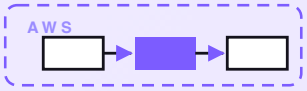
7-PART SERIES · FREE COMPANION



Applicant screener

A serverless screener that helps a small team handle a flood of job applications fairly. It reads each resume against the role's must-haves — which you write down — gives a clear yes/maybe/no with the reasons it found in the resume, and routes strong matches to the hiring manager with a short summary. A human makes every decision; it never rejects anyone on its own, and it sticks to job-related criteria only. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Applicant screener

PART 1 OF 7

MAY 21, 2026 PART 1 OF 7 · APPLICANT SCREENER SERIES ~5 MIN READ

An applicant screener on AWS for a few dollars a month

You post one job and 300 resumes show up in a week. Most are a poor fit; a handful are exactly who you need; and the good ones are buried somewhere in the pile. A small team can't read all of them carefully, so the careful reading gets skipped, and good people get missed. This post walks through the design of a small screener that reads every resume against the must-haves *you* wrote down, gives a clear yes/maybe/no with the reasons, and hands the strong matches to a human with a short summary. It never rejects anyone on its own. A person makes every call.

KEY TAKEAWAYS

- Three sources for applications: an apply inbox, a careers-form upload, and a job-board export.
- Every applicant ends in one of three piles: yes, maybe, or no — each with the reasons from the resume.
- It only judges against the job-related must-haves you write down. Name, age, school, and photo are stripped first.
- A human makes every decision. The screener sorts and explains; it never rejects anyone on its own.
- Designed on AWS for about $2/month at typical small-business volume.

The whole system on one page

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

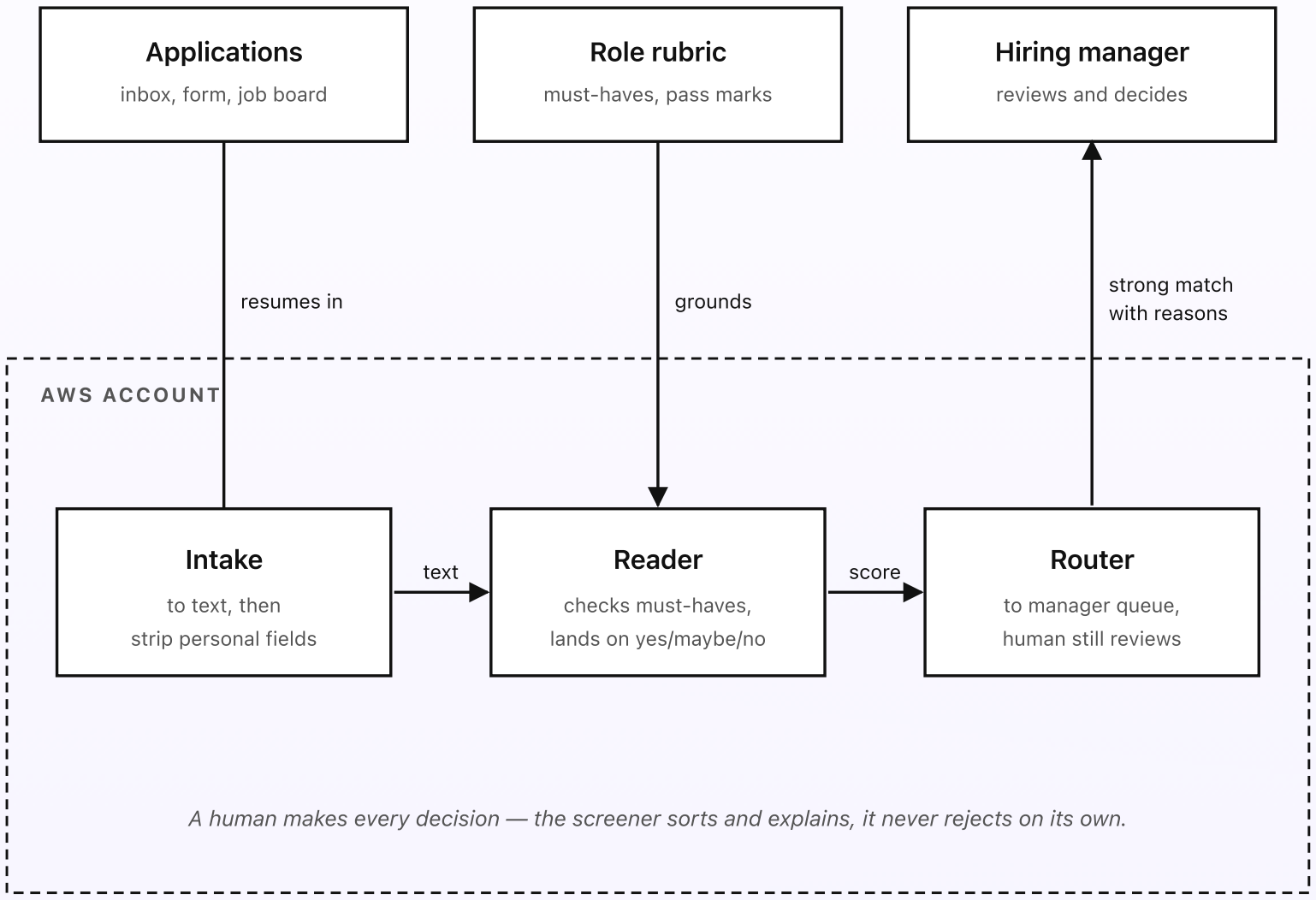

Applications
inbox, form, job board
Role rubric
must-haves, pass marks
Hiring manager
reviews and decides
resumes in
grounds
strong match
with reasons
AWS ACCOUNT
Intake
to text, then
strip personal fields
text
Reader
checks must-haves,
lands on yes/maybe/no
score
Router
to manager queue,
human still reviews
A human makes every decision — the screener sorts and explains, it never rejects on its own.

Fig 1. Three sources outside, three pieces inside AWS. Applications flow in from an apply inbox, a careers form, and a job-board export. The Reader checks each resume against your must-haves and lands on yes, maybe, or no. The Router sends strong matches to a human who decides.

What you set up once (the outside)

- **Applications.** Three ways a resume gets in, all covered in Part 2. The first is an apply inbox — candidates email a resume to a dedicated address. The second is the upload box on your careers page. The third is an export from a job board you already post to. All three land as a resume file (PDF, Word, or plain text) plus a few facts like the role applied for and when it arrived.
- **A role rubric.** One short Google Doc per role in a Drive folder. It lists the *must-haves* — the things a person genuinely needs to do the job, each written as a plain, job-related requirement ("has run payroll for a team of 10+", "holds a valid forklift license", "can work Saturdays"). It lists the *nice-to-haves*. And it sets the *pass marks*: how many must-haves earn a yes, and how many earn a maybe. The doc also names the personal fields to strip before scoring — name, age, gender, photo, home address, and school name — so the screener can't judge on them even by accident.
- **Hiring manager.** The person who reviews strong matches and makes every actual decision. They get a short summary per candidate, the resume, and the reasons each must-have was met or missed, with three buttons: advance, hold, and pass. The screener routes; the manager decides.

What runs on every application (the inside)

- **The intake.** Three sources feed one queue. Each resume is turned into plain text — a Word or text file reads straight through; a scanned PDF goes through Amazon Textract first to pull the words off the page. Then the intake strips the personal fields named in the rubric, so what reaches the reader is the work history and skills, not the name or the photo. A copy of the original is kept for the human; only the stripped text is scored.
- **The reader.** Runs once per application. It reads the stripped text and, for each must-have in the rubric, decides whether the resume meets it — quoting the exact line that does, or noting that it's missing. This is the one place a model earns its keep: reading messy resume prose and matching it to a plain requirement. The reader does *not* pick the label. Plain Python counts the met must-haves against your pass marks and lands on *yes*, *maybe*, or *no*. The cut-off is yours, and it's the same for every applicant.
- **The router.** Reads the label and decides where the candidate goes. A *yes* becomes a strong match: it gets a short plain summary and lands in the hiring manager's review queue at the top. A *maybe* lands in the same queue, lower down. A *no* is parked in a reviewable list — not deleted, not auto-rejected — so a human can still look. Every routing decision, and the criteria behind it, is written to DynamoDB so the whole round can be audited later.

In plain words

You're hiring a bookkeeper. The must-haves are: two years of bookkeeping, has used Xero or QuickBooks, and can work in-office two days a week. A resume comes in by email. The intake turns the PDF into text and strips the name, age, and the candidate's photo. The reader checks each must-have: "Bookkeeper at

Acme, 2021–2024" meets the first, "Reconciled accounts in QuickBooks Online" meets the second, and there's no mention of in-office days — missing. Two of three met. Your rubric says two must-haves earns a maybe, three earns a yes. So this one is a maybe, with the two quotes and the one gap shown plainly. The hiring manager opens the queue, sees the maybe, reads the gap, and decides it's worth a quick call to ask about in-office days. The screener didn't reject anyone. It saved the manager from reading 300 resumes to find this one.

The cost of running this is about $2 a month at small-business volume. The cost of *not* running it is the strong candidate buried on page nine that nobody reached, or the slow, uneven reading that lets bias creep in when a tired human skims 200 resumes at 6pm.

DESIGN RULES THAT SHAPED EVERY DECISION

- A human makes every decision. The screener sorts into yes/maybe/no and explains; it never rejects anyone on its own.
- It judges against the job-related must-haves only. Name, age, gender, photo, address, and school are stripped before scoring.
- Every label ships with reasons — the resume line that met each must-have, or a note that it's missing. No black box.
- The pass marks are yours. The model reads; your rubric sets the cut-off, the same for every applicant.
- The rubric lives in Drive. Changing a must-have or a pass mark doesn't need a deploy.
- Every decision is logged with the criteria used, so any hiring round can be audited later.

Why this shape

Most small teams screen resumes one of two ways: somebody reads every one (which doesn't scale past a few dozen and gets less careful as the pile grows), or somebody skims for a keyword and tosses the rest (which is fast, unfair, and misses good people who phrased things differently). Both get worse under pressure. The keyword skim is the more dangerous one — it feels rigorous and isn't. It rewards resume buzzwords over real fit, and it quietly filters on whatever the skimmer's eye lands on first, which is often a name or a school.

The setup above keeps the must-haves where the team already writes them — a doc — but adds a small system that reads *every* resume the same careful way, against the same job-related list, with the personal fields removed. It explains every call so a human can check it. It sends the strong matches up with a summary so the manager spends their time on the candidates worth time. And it never, ever closes the door on anyone by itself. The screener does the reading; the human does the deciding.

The next four posts walk through each piece in turn: how an application gets read, how an applicant gets scored, how a strong match reaches the hiring manager, and how a hiring decision actually gets made. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 21, 2026 PART 2 OF 7 · APPLICANT SCREENER SERIES ~4 MIN READ

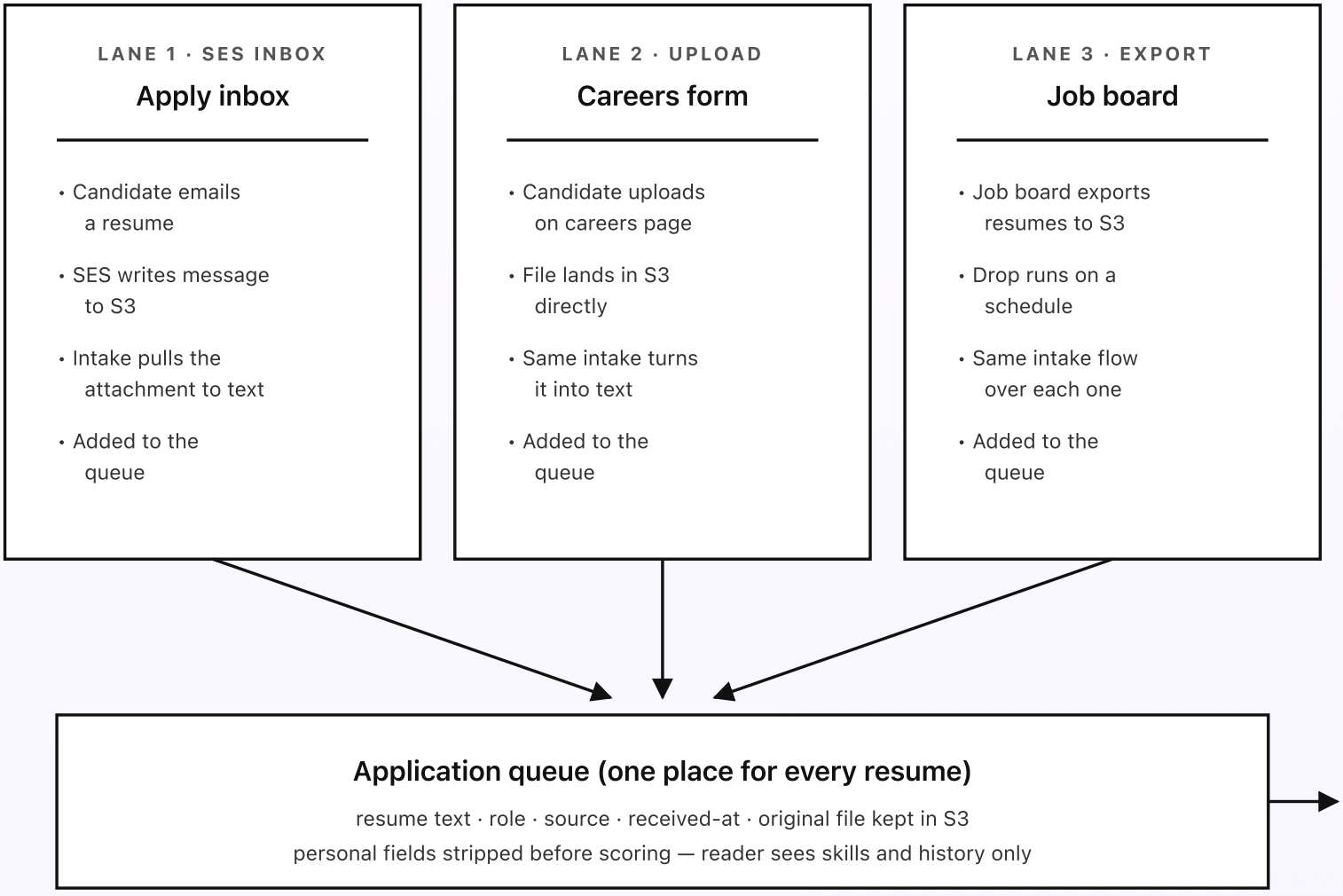
How an application gets read

The screener can only screen what reaches its queue. So the first job is getting every application in, no matter how a candidate chose to apply. There are three lanes: somebody emails a resume to your apply address, somebody uploads one through your careers form, or a resume arrives from a job board you post to. Whatever the lane, the resume is turned into plain text and the personal fields that don't belong in a fair screen are removed before any scoring happens. The candidate's original is kept safe for the human; the reader only ever sees the stripped version.

KEY TAKEAWAYS

- Three intake lanes feed one queue: an apply inbox, a careers-form upload, and a job-board export.
- Word and text resumes read straight through; scanned PDFs go through Textract to pull the words off the page.
- Personal fields — name, age, gender, photo, address, school — are stripped before the resume is scored.
- The original resume is always kept for the human; only the stripped text is sent to the reader.
- Every application lands in one queue, so “why was this scored that way?” has one place to look.

Three lanes into one queue



Every application lands in one queue — personal fields stripped before scoring, original kept for the human.

Fig 2. Three lanes converge on one queue. The apply inbox, the careers form, and the job-board export all feed the same place. Each resume is turned into plain text and the personal fields are stripped before scoring; the original is kept in S3 for the human.

Lane 1: the apply inbox

Set up a dedicated address — something like `apply@your-company.com` — through Amazon SES. A candidate emails their resume to it. SES writes the raw message to `s3://as-raw-mail/`. The S3 write triggers the intake Lambda, which walks the message to the resume attachment and turns it into plain text. A Word file or a text resume reads straight through. A PDF that was made from a document reads directly too. A PDF that's really a scanned photo of a printed page goes through Amazon Textract first — Textract reads the words off the image so the resume becomes text the reader can work with.

This lane is the one most candidates use without being told to. People email resumes. Meeting them where they already are means you don't lose applicants to a form they didn't feel like filling out.

Lane 2: the careers form (the tidy one)

Your careers page has an upload box: the candidate picks the role, attaches a resume, and submits. The file is sent straight to `s3://as-uploads/` along with the role they applied for and the date. The same intake Lambda runs over it — same to-text step, same Textract fallback for scanned PDFs. The form lane gives you cleaner data than the inbox (you know exactly which role they applied for, because they picked it), which is why it's worth offering even though the inbox alone would work.

Nothing about the form judges the candidate. It just collects the resume and the role. Everything that follows — the stripping, the reading, the scoring — is the same no matter which lane the resume came in through.

Lane 3: the job-board export

If you post the role to a job board, candidates apply there and the board collects their resumes. Most boards let you export those applications — either as a download or to a folder. A small sync drops each exported resume into `s3://as-jobboard/` on a schedule, and the same intake flow turns each one into text and adds it to the queue. This lane saves you from copying resumes by hand out of a job board's dashboard, which is exactly the kind of dull task that gets skipped when the pile is large.

Job-board exports are the messiest lane — formats vary, and some boards strip the resume down to their own template. The intake handles what it can and flags anything it can't read cleanly for a human to look at, rather than guessing.

Stripping the personal fields, before anything is scored

This is the step that makes the whole thing fair, so it happens before the reader ever sees the resume. The rubric names the fields to remove: name, age or date of birth, gender, photo, home address, and the name of the school the candidate attended. The intake finds and blanks those out of the text it passes on. What reaches the reader is the work history, the skills, the certifications, the dates — the things the job actually depends on.

Two things are worth saying plainly. First, the original resume is never changed; the full version is kept in S3 for the human, who will of course see the name when they decide to interview. Stripping only affects the copy that gets *scored*. Second, no stripper is perfect — a name might appear in an email address inside the resume body. The point isn't a flawless scrub; it's removing the obvious signals that have nothing to do with the job, so the reader can't lean on them. Where the screener is unsure, it strips more rather than less.

Next post: how the reader checks each must-have against the stripped text, and how plain Python turns that into a yes, a maybe, or a no using *your* pass marks.

PART 3 OF 7

MAY 21, 2026 PART 3 OF 7 · APPLICANT SCREENER SERIES ~5 MIN READ

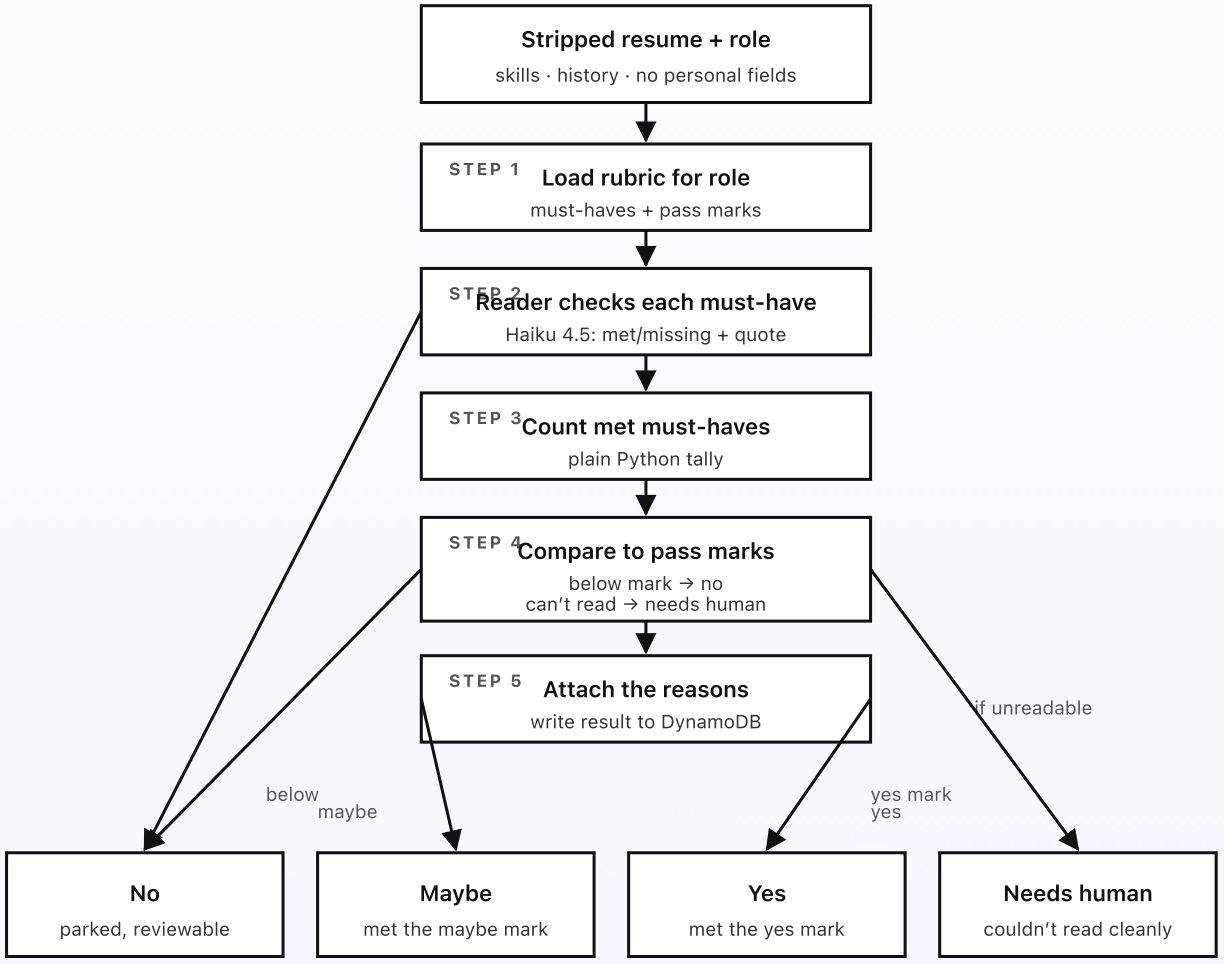
How an applicant gets scored

An application reaches the reader as stripped text and a role. The reader's job is narrow on purpose: for each must-have in the rubric, decide whether the resume meets it, and quote the line that proves it. That checking is the one place a model earns its keep. But the model never picks the label. Plain Python counts how many must-haves were met and compares that to *your* pass marks. The cut-off is yours, and it's the same for every applicant.

KEY TAKEAWAYS

- The reader checks one must-have at a time and quotes the resume line that meets it, or marks it missing.
- The yes/maybe/no label is plain Python — it counts met must-haves against the pass marks in your rubric.
- Your pass marks set the cut-off. Change a must-have or a pass mark in the doc and the next score uses it.
- Every score is saved with the per-must-have reasons, so a human can check the reasoning later.
- The model reads the resume; it never decides who passes. The rules do that.

The scoring flow, per applicant



The model reads and quotes; your pass marks decide — change a mark and the next score uses it.

Fig 3. The scoring flow, per applicant. The reader checks each must-have and quotes the proof; plain Python counts and compares to your pass marks; the result lands on yes, maybe, no, or needs-human. The rubric holds the cut-off; the code only enforces it.

What the reader actually does (and doesn't)

The reader is one Bedrock Haiku 4.5 call per application. The prompt is tight: here is a resume (stripped of personal fields), and here is a list of must-haves. For each must-have, say whether the resume meets it, and if it does, quote the exact line that shows it. If it doesn't, say so. Return a small structured result and nothing else. Do not invent experience that isn't written down. Do not consider anything outside this list.

That last instruction matters. The reader is told to judge against the must-haves and only the must-haves. It is not asked for an overall impression, a gut feel, or a ranking. Those are exactly the judgments where bias slips in. By keeping the model's job to "does this line meet this requirement, yes or no, show me the proof," the output is something a human can check in seconds rather than a score they have to trust.

Your pass marks, not the model's

Once the reader returns which must-haves were met, the model's job is done. Plain Python takes over. It counts the met must-haves and compares that count to two numbers from your rubric: the *yes mark* and the *maybe mark*. If you wrote "all four must-haves earns a yes; three earns a maybe," then four met is a yes, three is

a maybe, and two or fewer is a no. That's the whole rule. No weighting magic, no hidden model judgment, no surprise.

Putting the cut-off in plain Python and plain numbers is deliberate. It means the same resume always gets the same label. It means you can change the bar — tighten it when you have too many yeses, loosen it when the role is hard to fill — by editing one line in a doc. And it means when somebody asks "why was this person a maybe and not a yes," the answer is a number you can point at, not a model you have to defend.

Some must-haves are genuinely non-negotiable — a license the law requires, a shift the job can't flex on. The rubric can mark those as *required*. A required must-have that's missing caps the label at no regardless of the count. Even then, the candidate isn't deleted — the no is parked in a reviewable list, with the missing required item shown, so a human can still look and override if the resume simply didn't mention it.

When the reader isn't sure

Resumes are messy. Sometimes the text came out garbled from a bad scan. Sometimes a must-have is borderline — the resume says "managed a small team" and the must-have is "managed a team of 10+," with no number given. The reader is told that "not clearly met" is not the same as "met," and that a missing number is a gap, not a pass. But it's also told to flag the borderline ones rather than guess.

Anything the reader couldn't read cleanly, or where a key must-have was truly ambiguous, lands in a small *needs-human* pile instead of being forced into yes, maybe, or no. A person looks at those directly. This is the safety valve: the system

would rather ask a human than make up an answer, because a made-up “no” quietly closes a door that should have stayed open.

Every score is explained, and saved

For each application, the result written to DynamoDB has four things: the label, the count of met must-haves, the per-must-have reasons (met or missing, with the quoted line), and a stamp of which rubric version was used. That record is what the hiring manager sees on the candidate card in the next post, and it’s what an audit reads months later. There is no part of the score that isn’t written down and tied to a job-related requirement.

Next post: how a strong match reaches the hiring manager — the short summary, the four guardrails before routing, and the human review step that sits between a score and any decision.

PART 4 OF 7

MAY 21, 2026 PART 4 OF 7 · APPLICANT SCREENER SERIES ~5 MIN READ

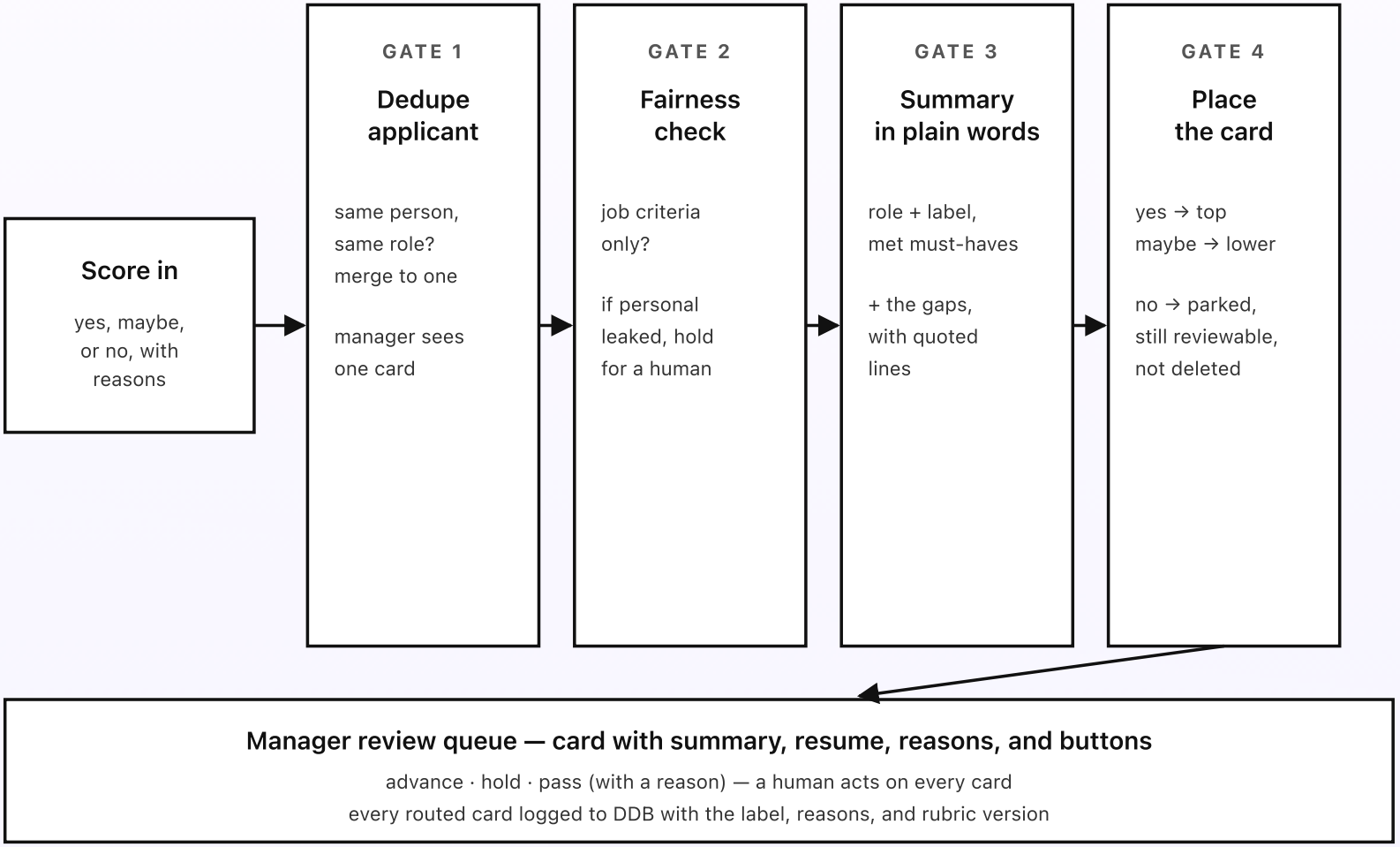
How a strong match reaches the hiring manager

The reader landed on a label — yes, maybe, or no. Now the router has to get the right candidates in front of the right person, in a form they can act on, without doing the one thing the system must never do: decide. A strong match should arrive with a short summary and the reasons, ready for a human to weigh. A weak one should still be reachable, not deleted. Four small guardrails sit between a score and a routed candidate.

KEY TAKEAWAYS

- Routing order: dedupe the applicant, run the fairness check, write the summary, then place the card for a human.
- A yes goes to the top of the manager's review queue; a maybe goes lower; a no is parked but still reviewable.
- The summary is short and job-related — the met must-haves, the gaps, and the role. No gut-feel, no ranking-by-name.
- The screener never sends a rejection. It routes cards; the hiring manager decides on every one.
- Every routed card is logged with the label, the reasons, and the rubric version used.

Four guardrails before a candidate is routed



The screener routes cards — it never sends a rejection and never decides; a human acts on every card.

Fig 4. Four guardrails between the score and the routed card. Dedupe the applicant. Run the fairness check. Write the plain summary. Place the card for a human. Then a person advances, holds, or passes — the screener never decides.

Gate 1: dedupe the applicant

People apply more than once. The same person emails a resume *and* fills out the careers form, or applies on two job boards you both export from. Without a check, the manager sees the same candidate three times and wonders which card to trust. Gate 1 looks for an applicant who already has a card for this same role — matching on the contact details and the resume content, not the stripped name — and merges the duplicates into one card. The manager sees a single candidate with a note that they came in through more than one lane.

Dedupe is a courtesy and a correctness fix at once. It saves the manager's time, and it stops a candidate from accidentally getting three bites at the queue just because they applied in three places.

Gate 2: the fairness check

Before a card is routed, the system double-checks the thing it cares most about: that the score was made on job-related grounds only. It confirms the personal fields were stripped, and it scans the reasons the reader produced for anything personal that slipped through — a mention of age, a school name buried in a quoted line, a comment about a gap that reads like an assumption about someone's life. If the reasons are clean and tied to must-haves, the card passes. If

something personal leaked in, the card is held for a human to look at instead of being routed on a tainted score.

This gate is cheap and it's the backstop. The stripping in Part 2 does the heavy lifting; this check catches the leftovers before they reach a decision. Held cards go to the same needs-human pile, and the leak is logged so the stripping rules can be tightened.

Gate 3: the short, plain summary

A manager opening a queue of candidates shouldn't have to re-read each resume to know why it's there. Gate 3 writes a short summary: the role, the label, the must-haves that were met (with the quoted line for each), and the must-haves that were missing. That's it. No "strong candidate, great culture fit" — that's the kind of vague praise that hides bias. The summary is a plain accounting of which job requirements the resume met and which it didn't, so the manager can decide on facts.

The summary is built from the reader's structured result, so it can't introduce a claim the resume didn't support. If the reader marked a must-have as missing, the summary says missing. The manager always sees the full resume too — the summary is a fast way in, not a replacement for reading the candidate when they're worth reading.

Gate 4: place the card, never reject

The last gate decides where the card goes — not whether the candidate is in or out. A *yes* lands at the top of the hiring manager's review queue, where the strong matches cluster. A *maybe* lands lower in the same queue. A *no* is parked in a reviewable list, sorted so the near-misses are easy to revisit if the role stays open. Nothing is deleted, and crucially, **no rejection email is ever sent by the screener.** If a candidate is to be turned down, a human does that, deliberately, from the card.

The whole queue lives behind a Function URL the manager opens in a browser — one page, the cards in order, each with the summary, the resume, the reasons, and three buttons. Every card that gets placed is written to DynamoDB with its label, its reasons, and the rubric version used, so the round is auditable. The next post covers what those three buttons do, and why every one of them is the human's action, not the system's.

Next post: how a hiring decision actually gets made — advance, hold, and pass, each logged, with the override that lets a human overturn any label the screener produced.

PART 5 OF 7

MAY 21, 2026 PART 5 OF 7 · APPLICANT SCREENER SERIES ~5 MIN READ

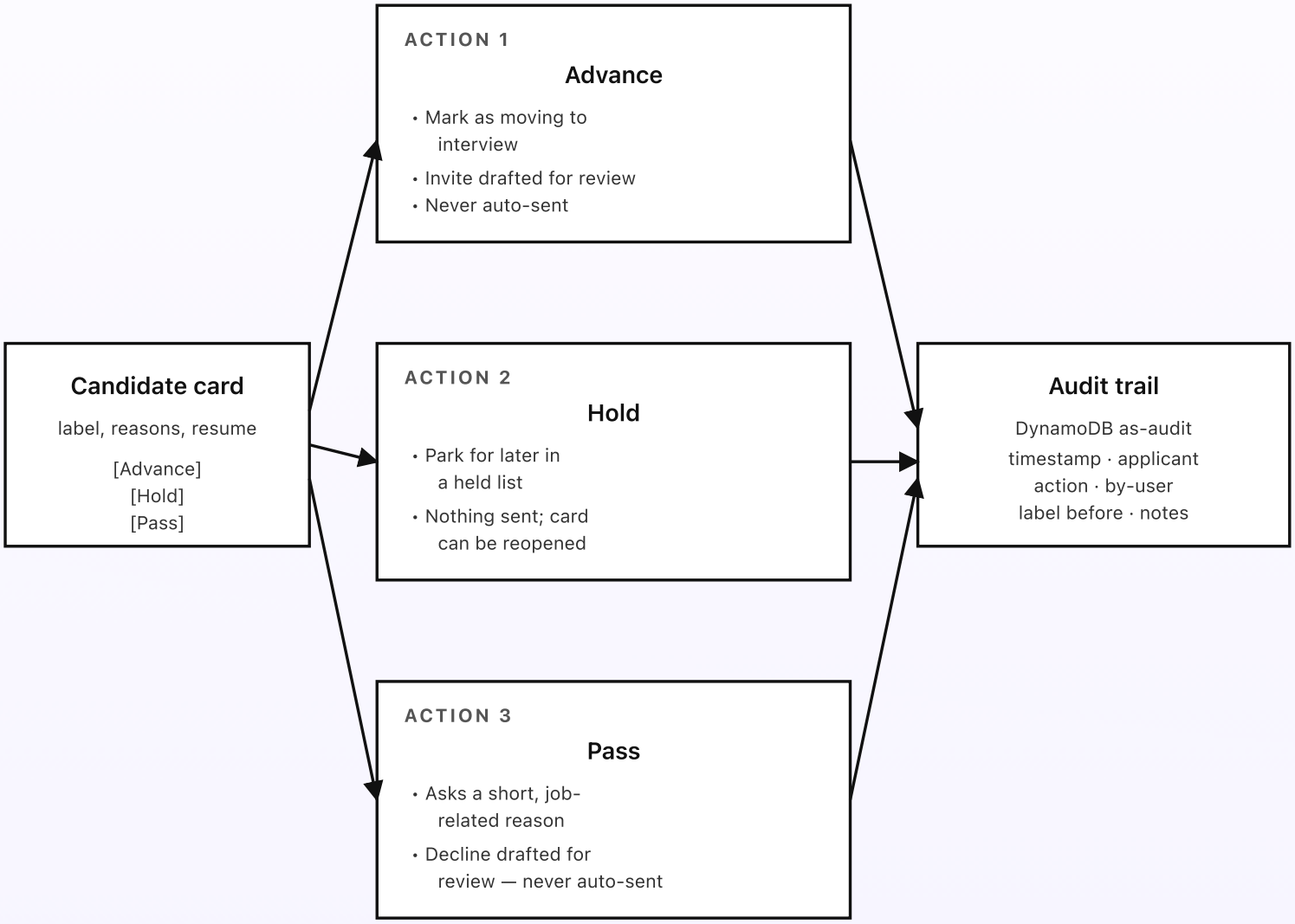
How a hiring decision gets made

A card sits at the top of the hiring manager's queue. It shows the role, the label the screener landed on, the must-haves met with their quoted lines, the gaps, and the full resume. There are three buttons: advance, hold, and pass. What happens when the manager taps one? The honest answer is that the system does the bookkeeping — but the *decision* is entirely the person's. This post walks through the three actions, how each is logged, and how a human can overturn any label the screener produced.

KEY TAKEAWAYS

- Three actions per card: *advance* (move to interview), *hold* (park for later), *pass* (turn down, with a reason).
- Every action is the human's. The screener does no advancing, holding, or passing on its own.
- A manager can open a parked "no" and advance it — the screener's label is a suggestion, not a gate.
- Pass asks for a short, job-related reason, so a turn-down is never a silent or unexplained one.
- Every action writes an audit row with the user, the time, the label before, and the action taken.

Three actions on a card



The screener's label is a suggestion — a human can advance a parked no, and every override is logged.

Fig 5. Three actions per card, three different effects. Advance moves the candidate to interview. Hold parks them for later. Pass turns them down with a job-related reason. Every action is the human's, and every action writes to the audit trail.

Action 1: advance (move to interview)

The manager reads the card, agrees the candidate is worth a conversation, and taps *Advance*. The button submits to a Function URL Lambda. The candidate is marked as moving to interview, and an `advanced` row is written to the audit trail with the manager's name, the time, and the label the screener had landed on. Optionally, the system drafts a friendly interview invite — pre-filled with the role and a couple of time slots — and shows it to the manager to edit and send. The draft is never sent on its own; a human reads it and presses send.

That "never auto-sent" rule holds for everything that leaves the building. The screener can prepare a message to save typing, but a person always chooses to send it. Reaching out to a candidate is a relationship, and the system doesn't get to start one by itself.

Action 2: hold (park for later)

Plenty of candidates are neither an obvious yes nor an obvious no. The manager isn't ready to interview them but doesn't want to turn them down — maybe they're a strong maybe to revisit after the first round of interviews, maybe the role might split into two. *Hold* parks the candidate in a held list. Nothing is sent. The screener's label isn't changed. The card can be reopened any time and acted on later.

Hold exists so the manager never feels forced into advance-or-reject on a candidate they're genuinely unsure about. A held candidate is a deliberate "not yet," recorded as such, rather than a card that quietly falls off the bottom of the queue.

Action 3: pass (turn down, with a reason)

Sometimes the answer is no, and a human decides it. *Pass* opens a small box asking for a short, job-related reason — "doesn't meet the required license," "not enough hands-on experience for this level," "role needs on-site, candidate is remote-only." On save, the Function URL Lambda records the pass with that reason and writes a `passed` row to the audit trail. As with advance, it can draft a courteous decline message for the manager to review and send — never auto-sent.

Asking for a reason isn't bureaucracy. It does two things. It nudges the manager to base the turn-down on the job, not a hunch — if you can't name a job-related reason, that's worth noticing. And it gives the audit trail something real: months later, "why did we pass on this person" has a plain answer tied to the role, not a blank. A pass is still always a person's decision; the screener only ever suggested a label.

Overturning a screener label

The screener's yes/maybe/no is a starting point, not a verdict. A parked *no* is not a closed door — the manager can open the reviewable list, read a candidate the screener scored low, and advance them anyway. Maybe the resume understated

the experience. Maybe a must-have was met in a way the reader didn't catch. Whatever the reason, the human wins. When a manager advances a candidate the screener marked *no* (or passes one it marked *yes*), that override is written to the audit trail as exactly that: a human overriding the suggestion.

Those override rows are worth watching. If managers routinely advance candidates the screener said no to, the rubric's must-haves or pass marks are probably wrong — too strict, or missing a way people phrase real experience. The override log is how you learn that and fix the doc, instead of trusting a cut-off that's quietly screening out good people.

Every action is logged, every action is reversible

The audit table records every advance, hold, and pass with the user who acted, the time, the screener's label at the moment of the action, and the reason where one was given. If a card is acted on by mistake, it can be reopened and re-decided, and that correction is itself an audit row. The trail is the memory of the hiring round: who decided what, on what job-related grounds, and when. That's exactly what you want to be able to show if anyone ever asks how a role was filled fairly.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at small-business volume; Part 6 explains exactly where the dollars go and why one model read per application keeps it cheap.

PART 6 OF 7

MAY 21, 2026 PART 6 OF 7 · APPLICANT SCREENER SERIES ~3 MIN READ

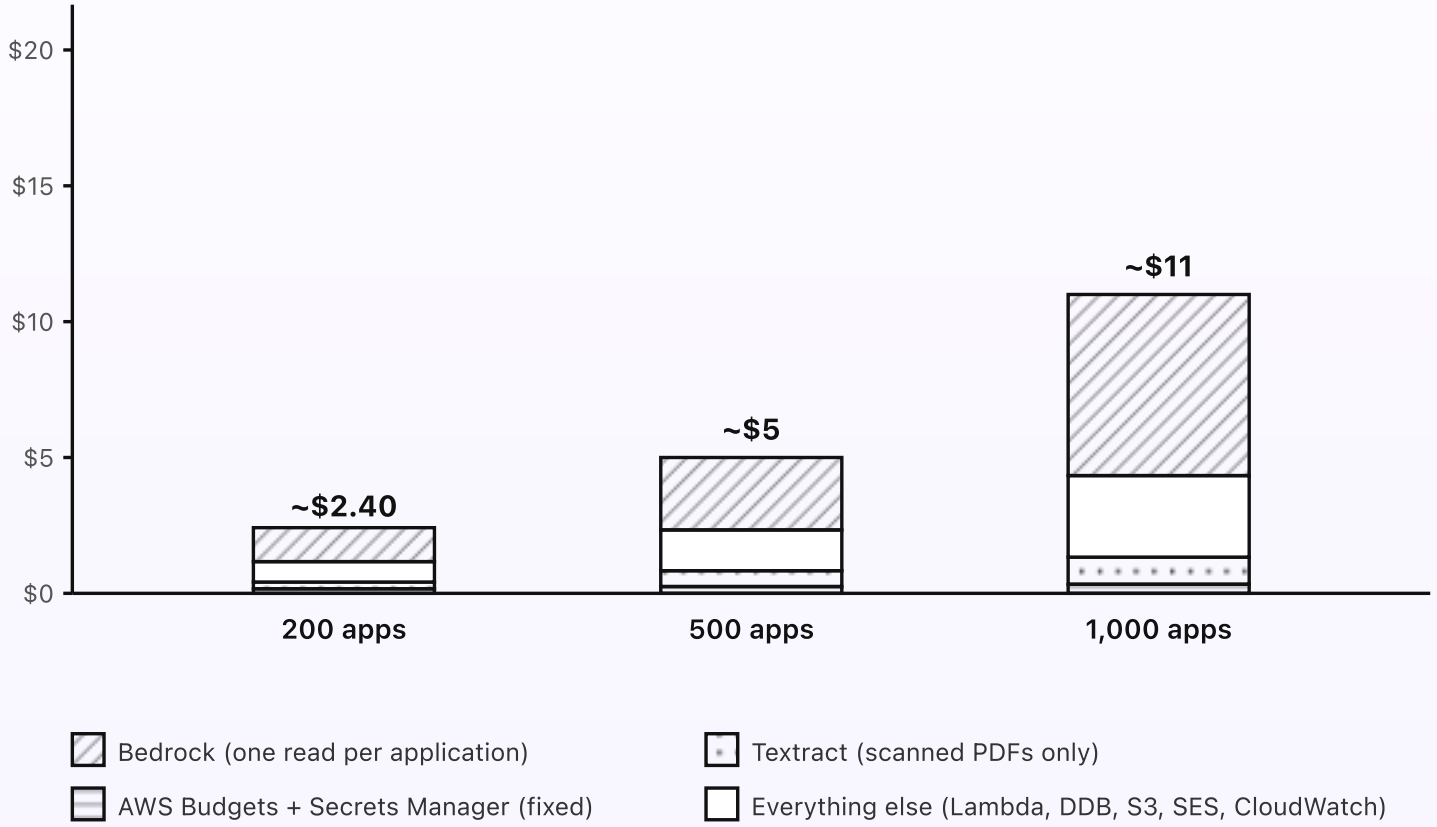
What the applicant screener costs

The screener is a cheap system to run. Each application is read once by a model, its result is written to a couple of small tables, and the card is placed in a queue a human opens in a browser. There's no always-on part, nothing waiting around, nothing in a private network running up a meter. At typical small-business volume the bill is a couple of dollars a month, fixed cost essentially zero. The one cost that grows with hiring is the per-application model read — and even that is a fraction of a cent each.

KEY TAKEAWAYS

- Around $2.40/month at typical small-business volume (around 200 applications a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The dominant cost is one Bedrock read per application — a fraction of a cent each.
- Textract fires only when a resume arrives as a scanned PDF, not on every application.
- At 500 applications the bill is around $5. At 1,000 it's around $11.

Cost at three volumes



The per-application model read is the dominant cost — and even that is a fraction of a cent per application.

Fig 6. Monthly cost at three application volumes. Bedrock is the dominant slice because every application is read once; Textract is small because only some resumes arrive as scanned PDFs; the fixed costs barely move. The bill scales with how many people apply.

Where the dollars actually go

Bedrock (the bulk). One Haiku 4.5 read per application: a few thousand input tokens (the stripped resume plus the must-haves) and a few hundred output tokens (the per-must-have result). That's a fraction of a cent each. At 200 applications a month it's about a dollar and a half; at 1,000 it's a handful of dollars. Because the read happens once per application and never on a loop, the cost tracks hiring activity exactly — a quiet month costs almost nothing.

Lambda runtime. The intake, the reader call, the router, and the Function URL behind the manager's queue and buttons. Each runs briefly, only when there's an application to handle or a button to process. Pennies a month at all three volumes.

DynamoDB on-demand. A couple of small tables for the scores, the cards, and the audit trail. Reads when the manager opens the queue, writes on each score and each decision. Pennies a month.

S3 + storage. The original resumes, the stripped text, and the mirrored rubric. A few hundred KB to a few MB at small-business volume. Effectively free.

SES. Inbound for the apply lane: $0.10 per thousand received messages. Outbound for the interview invites and declines a human chooses to send: $0.10 per thousand. Both are negligible at this scale.

Textract (only on scanned PDFs). Per-page pricing, and it only fires when a resume is a scanned image rather than real text. Most resumes are already text, so this slice stays small. A handful of cents at 200 applications; a dollar or two at 1,000 if many arrive as scans.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the careers upload and the manager's queue and buttons.
- **NAT Gateway.** Nothing is in a private network. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. Nothing runs between applications.
- **A search index.** The rubric is a short list of must-haves, read straight; no embeddings, no vector store needed.
- **A second model on the decision.** The label is plain Python counting against your pass marks. The one model call is the read; the deciding is a human's.

How the cost scales

The bill grows with applications, because each application is read once. Double the applications, roughly double the Bedrock slice; the rest barely moves. So 2,000 applications a month lands around $20, and 5,000 around $45. Those are busy-hiring numbers for a small business, and even then the cost is a rounding error next to the time saved and the good candidates that don't get missed. If you ever ran far higher volumes, you'd batch the reads — but that's a tuning step, not a redesign.

Set an AWS Budgets alarm at $15/month (raise it if you hire at higher volume) so anything unusual pages you before the bill matters. At normal small-business hiring volume the screener 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 SES inbound rule set, and the Bedrock model IDs.

PART 7 OF 7

MAY 21, 2026 PART 7 OF 7 · APPLICANT SCREENER SERIES ~8 MIN READ

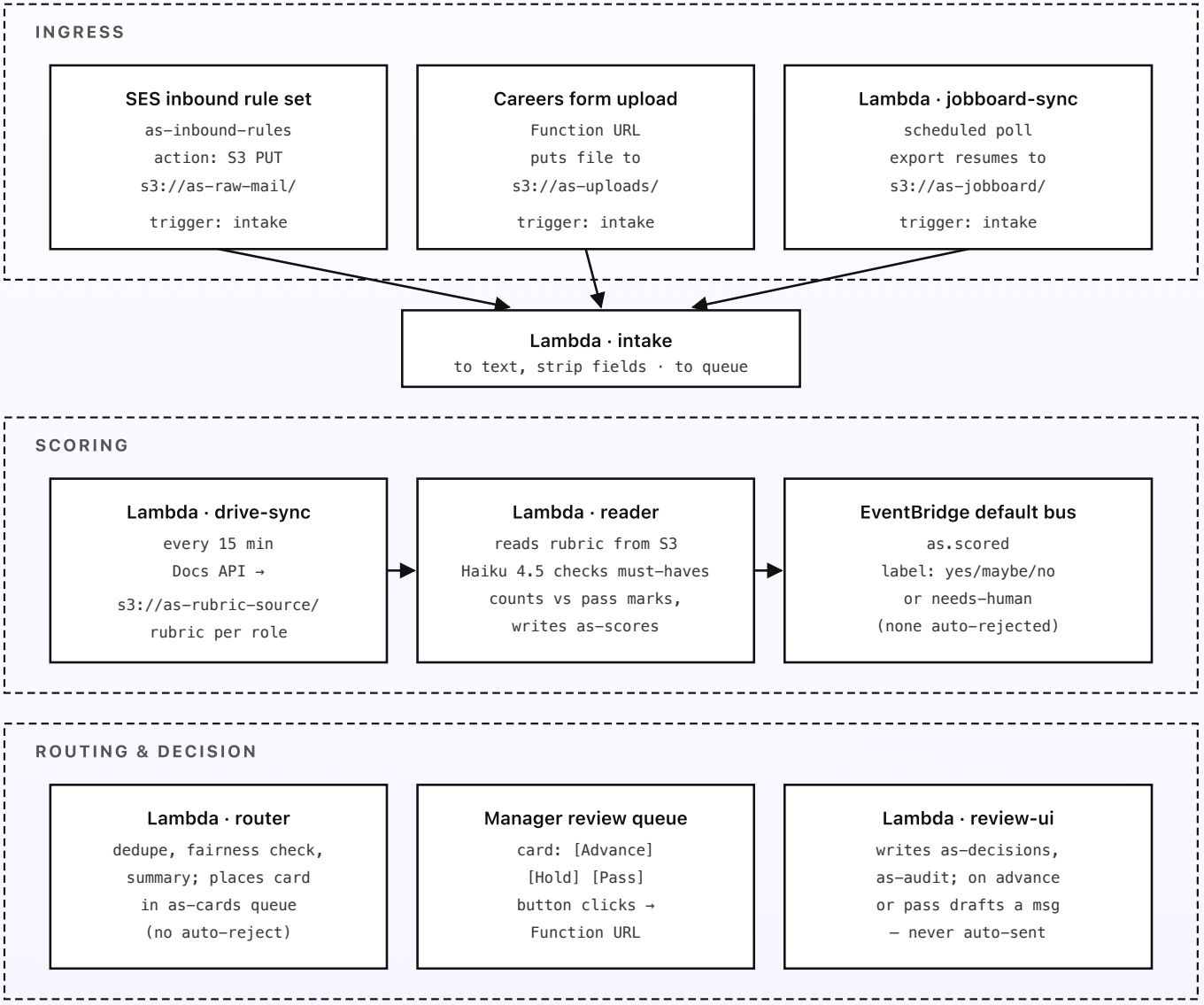
Engineering reference: the applicant screener architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, the EventBridge config, the DynamoDB schemas, and the fairness and human-in-the-loop controls expressed as concrete design decisions. 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 cross-Region inference, and Textract are all available there. A second region for multi-region resilience isn't worth the extra setup at small-business volume — the failure mode for an SMB hiring round is a delayed read, not a regional outage. One AWS account dedicated to the screener (separate from your other workloads) keeps the IAM blast radius small, isolates candidate data, and lets a single AWS Budgets alarm cover the whole system. Bucket policies block public access; candidate PII lives only in this account.

Topology



The screener routes and explains — a human decides, and every decision is logged to as-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the queue), scoring (the per-application read emitting a labelled event), routing and decision (the card is placed and the human acts). Every Lambda is event- or schedule-driven; nothing is synchronous-chained, and nothing is auto-rejected.

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.

- **`intake`** — triggered by S3 PUT on `as-raw-mail`, `as-uploads`, and `as-jobboard`. Parses MIME or reads the uploaded file, extracts the resume, and converts to text: real-text PDFs via `pdfminer.six`, DOCX via `python-docx`, and scanned PDFs/images via Textract `DetectDocumentText`. Then applies the field-stripping rules from the rubric (regex + a small allow/deny pass) to remove name, age/DOB, gender, photo, home address, and school, writing the stripped text and the role to the application queue while keeping the untouched original in `s3://as-originals/`. Memory: 512 MB. Timeout: 60 s.
- **`drive-sync`** — EventBridge Scheduler target, every 15 minutes. Uses the Google Docs/Drive API (service-account credentials in Secrets Manager under `as/drive/sa`) to export each role's rubric doc as plain text to `s3://as-rubric-source/<role>.txt` only if changed since the last sync. The rubric parses into must-haves (each with a `required` flag), nice-to-haves, the strip-list, and the yes/maybe pass marks. Memory: 256 MB. Timeout: 30 s.

- **`reader`** — triggered per application off the queue (SQS with a DLQ). Loads the role rubric from `s3://as-rubric-source/` and calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) once: a tool-use/JSON-mode prompt that returns, per must-have, `met` (bool) and `evidence` (the quoted resume line) or a missing flag, plus an `unreadable` flag. Plain Python tallies `met` against the pass marks — a missing `required` must-have caps the label at `no` — producing `yes`, `maybe`, `no`, or `needs_human`. Writes the result to `as-scores` and emits `as.scored`. *The model never returns the label; only the per-must-have facts. The label is computed.* Memory: 512 MB. Timeout: 60 s.
- **`router`** — EventBridge rule on `as.scored`. Dedupes the applicant against existing `as-cards` for the same role (contact + content hash, not the stripped name); runs the fairness check (confirms strip ran; scans `evidence` for leaked personal tokens and routes to `needs_human` if found); builds the short summary from the structured result; and writes a card to `as-cards` with a queue position (yes → top, maybe → mid, no → parked-reviewable). Never sends anything outbound. Memory: 256 MB. Timeout: 30 s.
- **`review-ui`** — Lambda Function URL, the hiring manager's queue and the action handler. `AuthType: AWS_IAM` behind an IAM-authenticated session (or a signed cookie issued after SSO); not public. GET renders the queue and cards; POST handles advance/hold/pass. Writes `as-decisions` and an `as-audit` row (user, ts, label-before, action, reason). On advance or pass, drafts a message via Haiku 4.5 and returns it to the UI for the human to edit and send — the send is a separate, explicit human action through `notify`. Memory: 256 MB. Timeout: 15 s.

- `notify` — invoked only when a human presses send on a reviewed draft. Sends the interview invite or the courteous decline via SES `SendEmail` from the verified sender. There is no code path that sends candidate-facing email without a human send action. Memory: 256 MB. Timeout: 15 s.
- `jobboard-sync` — EventBridge Scheduler target, hourly (or per the board's export cadence). Pulls exported applications from the configured job board's API/SFTP into `s3://as-jobboard/`. Flags any file it can't parse cleanly for human review rather than guessing. Memory: 256 MB. Timeout: 60 s.

Storage

- **DynamoDB** · `as-scores` — one row per scored application. PK `application_id`; attributes: `role`, `label`, `met_count`, `results` (per-must-have met/missing + evidence), `rubric_version`. On-demand.
- **DynamoDB** · `as-cards` — one row per routed card. PK `(role, queue_pos)`; attributes: `application_id`, `label`, `summary`, `state` (queued/held/decided). On-demand.
- **DynamoDB** · `as-decisions` — one row per human action. PK `application_id`; sort key `decided_at`; attributes: `action` (advance/hold/pass), `by_user`, `reason` (if pass), `label_before`. On-demand.
- **DynamoDB** · `as-audit` — one row per write action of any kind, including overrides. PK `(application_id, ts)`; attributes: `action`, `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term fairness audit trail.
- **S3** · `as-originals` — untouched original resumes. Versioning enabled. Lifecycle to Glacier at 90 days; expiry per your data-retention policy (default:

deleted after the role closes + the retention window).

- **S3 · `as-rubric-source`** — mirrored rubric text per role. Versioning enabled, so a bad rubric edit can be rolled back in one click.
- **S3 · `as-raw-mail`, `as-uploads`, `as-jobboard`** — raw intake by lane. Lifecycle to Glacier at 30 days; expiry per retention policy. All buckets block public access.

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: `reader` for the per-application must-have check, and `review-ui` for drafting the (human-reviewed) invite/decline messages. The heavier `anthropic.claude-sonnet-4-6-20250930-v1:0` is wired but off by default; switch the reader to it only if a role's must-haves need deeper reasoning than Haiku gives, accepting the higher per-read cost.
- **Embeddings.** Not used. The rubric is a short structured list; the model reads the resume directly against it. No Knowledge Base, no S3 Vectors. (Titan Text Embeddings V2, 1024-dim, would be the choice if a future feature needed semantic resume search.)
- **Guardrails.** The reader prompt forbids considering anything outside the listed must-haves and forbids inferring protected attributes; combined with field-stripping upstream and the fairness check downstream, the model has neither the inputs nor the instruction to score on protected grounds.

EventBridge and scheduling

- **`as-drive-sync`** — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- **`as-jobboard-sync`** — `rate(1 hour)`. Target: `jobboard-sync` Lambda.
- **`as.scored` rule** — EventBridge default bus rule matching `detail-type: as.scored`. Target: `router` Lambda.
- **Queue** — SQS standard queue feeding `reader`, with a dead-letter queue after 3 receives so a poison resume parks instead of looping. TZ for any human-facing timestamps is `Asia/Singapore` (`TZ_NAME`).

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `apply.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `as-inbound-rules`: one rule with recipient `apply@your-company.com` → spam scan → S3 PUT to `s3://as-raw-mail/<message-id>` → stop. The S3 PUT triggers `intake`.
- SES outbound (used only by `notify`, only on a human send): verify a sender identity at `hiring@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:

- **intake role:** `s3:GetObject` on the three intake buckets; `s3:PutObject` on `as-originals`; `textract:DetectDocumentText`; `sqs:SendMessage` to the reader

queue. *No* `bedrock:*`.

- **reader role:** `sqs:ReceiveMessage` / `DeleteMessage` on the queue; `s3:GetObject` on `as-rubric-source`; `bedrock:InvokeModel` on the Haiku ARN; `dynamodb:PutItem` on `as-scores`; `events:PutEvents` on the default bus.
- **router role:** `dynamodb:Query` / `PutItem` on `as-cards`; `dynamodb:GetItem` on `as-scores`. *No* SES, no Bedrock — the router never sends and never re-scores.
- **review-ui role:** `dynamodb:Query` on `as-cards` / `as-scores`; `dynamodb:PutItem` on `as-decisions` and `as-audit`; `bedrock:InvokeModel` on the Haiku ARN (drafts only); `lambda:InvokeFunction` on `notify`. Function URL is `AuthType: AWS_IAM`.
- **notify role:** `ses:SendEmail` from the verified sender only. Invocable only by `review-ui` on an explicit human send.
- **drive-sync / jobboard-sync roles:** `secretsmanager:GetSecretValue` on the relevant secret; `s3:PutObject` on the rubric / jobboard buckets; outbound network to the Google or job-board API host only.

Human-in-the-loop and fairness, as controls

The fairness and human-in-the-loop guarantees aren't prose — they're enforced by where permissions and code paths exist. There is *no* Lambda with permission to send a candidate-facing rejection, and *no* code path that marks an applicant rejected without an `as-decisions` row authored by a user. The label is computed from counts, not emitted by the model. Protected attributes are stripped before

the model sees the text, the model is instructed to ignore anything off-rubric, and a post-score fairness check parks anything where personal data leaked. Overrides (a human advancing a screener `no`, or passing a `yes`) are first-class audit rows; a recurring override pattern is the signal to fix the rubric, surfaced in a monthly review. Every step that touches a candidate is in `as-audit`, retained for the long term, so any hiring round can be reconstructed and shown to be decided on job-related grounds by named people.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON (no resume text or PII in logs — only IDs). Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a metric for alerting.
- **Alarms:** reader DLQ depth > 0 (a resume failed to score); review-ui 5xx rate > 1% in 24h; fairness-check leak rate > 0 over a day (the stripper may need tightening).
- **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 `as-cost-alarm` subscribed to the admin's email.

Config and secrets

Google service-account credentials for the Docs/Drive API live in Secrets Manager under `as/drive/sa`; job-board API credentials under `as/jobboard/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, the data-retention window, and the admin contact live in Parameter

Store under `/as/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. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for `as-rubric-source` and `as-originals` so a bad rubric edit or a wrong delete can be rolled back, and keep candidate buckets locked to block-public-access at the account level. Total deployable surface: around seven Lambdas, four DDB tables, six S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SQS queue with a DLQ, 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.