set of eyes is always worth it. Different categories carry different risk; the limits reflect that.

Per-claimant overrides exist too. The policy doc can name a person or a role with a different cap — a field sales rep whose meals cap is $60 because they're always on the road, say. The checker reads the override first and falls back to the category default. This is the right escape hatch for the people whose normal spend genuinely differs from the team's.

Four outcomes, always

Every claim lands in exactly one of four buckets. The names are simple on purpose.

- **Clear.** In policy, comfortably under the limit, receipt present. The manager gets a one-tap confirm card — they can approve in a tap, or open it if they want a closer look. Most everyday claims are clears.
- **Confirm.** In policy, but worth showing the detail — close to the cap, or a category that always wants a glance. Same one-tap card, but the amount and receipt are front and centre so the approver isn't approving blind.
- **Review.** Over a limit, or a category that requires a sign-off. The full approval card goes to the right person — often finance rather than the line manager — with the reason spelled out: "$60 over the meals cap" or "software, sign-off required."
- **Reject candidate.** Clearly outside policy — a banned category, or a receipt missing where one is required. The system proposes a reject with the reason and a draft note back to the claimant. A human still confirms it; the system never rejects a claim on its own.

State that makes the decision deterministic

The checker reads one DynamoDB table as it works: `ea-claims`, which holds every claim and its running status. For the daily-total check it queries the claimant's already-approved spend in the same category today, so three $15 lunches in one day are seen as $45 against a $40 cap, not three separate fine claims. With that one lookup, the decision logic is a few dozen lines of plain Python and zero magic. A given claim, with a given amount, a given category, and

a given day's history, always produces the same outcome. Re-running the checker produces the same result.

The category sort from Part 2 is the one place a model touched the claim, and that happened before the checker runs. By the time the checker compares an amount to a limit, the category is already settled and the math is pure arithmetic. Part 4 covers how the chosen outcome turns into a request to the right person.

Why the limit check uses no model

The checker could ask a model "does this seem reasonable?" It doesn't. Two reasons. First, the policy check is the one part of the system that has to be utterly predictable — if the doc says meals cap at $40 and the claim is $36, it clears; if it's $46, it goes to review. A model in that loop introduces variance nobody can reason about, and an expense decision the team can't explain is a decision they won't trust. Second, this runs on every single claim, and arithmetic is free while a model call is not.

Bedrock fires elsewhere — sorting the category in Part 2, and writing the monthly summary in Part 6. Not on the limit check. The checker is plain Python that reads a doc and writes an outcome.

Next post: how the chosen outcome finds the right approver, how quiet hours are honored, and what the approval card actually carries.

PART 4 OF 7

MAY 28, 2026 PART 4 OF 7 · EXPENSE APPROVER SERIES ~5 MIN READ

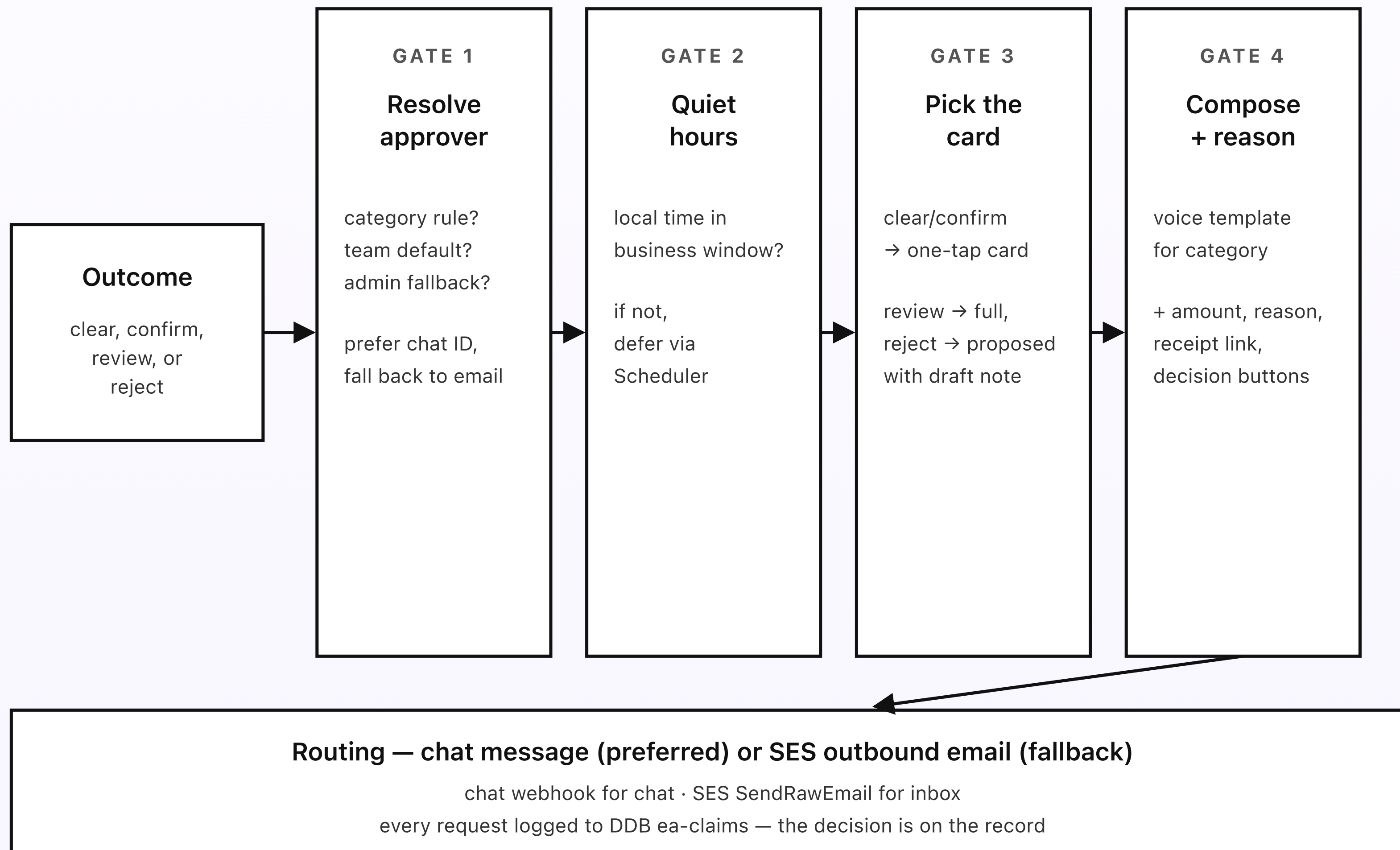
How an expense claim finds its approver

The checker picked an outcome — clear, confirm, review, or reject. Now the routing piece has to figure out who should decide it, on what channel, at what time of day, and with what reason attached. Get any of those wrong and the request is worse than useless: a $400 software buy that lands on the wrong manager, an approval card with no reason, a ping at 11pm. Four small guardrails sit between the outcome and the request landing.

KEY TAKEAWAYS

- Approver resolution: per-category rule beats team default beats fallback to the configured admin.
- Chat messages are the default; email is the fallback if no chat ID is configured.
- Quiet hours defer a request to the next business hour so nothing lands at 2am.
- Every request ships with the claim, the amount, the category, the reason, the receipt, and buttons.
- Clear and confirm get a one-tap card; review gets the full card with the policy reason spelled out.

Four guardrails on every request



Every gate is a deterministic check — no model calls, no surprise behavior on a Tuesday afternoon.

Fig 4. Four guardrails between the outcome and the approval request. Resolve the approver. Honor quiet hours. Pick the right card. Compose with the reason. Then ship via chat or email and log the request so the decision is on the record.

Gate 1: resolve the approver

Three places the routing Lambda looks for the approver, in order. First, the per-category approval rule in the policy doc — “software goes to the team manager,” “anything over $250 goes to finance.” The rule can depend on the outcome and the amount, so a clear meal and an over-limit software buy from the same person go to different approvers. Second, the claimant’s team default approver (“everyone on the design team reports to Dana”). Third, the configured admin fallback — the person who set up the approver and gets any claim that didn’t resolve. The fallback should never fire in steady state; if it does, the weekly digest names every claim that hit it so the policy doc can be fixed.

Once routing knows which person to ask, it looks up their delivery preference. The voice doc maps each approver to a chat ID if one is set, otherwise to an email address. Chat is preferred because a card with Approve, Reject, and Ask buttons is faster to act on than an email link. Email is the fallback so nobody’s claim falls through the cracks.

Gate 2: quiet hours

Claims arrive at all hours — the late client dinner gets submitted at 10pm, the airport taxi at midnight. The approval request for those shouldn’t buzz the approver’s phone the moment it lands. Gate 2 reads the policy doc’s quiet-hours

setting (default 7pm to 8am, configurable per business). If the current local time is in the quiet window, the request creates a one-off EventBridge Scheduler rule that fires at the next business-hour minute and exits without sending. The Scheduler re-invokes the routing Lambda with the same payload at the deferred time, where Gate 2 lets it through.

The claim still records as submitted instantly — the claimant knows it's in. Only the approver's ping waits for business hours. An approval that can wait until 8am is worth waiting until 8am for.

Gate 3: pick the right card

Not every outcome deserves the same card. A clear and a confirm get a compact one-tap card: the claimant, the amount, the category, a thumbnail of the receipt, and an Approve button right there. The approver can tap Approve without opening anything, or tap through if they want the detail. A review gets the full card: the same fields plus the reason it needs a closer look, the policy limit it exceeded, and the claimant's daily total in that category so far. A reject candidate gets a proposed-reject card — the reason and a draft note back to the claimant — that a human confirms or overrides.

The point of three card shapes is to spend the approver's attention where it matters. The $36 lunch shouldn't demand the same scrutiny as the $400 software buy, and making them look identical trains people to rubber-stamp both.

Gate 4: compose with the reason, then ship

The voice doc has one message template per category and outcome: a short message with placeholders for the claimant, amount, category, the reason line, and the receipt link. The routing Lambda fills the placeholders, attaches the decision buttons, and ships the message via the chat webhook. The webhook URL itself lives in Secrets Manager.

For email fallback, the same template is wrapped in a small HTML email with the same fields and a link that, when clicked, hits a Function URL that records the decision — the email equivalent of the chat buttons.

The reason line is the part that earns its place. “In policy — meals $36, under the $40 cap” tells the approver they can clear it with confidence. “$60 over the meals cap — review” tells them exactly what to weigh. The approver never has to reverse-engineer why this claim reached them; the reason travels with it.

Every request — chat or email, any outcome — updates the claim’s row in `ea-claims` in DynamoDB with the resolved approver, the channel, and the reason. The next post’s decision handler reads that row when the approver acts.

Why the guardrails exist

None of these gates are exotic. They’re the small care a thoughtful finance person would take by hand — send this to the right approver, don’t buzz them at midnight, match the scrutiny to the size of the claim, and always say why. Putting them in code as four small sequential gates makes them part of the design, not something you’re trusting whoever wrote any one message to remember.

Next post: how a claim actually gets paid once the approver acts — how Approve writes it to the payable sheet, how Reject sends a reason back, and how every action is logged so a reimbursement is auditable a year later.

PART 5 OF 7

MAY 28, 2026 PART 5 OF 7 · EXPENSE APPROVER SERIES ~5 MIN READ

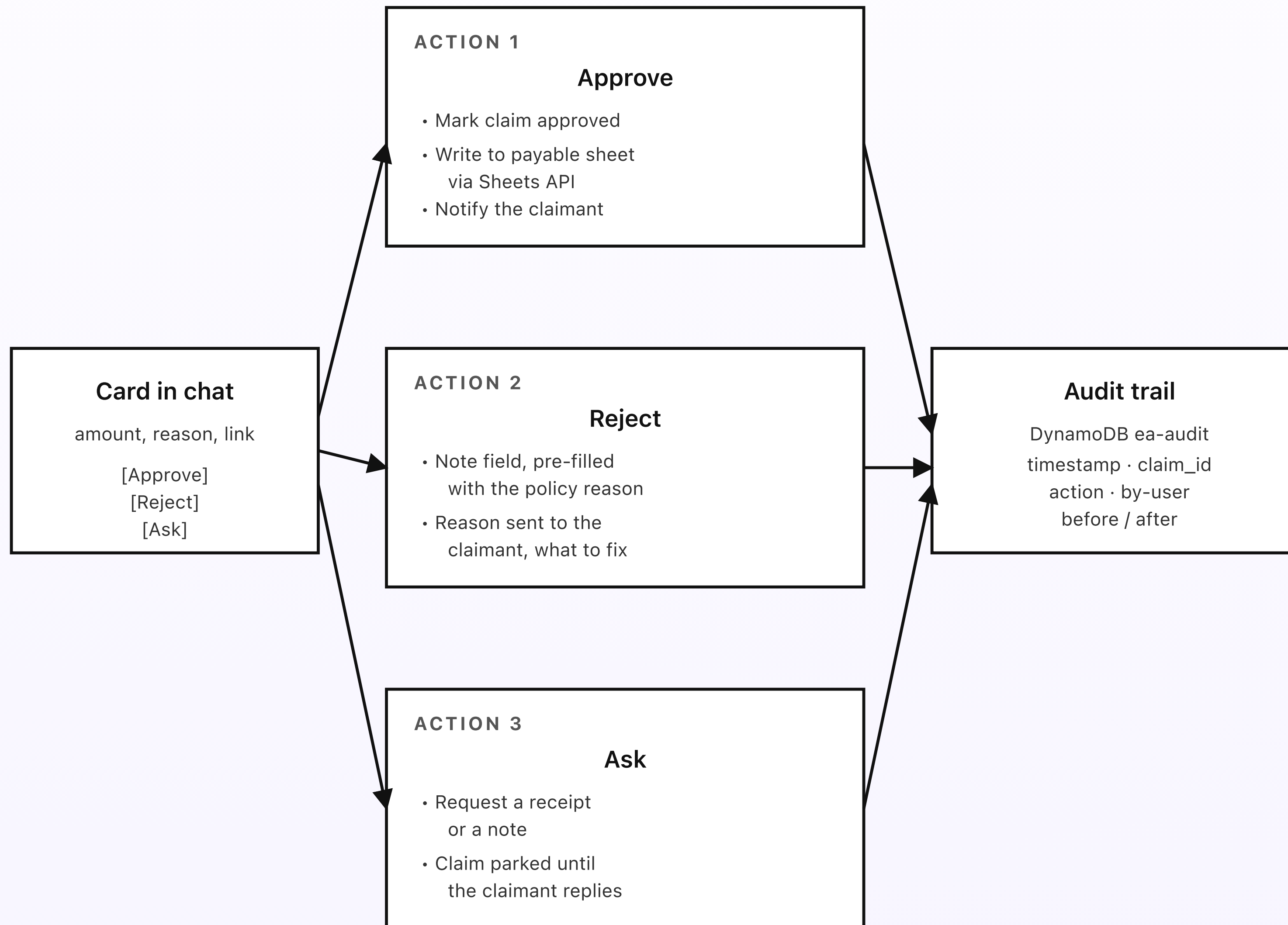
How an expense claim gets paid

An approval card lands in Dana's chat at 8:03am. Sam's client lunch, $36, in policy, receipt attached. There are three buttons: Approve, Reject, Ask. What happens when she taps one? The honest answer is "it depends on which one." This post walks through the three things an approver can do — approve, reject, ask — and how the claim record, the payable sheet, and the audit trail all stay in sync. And the one thing that never happens: the system moving money on its own.

KEY TAKEAWAYS

- Three actions per card: *approve* (write to the payable sheet), *reject* (reason back to the claimant), *ask* (request more).
- Approve writes a row to a payable sheet that your bookkeeper or payroll run reads — the system never pays anyone.
- Reject sends a plain-English reason to the claimant so they know why and what to fix.
- Ask requests a missing receipt or a note, and parks the claim until the claimant replies.
- Every action writes a before-and-after snapshot to the audit table, so each decision is reversible.

Three actions on the card



Approve writes to the payable sheet — it never sends a payment. A person runs the reimbursement.

Fig 5. Three actions per card, three different effects. Approve writes the claim to the payable sheet. Reject sends a reason to the claimant. Ask requests more and parks the claim. Every action writes to the audit trail, and none of them moves money.

Action 1: approve (the most common)

Dana reads the card — Sam's $36 lunch, in policy, receipt attached — and taps *Approve*. The tap submits to a Function URL Lambda. Three things happen, in order. First, the claim's row in `ea-claims` is marked `approved` with Dana's identity and the timestamp. Second, a row is appended to the payable sheet in Drive via the Sheets API: employee, amount, category, approver, date, and a link to the receipt. Third, an `action: approved` row is written to `ea-audit` with the user, timestamp, and the claim snapshot. Sam gets a one-line note: "Your $36 lunch claim is approved."

The payable sheet is the boundary the system never crosses. It does not call a bank. It does not trigger a transfer. It writes a clean, approved, audit-trailed row, and whoever runs payroll or the weekly reimbursement batch reads that sheet and pays from it. Keeping the system one step back from the money is the whole point — the riskiest action in the pipeline stays in human hands, and the system's job is to make that human's decision fast and well-grounded.

Action 2: reject (with a reason)

Sometimes the claim shouldn't be paid. The $90 dinner that's well over the meals cap with no business reason given. The personal item that slipped into the wrong category. The duplicate of a claim already submitted last week. Dana taps *Reject*.

A small note field opens, pre-filled with the policy reason the checker already worked out ("$50 over the $40 meals cap"), which Dana can edit or add to. On send, a Function URL Lambda marks the claim `rejected`, sends the reason to Sam so he knows exactly why and what to do ("split the personal portion and resubmit," say), and writes a `rejected` audit row.

The reason matters because a rejection without one just breeds a follow-up question and a grumble. "Rejected" on its own feels arbitrary; "rejected because it's $50 over the meals cap, please resubmit the business portion" is a clear next step. The claimant is never left guessing.

Action 3: ask (the "I need more")

Sometimes the approver can't decide yet. The receipt photo is too blurry to read. The amount is fine but there's no note about which client the dinner was for. The category looks off. Dana taps *Ask*, picks what she needs ("clearer receipt," "which client?"), and the claim is parked in a `waiting` state. A message goes to Sam asking for exactly that.

When Sam replies — through the same submit lane he used in Part 2, by re-uploading the receipt or adding the note — the claim re-enters the checker, gets re-evaluated against policy, and comes back to the approver with the new information attached. The claim doesn't leave the system or get lost; it just pauses until the missing piece arrives. *Ask* is the pressure valve that keeps a half-complete claim from being either wrongly approved or unfairly rejected.

Every action is logged, every action is reversible

The `ea-audit` table records every approve, reject, and ask with the user who took the action, the timestamp, and a snapshot of the claim before and after. If a claim was approved in error — wrong amount, wrong person, approved twice — an admin can run an “undo last action” that reads the previous-state snapshot, restores the claim, and removes the row from the payable sheet if it hasn’t been paid yet. The undo is itself an audit row, so the trail of edits stays clean.

This kind of reversibility matters because expenses get audited. When a year-end review asks “who approved this $400 software buy and why?”, the audit trail answers in one query: the approver, the time, the reason, and the policy that applied. The trail is the memory the business has when the people who made the call have moved on.

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 policy check itself is almost free.

PART 6 OF 7

MAY 28, 2026 PART 6 OF 7 · EXPENSE APPROVER SERIES ~3 MIN READ

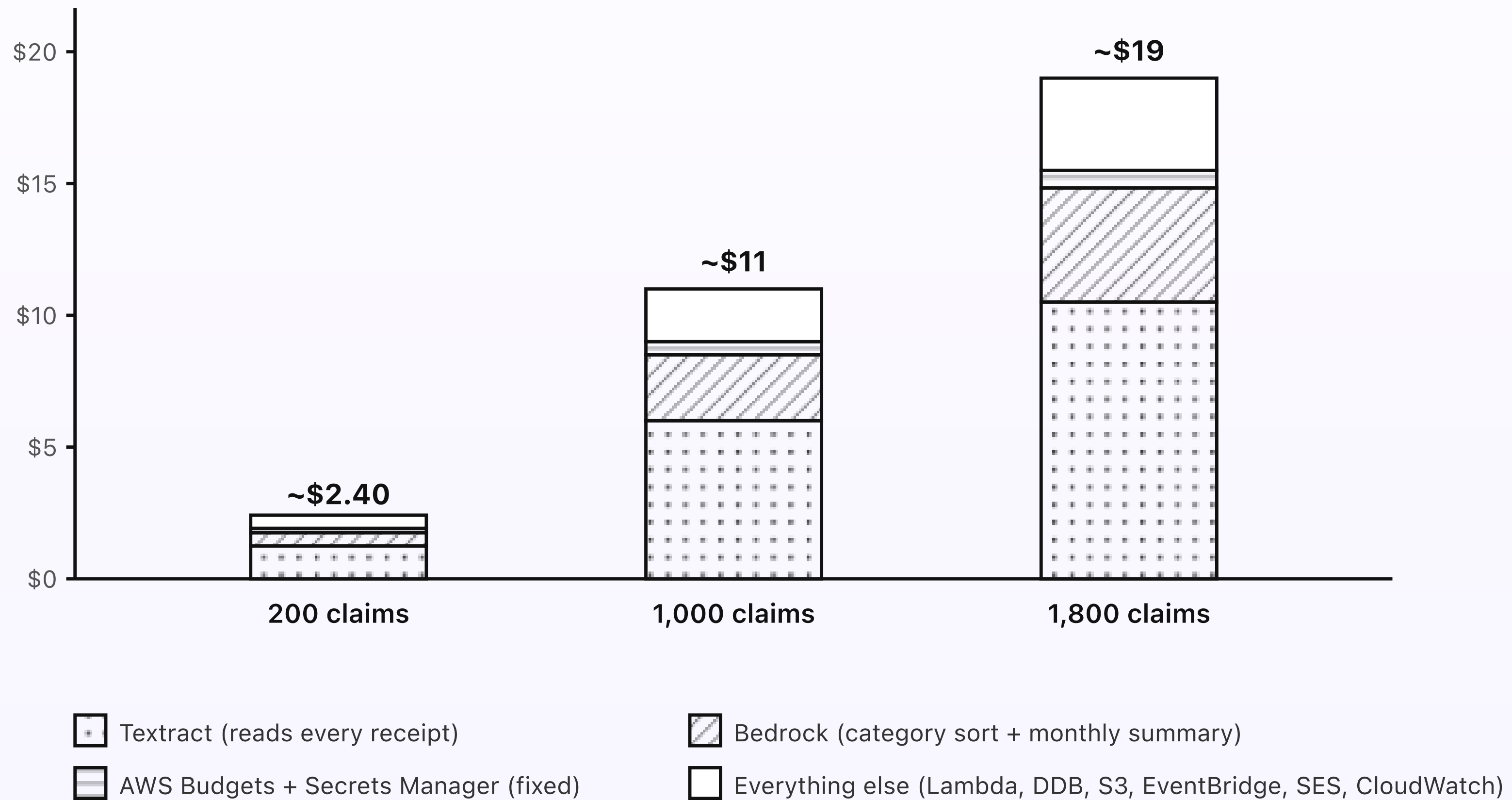
What the expense approver costs

The approver is one of the cheaper systems in this whole series. The expensive-sounding part — reading a photo of a receipt — is a few tenths of a cent per claim. The policy check that decides each claim's fate is plain Python and costs almost nothing. Bedrock fires once per claim to sort the category and once a month for a 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 claims a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The policy check costs pennies — no model calls on the decision.
- Textract reading each receipt is the dominant cost; the category sort is a small Bedrock sliver.
- At 1,000 claims a month the bill is around $11. At 1,800 it's around $19.

Cost at three volumes



Textract is the dominant cost — and even that is fractions of a cent per receipt read.

Fig 6. Monthly cost at three claim volumes. Textract reading each receipt is the largest slice; the category sort is a small Bedrock sliver. The policy check itself is plain Python and barely registers.

Where the dollars actually go

Textract (the bulk). Each claim has one receipt, and Textract reads it once. Receipt reading runs a few tenths of a cent per page, and most receipts are one page. At 200 claims that's around a dollar; at 1,800 it's a handful of dollars. This is the single largest line, and it's still small — reading a receipt by machine is cheap, and it only happens once per claim.

Bedrock (the category sort). One small Haiku 4.5 call per claim reads the receipt text and picks a category. A few hundred input tokens and a handful of output tokens, so a fraction of a cent each. The monthly summary is one larger call: a board-ready paragraph on the month's spend by category. Bedrock lands at cents to a couple of dollars across these volumes.

Lambda runtime. The intake Lambda, the checker, the routing Lambda, and the decision handler each run for a fraction of a second per claim. At any of these volumes the Lambda total is well under a dollar.

DynamoDB on-demand. Two small tables: `ea-claims` and `ea-audit`. A few reads and writes per claim. Pennies a month at any of these volumes.

S3 + storage. The receipt images plus the raw MIME from any forwarded receipts. A few megabytes a month at SMB volume. Effectively free, and lifecycle rules push old receipts to cheaper storage.

SES. Inbound for the forwarding lane and outbound for email-fallback approvals and claimant notices: $0.10 per thousand messages either way. A few cents a month at this scale.

EventBridge. Routing claim events plus the deferred one-offs from the quiet-hours gate. A few events per claim. Pennies.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the submit form and the approve buttons.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Every Lambda sleeps until a claim or a decision wakes it.
- **A Knowledge Base.** The policy is short structured rules, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **A model on the policy check.** The decision is plain Python comparing an amount to a limit. Bedrock fires only on the category sort and the monthly summary.

How the cost scales

Textract and Bedrock grow linearly with claim count, because each claim has one receipt to read and one category to sort. Lambda and DynamoDB grow linearly too but stay tiny. So the bill at 5,000 claims a month is around $50; at 10,000 it's around $100. Past those volumes you're a bigger business than this design targets, and you'd look at batching Textract or caching common-vendor categories — but those are optimizations, not redesigns.

Set an AWS Budgets alarm at $25/month so anything unusual pages you before the bill matters. The approver's normal-volume bill stays well under that ceiling.

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

PART 7 OF 7

MAY 28, 2026 PART 7 OF 7 · EXPENSE APPROVER SERIES ~8 MIN READ

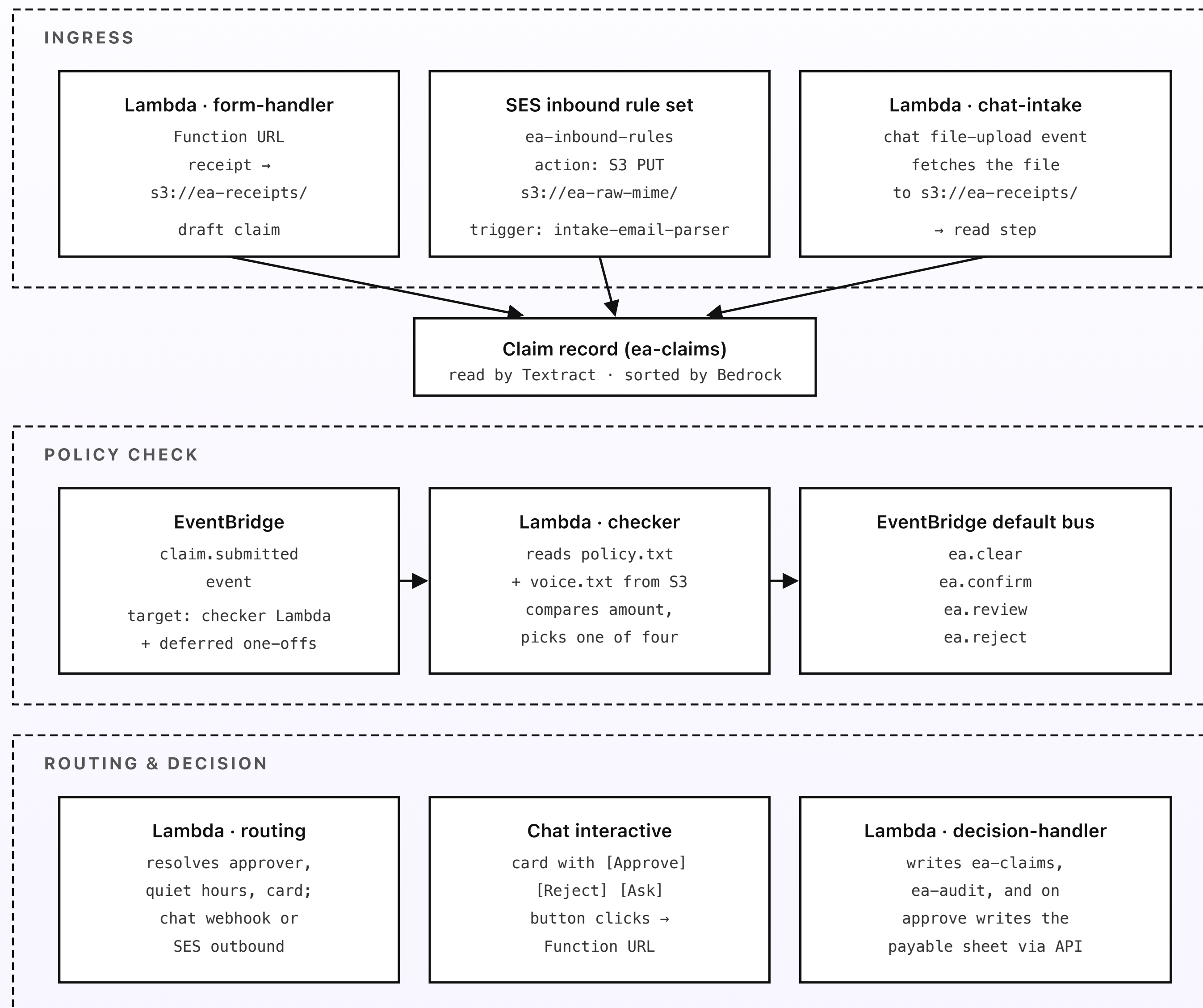
Engineering reference: the expense approver architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the Textract flow, EventBridge config, the DynamoDB schemas, and the chat 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, Textract, Bedrock cross-Region inference, and EventBridge 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 claim waiting an hour for approval, not a regional outage. One AWS account dedicated to the approver (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



A human approves every payment — and every interaction is logged to ea-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the claim record), policy check (the checker emitting an outcome event), routing and decision (the card ships and the approver's response is recorded). Every Lambda is event-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.

- **`form-handler`** — Lambda Function URL, behind sign-in (the form posts the claimant's identity token). Validates the form fields, writes the uploaded receipt to `s3://ea-receipts/<claim-id>`, creates a draft row in `ea-claims`, and starts the read step by invoking `intake-read`. Memory: 256 MB. Timeout: 15 s.
- **`intake-email-parser`** — S3 PUT trigger on `s3://ea-raw-mime/`. Parses the MIME tree, extracts the receipt (PDF, image, or body text), writes it to `s3://ea-receipts/`, matches the claimant from the forwarding "From" address against the directory, and starts the read step. If the claimant can't be matched, holds the claim and emails the sender to confirm. Memory: 512 MB. Timeout: 60 s.
- **`chat-intake`** — triggered by a chat file-upload event in the configured expenses channel (events delivered to a Function URL; the handler verifies the chat signing secret). Fetches the uploaded file to `s3://ea-receipts/`, sets the claimant to the posting user, and starts the read step. Memory: 256 MB. Timeout: 30 s.

- **`intake-read`** — the shared read step. Runs Textract via `AnalyzeExpense` (the receipt-specialized API that returns total, date, and vendor as typed fields) on the receipt in S3. Then calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to sort the receipt into a policy category. Writes the read-back values to `ea-claims` and surfaces them to the claimant for confirmation. On confirm/submit, emits `claim.submitted` to EventBridge. Memory: 512 MB. Timeout: 60 s.
- **`checker`** — EventBridge rule on `claim.submitted`. Reads `s3://ea-policy-source/policy.txt` and `voice.txt`, reads the claimant's same-day category total from `ea-claims`, computes the outcome, and emits one event per claim: `ea.clear`, `ea.confirm`, `ea.review`, or `ea.reject`, with the claim context as the payload. Memory: 256 MB. Timeout: 30 s. *No Bedrock calls.*
- **`routing`** — EventBridge rule on the four outcome events. Resolves the approver, checks quiet hours, picks the card shape, formats from the voice template, and ships via chat webhook (`ea/chat/webhook` in Secrets Manager) or SES `SendRawEmail`. On a quiet-hours defer, creates a one-off EventBridge Scheduler rule that re-invokes `routing` at the next available business minute. Updates the claim in `ea-claims` after a successful send. Memory: 256 MB. Timeout: 30 s.
- **`decision-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a chat signature on the request body. Triggered by chat button clicks (Approve/Reject/Ask) and by email-link clicks. Writes to `ea-claims` and `ea-audit`; on approve, appends a row to the payable sheet via the Sheets API; on ask, parks the claim in `waiting` and messages the claimant. Memory: 256 MB. Timeout: 15 s.

- **`digest`** — EventBridge Scheduler target, weekly Sunday 6pm. Reads `ea-claims` for the past week; sends a digest to a configured chat channel summarizing what was approved, rejected, and still waiting. 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 `ea-claims` and `ea-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph board narrative on spend by category; emails it via SES to the configured stakeholder list. Memory: 512 MB.

Storage

- **DynamoDB** · **`ea-claims`** — one row per claim. PK `claim_id`; GSI on `(claimant, category, date)` for the same-day total query. Attributes: `amount`, `vendor`, `category`, `status` (draft/submitted/waiting/approved/rejected), `outcome`, `approver`, `reason`, `receipt_key`. On-demand.
- **DynamoDB** · **`ea-audit`** — one row per write action of any kind. PK `(claim_id, ts)`; attributes: `action` (approve/reject/ask/undo), `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **S3** · **`ea-receipts`** — receipt images and PDFs, one prefix per claim. Versioning enabled. Lifecycle to a cheaper storage class at 90 days; expiry at 7 years (tax-retention friendly).
- **S3** · **`ea-policy-source`** — mirrored policy and voice docs as plain text. Versioning enabled so a bad policy edit can be rolled back in one click.

- **S3** · `ea-raw-mime` — raw inbound MIME from forwarded receipts. Lifecycle to a cheaper class at 30 days; expiry at 7 years.

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-read` for the category sort, and `summary` for the monthly board narrative. The heavier `anthropic.claude-sonnet-4-6-20250930-v1:0` is not used — sorting a receipt into one of a dozen categories doesn't justify it; Haiku 4.5 handles it cheaply.
- **Embeddings.** Not used. The policy is short structured rules; 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 doesn't call Bedrock; the category sort is one small call per claim.

Textract

- **API.** `AnalyzeExpense` — the receipt-and-invoice specialized call that returns typed summary fields (total, tax, date, vendor) plus line items, which is exactly the shape a receipt needs. Synchronous for single-page receipts; the async `StartExpenseAnalysis` path is used only for multi-page PDFs.
- **Fallback.** If `AnalyzeExpense` returns low confidence on the total, the read step falls back to plain `DetectDocumentText` and a Bedrock pass to pull the amount, then always surfaces the value to the claimant to confirm before the claim is filed.

EventBridge config

- `ea-claim-submitted` — rule on `claim.submitted` on the default bus. Target: `checker` Lambda.
- `ea-outcome-routing` — rule matching `ea.clear`, `ea.confirm`, `ea.review`, `ea.reject`. Target: `routing` Lambda.
- `ea-weekly-digest` — Scheduler, `cron(0 18 ? * SUN *)` in TZ. Target: `digest` Lambda.
- `ea-monthly-summary` — Scheduler, `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `routing` 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. `expenses.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `ea-inbound-rules`: one rule with recipient `expenses@your-company.com` → spam scan → S3 PUT to `s3://ea-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-email-parser`.
- SES outbound for the email-fallback approvals and the claimant notices: verify a sender identity at `expenses@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **checker role:** `s3:GetObject` on the policy and voice keys; `dynamodb:Query` + `GetItem` on `ea-claims`; `events:PutEvents` on the default bus. *No* `bedrock:*`.
- **routing role:** `events:CreateSchedule` for the deferred one-offs; `secretsmanager:GetSecretValue` on the chat webhook secret; `ses:SendRawEmail` from the verified sender identity; `dynamodb:UpdateItem` on `ea-claims`; outbound network access to the chat host.
- **decision-handler role:** `dynamodb:PutItem` on `ea-audit` and `dynamodb:UpdateItem` on `ea-claims`; `secretsmanager:GetSecretValue` on the Sheets-API service-account secret; outbound network access to `sheets.googleapis.com` for the payable-sheet write.
- **intake-read role:** `s3:GetObject` on `ea-receipts`; `textract:AnalyzeExpense` + `StartExpenseAnalysis`; `bedrock:InvokeModel` on the Haiku ARN; `dynamodb:UpdateItem` on `ea-claims`; `events:PutEvents`.
- **intake-email-parser and chat-intake roles:** `s3:GetObject` / `PutObject` on the raw-MIME and receipts buckets; `secretsmanager:GetSecretValue` on the chat signing secret; permission to invoke `intake-read`.

Chat interactive flow

The chat incoming webhook is the simplest delivery surface but doesn't support interactive button responses. So the approval cards are posted via the chat

platform's post-message Web API instead, with interactive blocks containing the Approve/Reject/Ask buttons. Button clicks are sent by the platform to the configured interactivity request URL, which is the `decision-handler` Function URL. `decision-handler` verifies the chat signing secret on the inbound request, parses the action id (`approve`, `reject`, `ask`), opens a note modal if needed (Reject and Ask open modals; Approve is one-tap), and processes the response when the modal is submitted.

The chat app needs message-write and direct-message scopes and the interactivity URL configured. The bot token lives in Secrets Manager under `ea/chat/bot-token`. The signing secret is `ea/chat/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 (a claim that never gets an outcome is a claim that silently stalls); decision-handler signature-verification failures > 5/hour (might mean the chat secret rotated); a claim sitting in `submitted` for > 3 business days (an approval nobody acted on).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `ea-cost-alarm` subscribed to the on-call admin's email and chat.

Config and secrets

Service-account credentials for the Sheets API (the payable sheet write) and the Drive API (the policy/voice doc sync) live in Secrets Manager under `ea/google/sa`. Chat bot token and signing secret live under `ea/chat/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, quiet-hours window, per-team default approvers, and admin fallback all live in Parameter Store under `/ea/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment. A small `drive-sync` Lambda (Scheduler, every 15 minutes) mirrors the policy and voice docs to `s3://ea-policy-source/` so the checker reads from S3, not Drive, on every claim.

Deploy

GitHub Actions with OIDC into a deploy role — no long-lived AWS keys — running AWS SAM. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for `ea-receipts` and `ea-policy-source` so a bad edit can be rolled back in one click, and keep the payable-sheet write idempotent (key the row on `claim_id`) so a retried approve can never double-pay. SAM with a single template fits the whole surface: around nine Lambdas, two DynamoDB tables, three S3 buckets, two EventBridge rules 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.