7-PART SERIES · FREE COMPANION

Waitlist manager

A serverless system that turns no-shows and cancellations into filled slots. For a business that books slots — a salon, a clinic, a restaurant — it keeps a waitlist; when a slot frees up, it offers that slot to the next suitable person on the list with a short claim window, rolls to the next if they don't take it, and confirms the booking. Fair order; no double-booking. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Waitlist manager

PART 1 OF 7

JUNE 5, 2026 PART 1 OF 7 · WAITLIST MANAGER SERIES ~5 MIN READ

A waitlist manager on AWS for a few dollars a month

A salon, a clinic, and a restaurant all share one quiet leak: a slot that nobody fills. A client cancels the 2pm cut at 1:40. A patient doesn't show for the 10am. A four-top calls off an hour before. The chair sits empty, the room sits empty, the table sits empty — and there were people who would have jumped at that time, if only someone had phoned them. This post walks through the design of a small system that keeps a waitlist, notices the moment a slot frees up, offers it to the next suitable person with a short claim window, and confirms the booking — without anyone on staff working the phones.

KEY TAKEAWAYS

- Three sources for waitlist entries: a Drive list, an inbox forwarding lane, and a web form.
- Every freed slot ends in one of four moves: nobody fits, make an offer, roll on, or hand back to staff.
- Fair order by join time, but only among entries that actually fit the slot. A priority flag can move someone up.
- One offer is live at a time and a single conditional write books the slot — so two people never get the same one.
- 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.

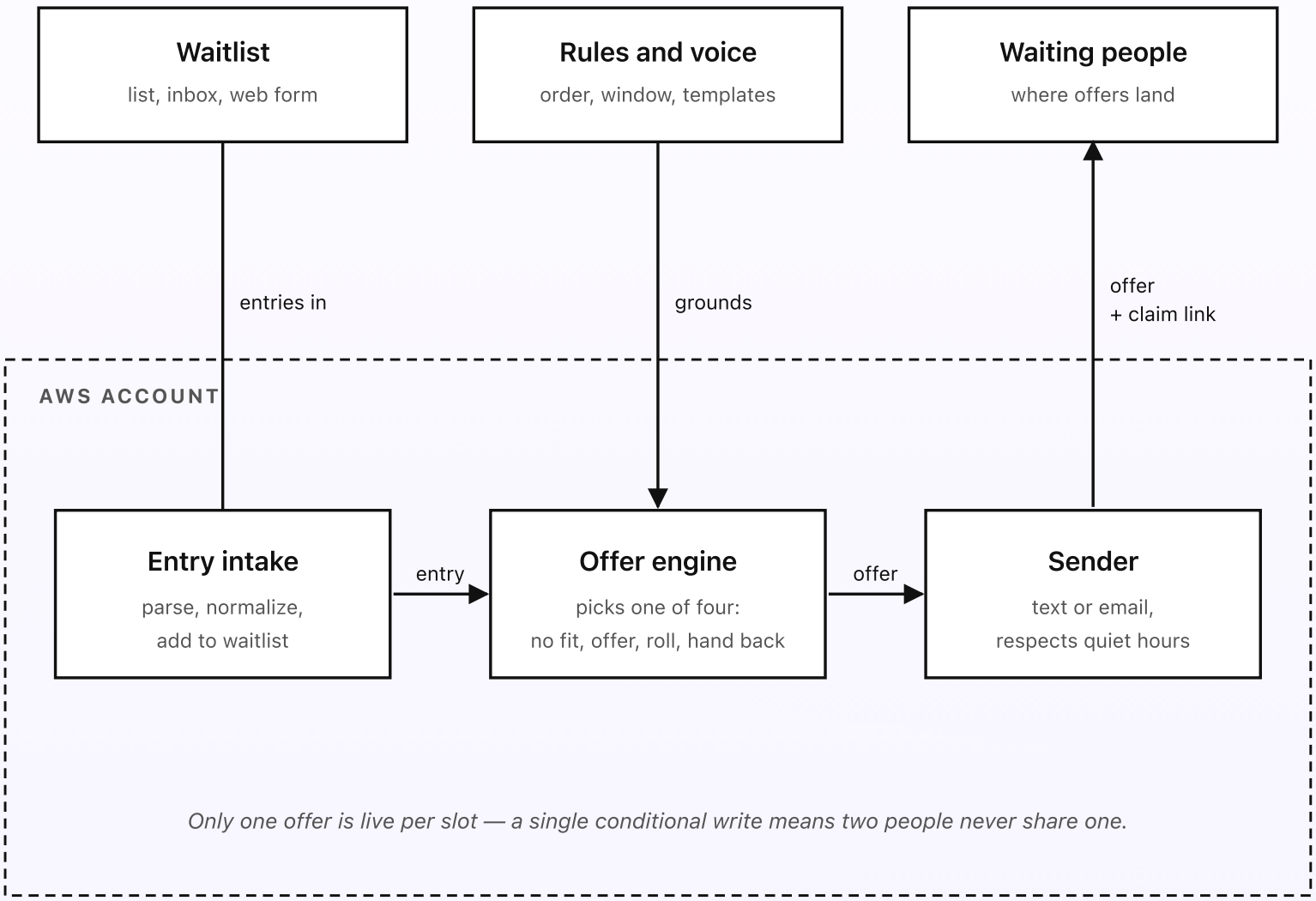

Waitlist
list, inbox, web form
Rules and voice
order, window, templates
Waiting people
where offers land
entries in
grounds
offer
+ claim link
AWS ACCOUNT
Entry intake
parse, normalize,
add to waitlist
entry
Offer engine
picks one of four:
no fit, offer, roll, hand back
offer
Sender
text or email,
respects quiet hours
Only one offer is live per slot — a single conditional write means two people never share one.

Fig 1. Three sources outside, three pieces inside AWS. Entries flow in from a Drive list, an inbox forwarding lane, and a web form. The Offer engine wakes when a slot frees and picks one of four moves. The Sender gets the offer to the right person with a short claim window.

What you set up once (the outside)

- **The waitlist.** A Google Sheet in a Drive folder, one row per person waiting: name, contact (mobile and/or email), the service or table they want, party size, the earliest and latest dates that work for them, any staff preference, a priority flag, and the time they joined. You can fill it in by hand, but most entries arrive through the two other lanes covered in Part 2 — an inbox-forwarding lane (forward a booking request and the system proposes a row for one-tap approval) and a web form (a "join the waitlist" page that drops a row straight in).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc holds the order and the checks. The order is first-come, first-served by join time unless a priority flag moves someone up. The checks decide who actually fits a freed slot — right service, right party size, the date inside their window, a matching staff preference if they set one. The doc also holds the claim-window length (default 10 minutes), the quiet hours, and how many people to try before handing the slot back to staff. The *voice* doc holds one offer message template per channel — what the text or email actually says.
- **Waiting people.** The customers on the list. Each one has a mobile number (so the offer is a text) or, if no number is set, an email address. Offers land with the freed slot's date and time, the service, a one-tap claim link, and a countdown that says how long they have to take it before it rolls to the next person.

What runs when a slot frees (the inside)

- **The entry intake.** Three sources feed the waitlist. The Drive sheet itself is the canonical store. New entries can also arrive via the inbox forwarding lane (forward a booking request to `waitlist@your-business.com`; the system uses Textract to read any attachment and Bedrock Haiku 4.5 to pull out name, service, party size, and dates, then drops a one-tap approval card for a staffer to confirm before the row is added) and the web form (a simple hosted page that writes a clean row directly).
- **The offer engine.** Wakes the moment a slot frees — a cancellation, a no-show marked by staff, or a fresh opening. It reads the waitlist, keeps only the entries that fit the slot, sorts them in the fair order from the rules doc, and picks one of four moves. *Nobody fits*: the list has no eligible entry — leave the slot open and tell staff. *Make an offer*: send the slot to the top eligible person and start a short claim window. *Roll on*: the window ended unclaimed — send the same slot to the next eligible person with a fresh window. *Hand back to staff*: the list is exhausted or the cut-off is reached — return the slot with a note. The engine calls no model on this path — the move logic is plain Python.
- **The sender.** Reads the voice doc, formats the offer for the chosen channel, and sends it. Texts go through SNS; email goes through SES outbound. Both honor quiet hours so a 6am cancellation doesn't wake the whole list. The claim link points at a Lambda Function URL; the first valid claim flips the slot to booked with a single conditional write, sends a confirmation, and cancels the window timer. Every offer, claim, and roll writes a row in DynamoDB so the trail is clean. A monthly summary writes an owner-ready paragraph: slots freed, slots filled, average time-to-fill, and how much empty-chair revenue was recovered.

In plain words

It's a Saturday salon. The 2pm colour cancels at 1:35. Three people are on the waitlist for a colour this week. The engine keeps the two whose date window includes today, drops the one who asked for a specific stylist who isn't in, and sorts the remaining two by who joined first. It texts the first: "A colour opened up today at 2:00pm with Jess. Tap to claim — you have 10 minutes before we offer it to the next person." She's at lunch and doesn't look. At 1:45 the window ends; the offer rolls to the second person, who claims it in two minutes. The slot flips to booked, she gets "You're confirmed for 2:00pm today", the first person's link goes dead, and the front desk sees the chair is filled. Nobody phoned anyone.

The cost of running this is about $2 a month at SMB volume. The cost of *not* running it is every empty chair, room, or table that someone on a list would have happily taken — revenue that walks out the door because nobody had time to work the phones in the ten minutes that mattered.

DESIGN RULES THAT SHAPED EVERY DECISION

- Only one offer is live per slot. The next person is never texted until the current window ends or the slot is declined.
- Four moves, always. Nobody fits, make an offer, roll on, hand back to staff. There is no fifth.
- Fair order is written down. Join time by default; a priority flag can lift someone; the same inputs always give the same order.
- Eligibility is checked before anyone is offered — right service, party size, date window, staff preference. No offer that can't actually be honored.
- The claim is a single conditional write. The first valid click wins; every later click is told the slot is taken. No double-booking.
- Every offer, claim, roll, and time-out is logged. Audit a busy Saturday later and you can see exactly what happened.

Why this shape

Most businesses "run a waitlist" on a sticky note by the till or a note in someone's head. It works right up until the slot actually opens — at which point the person who could fill it is busy with a customer, the list is in a drawer, and by the time anyone gets to the phone the moment has passed. The few tools that automate this tend to blast the whole list at once, which creates the opposite problem: ten people race for one slot, nine get a "sorry, taken", and you've trained your best customers to ignore your texts.

The setup above keeps the list somewhere staff already edit, but adds a small system that *watches* for a freed slot and acts the instant one appears. It offers to one person at a time, in an order that's fair and written down. It gives a real countdown so nobody sits on an offer forever. It rolls on cleanly when somebody's not looking. And the booking is locked by a single write, so two people are never sent to the same chair. The engine is invisible most of the day; it only does something in the few minutes after a slot opens — which is exactly when it matters.

The next four posts walk through each piece in turn: how a waitlist entry gets added, how a freed slot gets noticed, how an offer reaches the next person, and how a slot gets claimed or rolls on. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 5, 2026 PART 2 OF 7 · WAITLIST MANAGER SERIES ~4 MIN READ

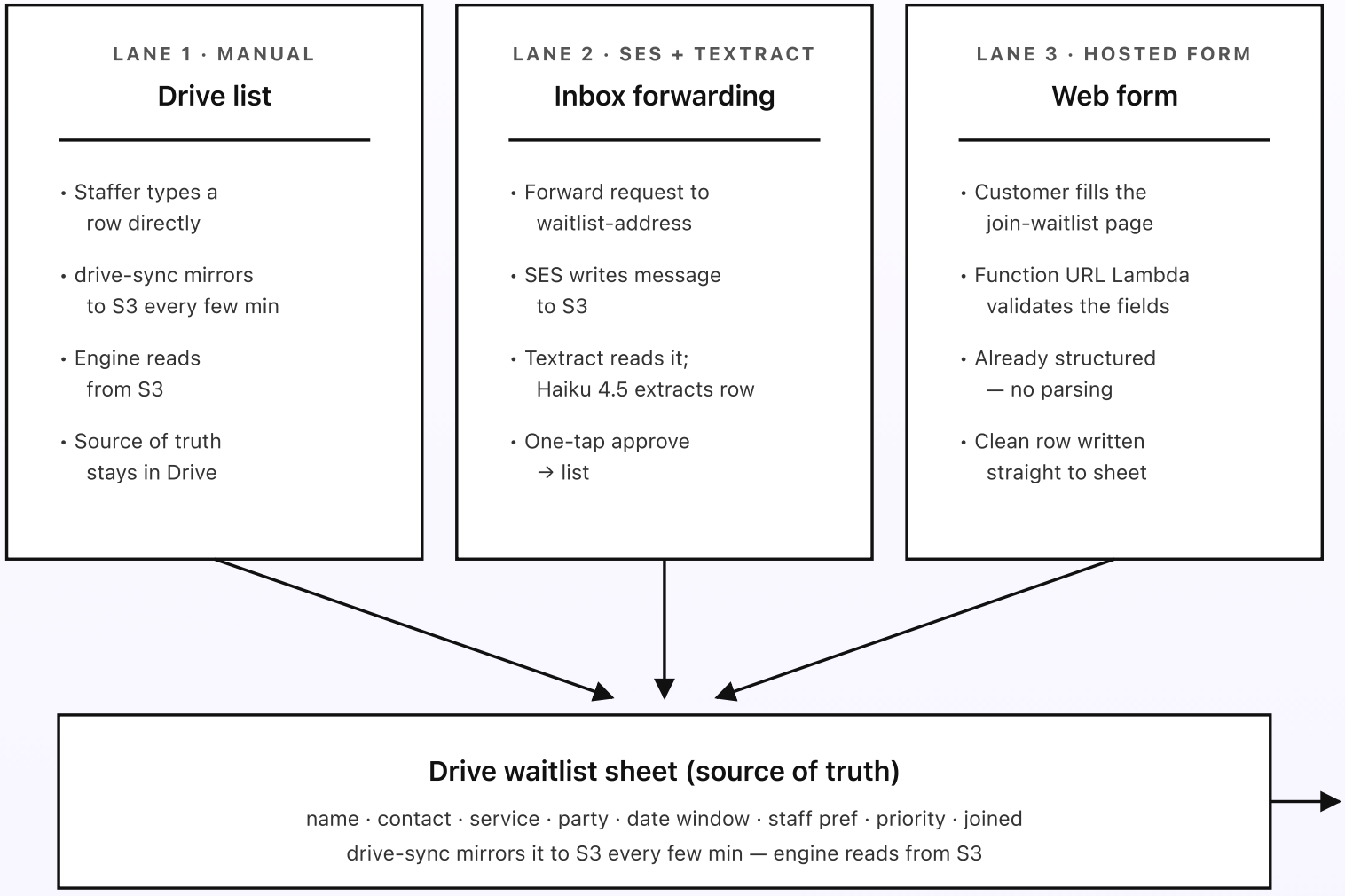
How a waitlist entry gets added

The engine can only offer a slot to someone who's actually on the list. So the first job is making sure the list reflects everyone who wants an earlier time. There are three ways an entry gets in: a staffer types a row in the Drive list, somebody forwards a booking request to a dedicated address, or a customer fills in a short web form. The first one is obvious. The other two exist because in real life nobody stops mid-shift to type a row for the person standing at the desk asking to be called if anything opens up.

KEY TAKEAWAYS

- Three intake lanes feed one waitlist: the Drive list, an inbox-forwarding lane, and a web form.
- Forwarded requests are parsed by Textract; Bedrock Haiku 4.5 reads the text and proposes a row.
- Every parsed row goes to a staffer for one-tap approval before it lands in the list.
- The web form writes a clean row directly — it's already structured, so there's nothing to parse.
- The Drive list stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one list



The Drive list stays the source of truth — the other lanes are conveniences that add rows to it.

Fig 2. Three lanes converge on one Drive list. The list is the source of truth; the inbox lane and the web form are conveniences that add rows. The drive-sync Lambda mirrors the sheet to S3 so the engine can read it without hitting Drive every time a slot frees.

Lane 1: the Drive list itself

The simplest lane. Open the waitlist sheet in Drive, add a row, save. The columns are short: name, contact, the service or table wanted, party size, the earliest and latest dates that work, any staff preference, a priority flag, and the time they joined. A small Lambda — `drive-sync` — runs every few minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://wl-waitlist-source/waitlist.csv` if the sheet has changed since the last sync. The engine reads from S3, not Drive directly. That keeps Drive calls predictable and gives you S3 versioning for free, so a bad bulk-edit can be rolled back in one click.

This lane covers the front-desk case: a customer rings or walks in, asks to be called if anything opens, and a staffer has thirty seconds to type it in. It's also how staff fix a row — change someone's date window, bump a priority flag, or take a person off the list once they've been booked.

Lane 2: inbox forwarding (the lane for messy requests)

Set up a dedicated inbound address — something like `waitlist@your-business.com` — via Amazon SES. When a request comes in by email (a customer writes "can you call me if a Saturday colour opens up?", or a booking-site notification lands), a staffer forwards it to that address and the system takes it from there. SES writes the raw message to `s3://wl-raw-mime/`. The S3 write triggers a parser Lambda. The Lambda walks the message to any attachment,

runs Amazon Textract on it if it's a PDF or image (Textract reads PDF, PNG, JPEG, and TIFF natively), and otherwise just reads the email body text.

Then a Bedrock Haiku 4.5 call reads the text and emits a structured row: name, contact, service, party size, the earliest and latest dates that work, and any staff preference mentioned. The model prompt is short: "Extract a waitlist entry. Return JSON only. Mark each field with a confidence score. Do not invent a date that isn't in the text." The output goes to a small interactive message on the staffer's phone: the proposed row, the confidence per field, and three buttons — *approve*, *edit*, *discard*. On *approve*, a Lambda writes the row to the Drive list via the Sheets API. On *edit*, the staffer gets a fillable form pre-populated with the proposal. On *discard*, the message is logged and the raw email moved to a discarded prefix in S3 for audit.

The reason every parsed row goes to a human first is simple: an entry the model misread is worse than one that never made it onto the list. A wrong phone number or a wrong date means an offer that can't be honored — and the whole point of the system is that every offer is real.

Lane 3: the web form

The lowest-effort lane for the customer, and the one that scales without any staff time at all. A simple hosted page — "Join the waitlist" — collects the same fields: name, contact, service, party size, the date range that works, and an optional staff preference. The page posts to a Lambda Function URL. The Lambda validates the fields (a real phone number or email, a sensible date range, a service that exists), drops obvious spam, and writes a clean row straight to the Drive list via the Sheets

API. Because the form already collects structured fields, there's nothing to parse and no approval step — the row is good the moment it's submitted.

Put the form link on your booking confirmation page ("Want an earlier time? Join the waitlist"), in your booking emails, and on your social profiles. It turns "we're fully booked" from a dead end into a captured lead.

Why the list stays the source of truth

Three lanes in, but only one place the engine actually reads. That's a deliberate constraint. If two lanes both wrote directly to the engine's state, every "why did this person get offered the slot?" question would mean checking three places. Funneling everything through the Drive list means there is exactly one row per entry, and any staffer can read or edit any of it without learning a new tool. The convenience lanes are first-class for getting entries in, but they always pass through the list on the way.

Next post: how the engine notices the moment a slot frees up, works out who on the list actually fits it, and picks one of four moves.

PART 3 OF 7

JUNE 5, 2026 PART 3 OF 7 · WAITLIST MANAGER SERIES ~5 MIN READ

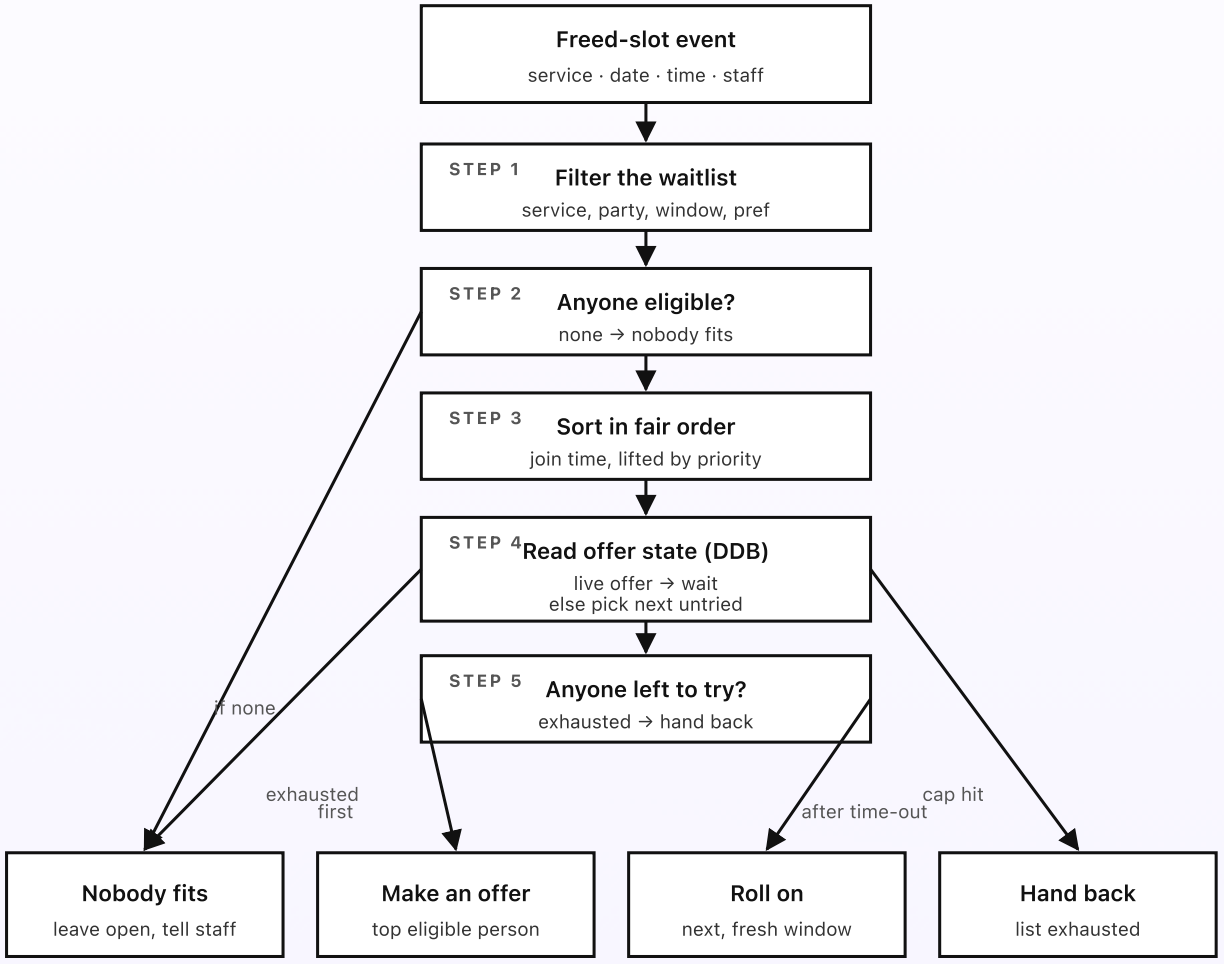
How a freed slot gets noticed

A slot frees up three ways: a customer cancels, staff mark a no-show, or a fresh opening appears on the calendar. Any of those sends an event to the offer engine. The engine reads the waitlist, looks at the freed slot, keeps only the entries that actually fit it, sorts what's left in the fair order, and decides whether to offer the slot to someone or hand it back. The whole decision is plain Python. No model. No vector search. Every rule lives in the rules doc, where staff can edit it without a deploy.

KEY TAKEAWAYS

- A cancellation, a no-show, or a new opening sends a freed-slot event to the engine.
- Eligibility filters the list first — right service, right party size, date inside the window, matching staff preference.
- Survivors are sorted in fair order: join time by default, lifted by a priority flag.
- Four moves per slot: nobody fits, make an offer, roll on, or hand back to staff.
- The engine never calls a model. The decision is entirely deterministic.

The decision flow, per freed slot



The rules doc holds every rule — change the order or window and the next freed slot uses it.

Fig 3. The engine's decision tree, per freed slot. Five steps decide which of four moves applies. The rules doc holds the eligibility checks and the order; the engine only enforces them.

What triggers the engine: three ways a slot frees

The engine doesn't poll all day — it wakes on an event. Three things send one. A *cancellation*: the customer cancels through your booking tool, which fires a webhook to a Function URL, or a staffer marks the slot cancelled in the calendar and a small sync Lambda notices. A *no-show*: staff tap "no-show" at the front desk a few minutes after the appointment time, which frees the rest of that slot. A *new opening*: someone adds availability — a stylist picks up an extra hour, a table is reconfigured — and that new gap counts as a freed slot too. Each one becomes a `wl.slot_freed` event carrying the slot's service, date, time, party capacity, and staff member.

Keeping the engine event-driven matters for both cost and speed. There's nothing running between slots, so the bill is near zero on a quiet day; and when a slot does free, the first offer goes out in seconds, not on the next poll.

Eligibility: who actually fits this slot

Before anyone is sorted, the list is filtered down to entries that can really take the slot. The rules doc spells out the checks in plain prose, and the engine applies them in order. *Service*: a colour slot only fits people waiting for a colour. *Party size*: a two-top doesn't fit a party of four; a four-top can fit a two if the rules doc allows down-fitting, or not if it doesn't. *Date window*: the slot's date has to fall

between the entry's earliest and latest dates. *Staff preference*: if someone asked specifically for Jess, they're only eligible for Jess's slots. An entry that fails any check is dropped for this slot — it stays on the list for the next one.

This filter is the difference between a useful system and a noisy one. Offering a Tuesday slot to someone who only wants Saturdays, or a kids' cut to someone waiting for a colour, trains people to ignore the texts. Filtering first means every offer that goes out is one the person could actually say yes to.

Fair order: written down, the same every time

The survivors are sorted by the order in the rules doc. The default is first-come, first-served by the `joined` timestamp — the person who's been waiting longest gets first refusal. A `priority` flag can lift an entry above plain join order: a regular who got bumped by a previous cancellation, a VIP, or someone staff have promised. The doc names exactly how priority interacts with join time (for example, "priority entries first, then by join time within each tier"), so two staffers reading it get the same answer.

The point of writing the order down is fairness you can defend. When a customer asks "why did she get the slot and I didn't?", the answer is in the doc and the audit log, not in whoever happened to be at the desk.

Four moves, always

Every freed slot, every time, lands in exactly one of four moves. The names are simple on purpose.

- **Nobody fits.** The filter left no eligible entry. Leave the slot open and post a note to staff so they can fill it by phone or take a walk-in. Most slots in a thin-list business land here, and that's fine — the system just got out of the way.
- **Make an offer.** There's a top eligible person and no offer is live yet. Send them the slot and start a claim window. Write a row to the `wl-offers` DynamoDB table marking the offer as live with its expiry time.
- **Roll on.** The previous person's window ended or they declined, and there's a next eligible person who hasn't been tried. Send the same slot to them with a fresh window. Mark the previous attempt as timed-out or declined in `wl-offers`.
- **Hand back to staff.** The eligible list is exhausted, or the per-slot try cap in the rules doc is reached. Return the slot to staff with a short note: how many people were tried, and that the list is spent. The slot stays open for a walk-in or a phone fill.

State that keeps it deterministic and double-book-proof

The engine reads one DynamoDB table to make the call: `wl-offers`, which records the live offer per slot and the history of who's been tried — `(slot_id, offer_seq, entry_id, status, window_ends_at)`. With that, the move logic is a few dozen lines of Python and zero magic. A given slot, a given list, and a given offer history always produce the same move. And because only one offer per slot is ever marked *live*, the next person is never texted while someone else is still inside their window — which is the rule that, together with the conditional-write claim in Part 5, makes double-booking impossible.

Why this path uses no model

The engine could call a model to “decide” who deserves the slot, or to write a cleverer offer. It doesn’t. Two reasons. First, who gets offered a slot has to be utterly predictable and defensible — if the rules doc says first-come among those who fit, that’s who gets it. A model in that loop introduces variance staff can’t reason about and a customer can’t be told. Second, slots free up at the busiest moments, when speed matters; a deterministic filter-and-sort runs in milliseconds. Bedrock fires elsewhere — on the inbound parsing lane in Part 2 and on the monthly summary in Part 6 — not on the offer path.

Next post: how an offer reaches the next person — channel choice, quiet hours, the claim link and countdown, and the four guardrails on every send.

PART 4 OF 7

JUNE 5, 2026 PART 4 OF 7 · WAITLIST MANAGER SERIES ~5 MIN READ

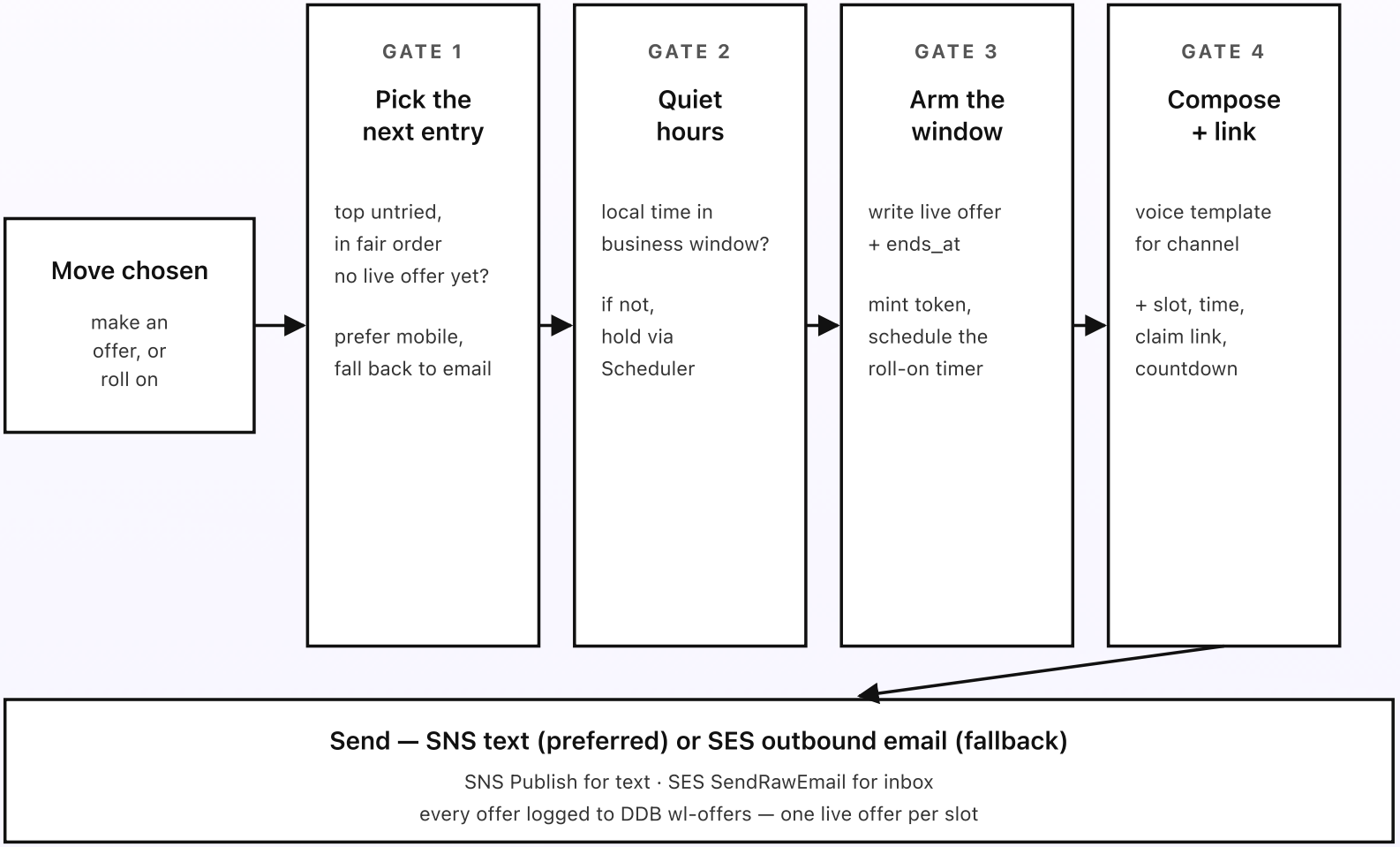
How an offer reaches the next person

The engine picked a move — make an offer, or roll on to the next person. Now the sender has to figure out who exactly to contact, on what channel, at what time of day, and with what link attached. Get any of those wrong and the offer is worse than useless: a 6am text, a dead link, an offer to a number that's been disconnected. Four small guardrails sit between the move and the offer actually landing.

KEY TAKEAWAYS

- Pick the next eligible entry in fair order — only one offer is live per slot at a time.
- Texts are the default; email is the fallback if no mobile number is on the entry.
- Quiet hours hold an offer until business hours — a dawn cancellation doesn't wake the list.
- Every offer ships with the slot, the time, a one-tap claim link, and a live countdown.
- A claim-window timer is armed on send; when it ends, the offer rolls to the next person.

Four guardrails on every offer



Every gate is a deterministic check — no model calls, and never two live offers on one slot.

Fig 4. Four guardrails between the move and the sent offer. Pick the next eligible person. Honor quiet hours. Arm the claim window and timer. Compose with the claim link. Then ship via text or email and log the offer so exactly one is live per slot.

Gate 1: pick the next entry, resolve the channel

Part 3 already filtered and sorted the list, so Gate 1's job is to take the next person who hasn't been tried yet for this slot — the top of the eligible, fair-ordered list minus anyone already offered and timed-out. Before it does anything, it double-checks that no other offer is currently marked *live* for this slot in `wl-offers`. That check is what keeps the "one offer at a time" promise: even if two events arrive close together (a webhook and a timer firing at almost the same moment), only one offer goes out.

Once the sender knows which entry to contact, it looks up the channel. The entry's `contact` field gives a mobile number if one was provided — texts are preferred because a slot opening today needs a reply in minutes, and a text is read in minutes. If there's no mobile, it falls back to the email address. Either way, the claim link works the same.

Gate 2: quiet hours

Slots free up at all hours — a customer cancels their morning appointment at 6:15am, a no-show is marked at 8:50pm. The offer itself is fine to compute then, but sending it then is not: an offer that lands while someone's asleep wastes the claim window on a phone that won't be looked at, and a 6am text is a fast way to get a customer to mute you.

Gate 2 reads the rules doc's quiet-hours setting (for example, no offers between 9pm and 8am, configurable per business). If the current local time is inside the quiet window, the sender holds the offer: it creates a one-off EventBridge Scheduler rule that fires at the next business-hour minute and exits without sending. At that time the same send Lambda runs with the same payload, Gate 2 lets it through, and only then is the claim window armed — so the countdown always starts when the offer is actually delivered, never while it's being held.

Gate 3: arm the claim window and the timer

This is the gate unique to a waitlist. Before the offer goes out, the sender writes the live offer to `wl-offers` with a `window_ends_at` timestamp (now plus the claim-window length from the rules doc, default 10 minutes). It mints a short-lived claim token tied to this exact offer — `(slot_id, offer_seq)` — so the link can only book this one offer and goes dead the instant the offer rolls on. And it schedules an EventBridge Scheduler one-off rule for `window_ends_at` that, if the slot is still open at that time, fires a roll-on event back to the engine.

Arming the window *before* sending matters: if the send fails, there's no orphaned live offer with no message behind it, because the write and the schedule are part of the same handler and a send failure rolls the offer back to pending for a clean retry.

Gate 4: compose with the claim link, then ship

The voice doc has one offer template per channel: a short message with placeholders for the service, the date and time of the freed slot, the staff member,

the claim link, and the countdown (“you have 10 minutes”). The sender fills the placeholders, attaches the tokenized claim link, and ships — via SNS for a text, or wrapped in a small HTML email via SES for the email fallback. The link points at the `claim-handler` Function URL covered in Part 5; tapping it is what books the slot.

A roll-on offer uses the same template — the next person has no idea they’re second in line, and shouldn’t; from their side it’s simply “a slot opened, here it is, you have ten minutes.” Every offer, first or rolled, writes its row to `wl-offers`. The timer, the claim, and the next roll all read that table, so they always agree on which offer is live and which have already passed.

Why the guardrails exist

None of these gates are exotic. They’re the care a thoughtful person at the front desk would take if they were working the list by phone — call the right person next, don’t ring at 6am, give them a real but finite chance to say yes, and make sure the offer you give can actually be honored. Putting them in code as four small sequential gates makes them part of the design, not something you’re trusting a busy staffer to remember on a Saturday.

Next post: what happens when the offer lands — how a single conditional write books the slot for exactly one person, and how an unclaimed window rolls cleanly to the next.

PART 5 OF 7

JUNE 5, 2026 PART 5 OF 7 · WAITLIST MANAGER SERIES ~5 MIN READ

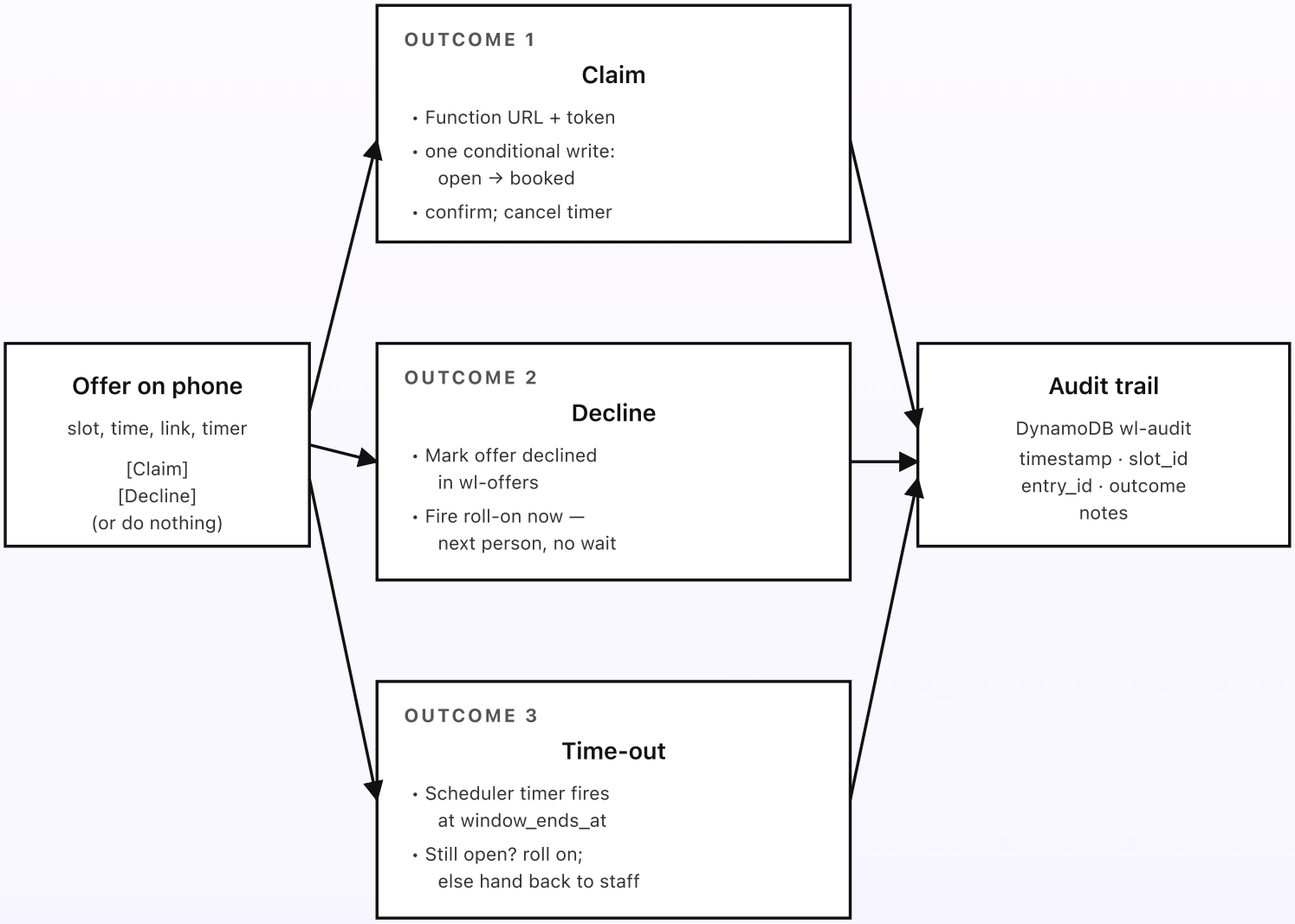
How a slot gets claimed or rolls on

An offer lands in a customer's texts: "A 2:00pm colour opened up today — tap to claim, 10 minutes." There's a claim link. What happens when she taps it? And what happens if she doesn't? This post walks through the three things that can happen to a live offer — claim, decline, time-out — and how a single conditional write makes sure exactly one person ever gets the slot, with the chain state and the audit trail staying in sync.

KEY TAKEAWAYS

- Three outcomes per offer: *claim* (book the slot), *decline* (roll on now), *time-out* (window ends, roll on).
- Claiming is one conditional write to DynamoDB — it succeeds for exactly one person.
- Any later click on a filled slot gets a polite “this slot was just taken” message.
- A time-out fires from an EventBridge Scheduler timer and rolls the offer to the next eligible person.
- The claim link is a Function URL carrying a short-lived token tied to one offer.

Three things can happen to a live offer



One conditional write decides the booking — the first valid claim wins, later clicks are told it's taken.

Fig 5. Three outcomes for a live offer, three different effects. Claim books the slot with one conditional write. Decline rolls on immediately. Time-out rolls on when the window ends. Every outcome writes to the audit trail.

Outcome 1: claim (the one that fills the chair)

The customer taps the claim link. It points at the `claim-handler` Function URL and carries the short-lived token minted in Part 4, which names exactly one offer: `(slot_id, offer_seq)`. The handler does the most important thing in the whole system: **one conditional write**. It updates the slot's row in DynamoDB to `status = booked, booked_by = entry_id` with a condition that the write only succeeds if the slot is still `open` and this offer is still the live one.

If the condition holds, exactly one thing happens, atomically: the slot flips to booked. The handler then confirms to the customer ("You're confirmed for 2:00pm today — see you then"), cancels the window timer so no roll-on fires, marks the token spent, and updates the Drive list via the Sheets API so staff see the chair is filled. If the condition fails — because someone else's claim landed a half-second earlier, or the offer already rolled on — the write simply doesn't apply, and the customer sees a calm page: "Sorry, this slot was just filled. You're still on the waitlist for the next one." No error, no double-booking, no two people sent to the same chair. The whole guarantee rests on that single conditional write: a database either accepts it or rejects it, and only one can win.

Outcome 2: decline (the polite no)

Every offer carries a second link: “Can’t make it.” Tapping it is the customer doing the system a favour — instead of letting the ten-minute window run down, they free the slot to roll on immediately. The decline link hits the same Function URL with a `decline` action. The handler marks this offer `declined` in `wl-offers` and fires a roll-on event straight to the engine, which picks the next eligible person and sends a fresh offer with a full window. The person who declined stays on the list for future slots unless they ask to come off.

Decline matters because it shortens the fill time. On a busy Saturday, the difference between rolling at ten minutes and rolling the instant someone says “not me” can be the difference between filling the 2pm and losing it. It also keeps the list honest — a quick decline is a clearer signal than silence.

Outcome 3: time-out (nobody tapped)

The most common outcome, especially for the first person offered: they’re busy, the phone’s in a bag, the offer sits unread. That’s fine — it’s exactly what the window is for. When `window_ends_at` arrives, the EventBridge Scheduler one-off rule armed in Part 4 fires. It re-reads the slot. If the slot is still `open` (nobody claimed), it marks this offer `timed_out` in `wl-offers` and rolls on: the engine’s “roll on” move sends the same slot to the next eligible person with a brand-new window. If the slot is no longer open — somebody claimed it in the last few seconds — the timer sees that and does nothing, because the claim already cancelled it (and even if a stray timer fires, the slot isn’t open, so the roll-on is a no-op). If the eligible list is exhausted or the per-slot try cap is hit, the slot is handed back to staff with a note instead of rolling further.

There's a deliberate ordering here that keeps things safe: a claim cancels the timer, but the timer also re-checks the slot status, so the two can never both act on the same open slot. The conditional write is the backstop — even in the worst race, only one booking can land.

Every outcome is logged, every fill is explainable

The `wl-audit` table records every claim, decline, and time-out with the slot, the entry, the outcome, and a timestamp. Pull up a busy Saturday later and you can reconstruct exactly what happened to each freed slot: who was offered first, who declined, who timed out, who finally claimed, and when each link fired. If a customer says "I tapped claim and it said taken", the log shows the winning claim landed eight seconds before theirs — the system did exactly the right thing, and you can prove it.

That explainability is the quiet payoff of doing the booking with one conditional write and logging every step. The fast path is simple — tap, book, confirm — and the edge cases (two taps at once, a late tap on a filled slot, a timer that fires just after a claim) all resolve to the same safe answer without anyone having to reason about them in the moment.

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.

PART 6 OF 7

JUNE 5, 2026 PART 6 OF 7 · WAITLIST MANAGER SERIES ~3 MIN READ

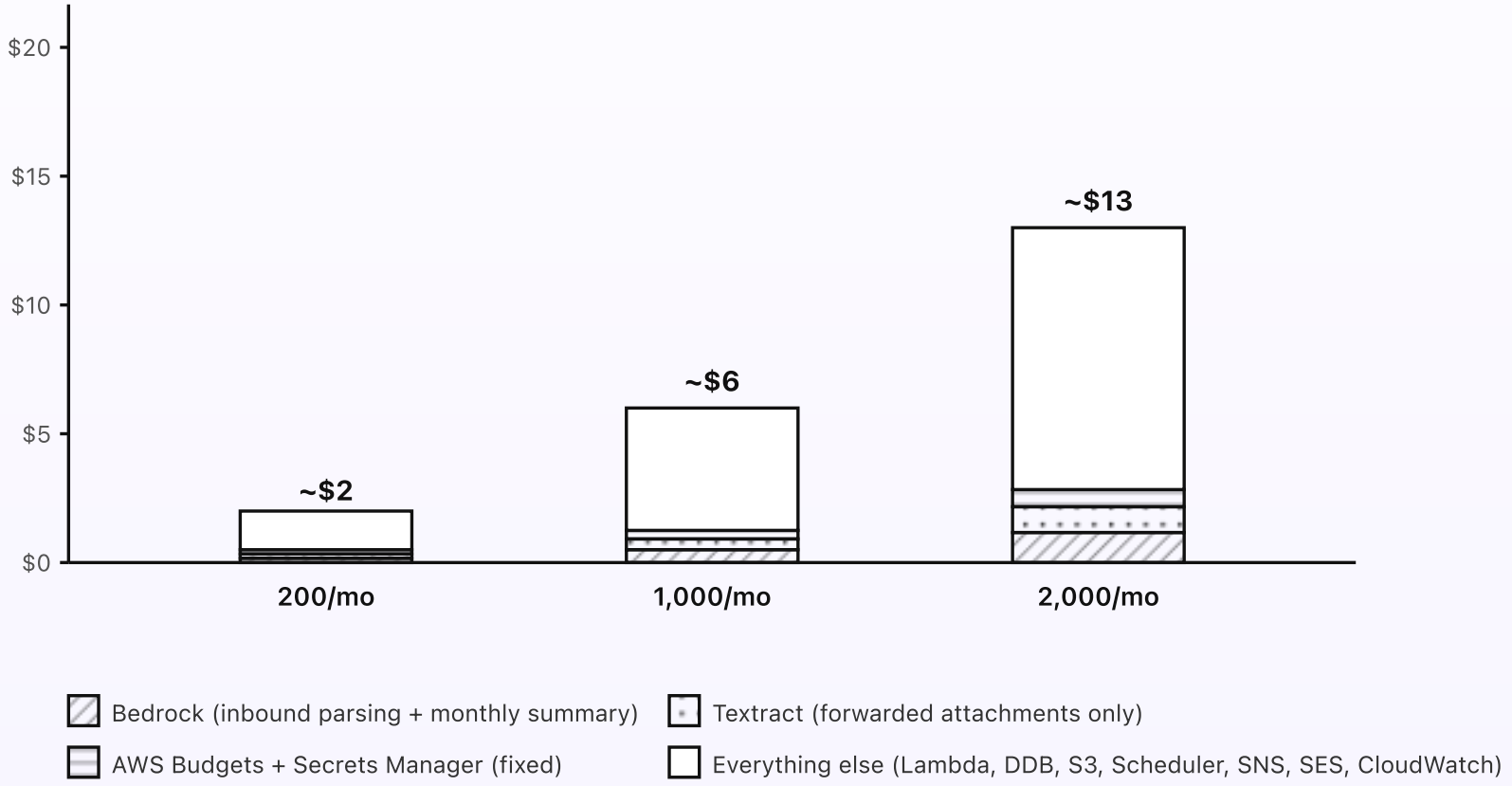
What the waitlist manager costs

The waitlist manager is one of the cheapest systems in this whole series. It does nothing between slots — the engine wakes only when a slot frees up, runs some filtering and sorting, sends a text or two, and waits on a timer. It calls no models on the offer path. Bedrock fires only when somebody forwards a booking request and once a month for the owner summary. At typical SMB volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2/month at typical SMB volume (around 200 waitlist entries a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The offer path costs pennies — no model calls; the bulk is the texts and emails.
- Bedrock fires only on inbound parsing (a few times a month) and the monthly summary.
- At 1,000 entries the bill is around $6. At 2,000 entries it's around $13.

Cost at three volumes



The offer texts are the dominant cost — and even those are a fraction of a cent each.

Fig 6. Monthly cost at three waitlist-entry volumes. Bedrock and Textract are small slivers because they only fire on the inbound parsing lane and the monthly summary. The dominant cost is the everything-else bucket: the offer texts and the per-offer writes.

Where the dollars actually go

SNS texts and SES email (the bulk). Each freed slot sends one offer per person tried — usually one to three texts before the slot fills or rolls out. A text in most regions is a fraction of a cent. At 200 entries a month, with maybe 60 to 100 slots freeing and a couple of offers each, that's a few hundred messages — well under a dollar. Email fallback through SES is $0.10 per thousand, so cheaper still. This is the one line that scales with how busy you are, which is exactly the line you want to grow, because every text is a chance to fill a chair.

Lambda runtime. The engine, the sender, the claim handler, the drive-sync every few minutes, and the inbound parser. Each invocation is short — read a small CSV, filter and sort a list of dozens, do a write or two. Even at 2,000 entries a month the Lambda total lands under a dollar.

DynamoDB on-demand. Three small tables: `wl-offers`, `wl-slots`, `wl-audit`. Writes are offers, claims, and audit rows; reads are the engine's eligibility lookups and the conditional-write claim. Pennies a month at any of these volumes.

S3 + storage. The mirrored waitlist CSV plus the archived raw email from any forwarded requests. A few hundred KB total at SMB volume. Effectively free.

EventBridge Scheduler. The claim-window one-off timers (one per offer) plus the daily housekeeping tick and the drive-sync rule. A handful of invocations per freed slot. Pennies.

Bedrock (only when something fires it). The offer path uses no Bedrock. The inbound parsing lane fires Haiku 4.5 once per forwarded request: a few thousand input tokens and a few hundred output tokens, so a fraction of a cent per parse. At

a few forwarded requests a month, Bedrock costs cents. The monthly summary is one larger call — a paragraph on slots freed, slots filled, and revenue recovered — a couple of cents.

Textract (only on forwarded attachments). Per-page pricing; most forwarded requests are plain email text with no attachment, so Textract fires rarely. A few cents per parse when it does. At SMB volume, a few cents to a dollar.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the claim link and the web form.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The engine does nothing between slots.
- **A Knowledge Base.** The waitlist is structured rows, not free text — deterministic filter-and-sort beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the offer path.** The offer decision is plain Python. Bedrock fires only on the inbound parsing lane and the monthly summary.

How the cost scales

The messaging line grows with how many slots free up and how many people each offer is tried on, so it tracks how busy the business is — which is the right thing to scale with. Lambda and DynamoDB grow roughly linearly too. Bedrock and

Textract are uncorrelated with volume — they only fire when somebody forwards a request or it's the first of the month. So the bill at 5,000 entries a month is around $30; at 10,000 it's around $60. Past those volumes you're a busy multi-location operation and the texting cost dominates — at which point negotiating a messaging rate matters more than any AWS tuning.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The waitlist manager'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 conditional-write claim, and EventBridge Scheduler config.

PART 7 OF 7

JUNE 5, 2026 PART 7 OF 7 · WAITLIST MANAGER SERIES ~8 MIN READ

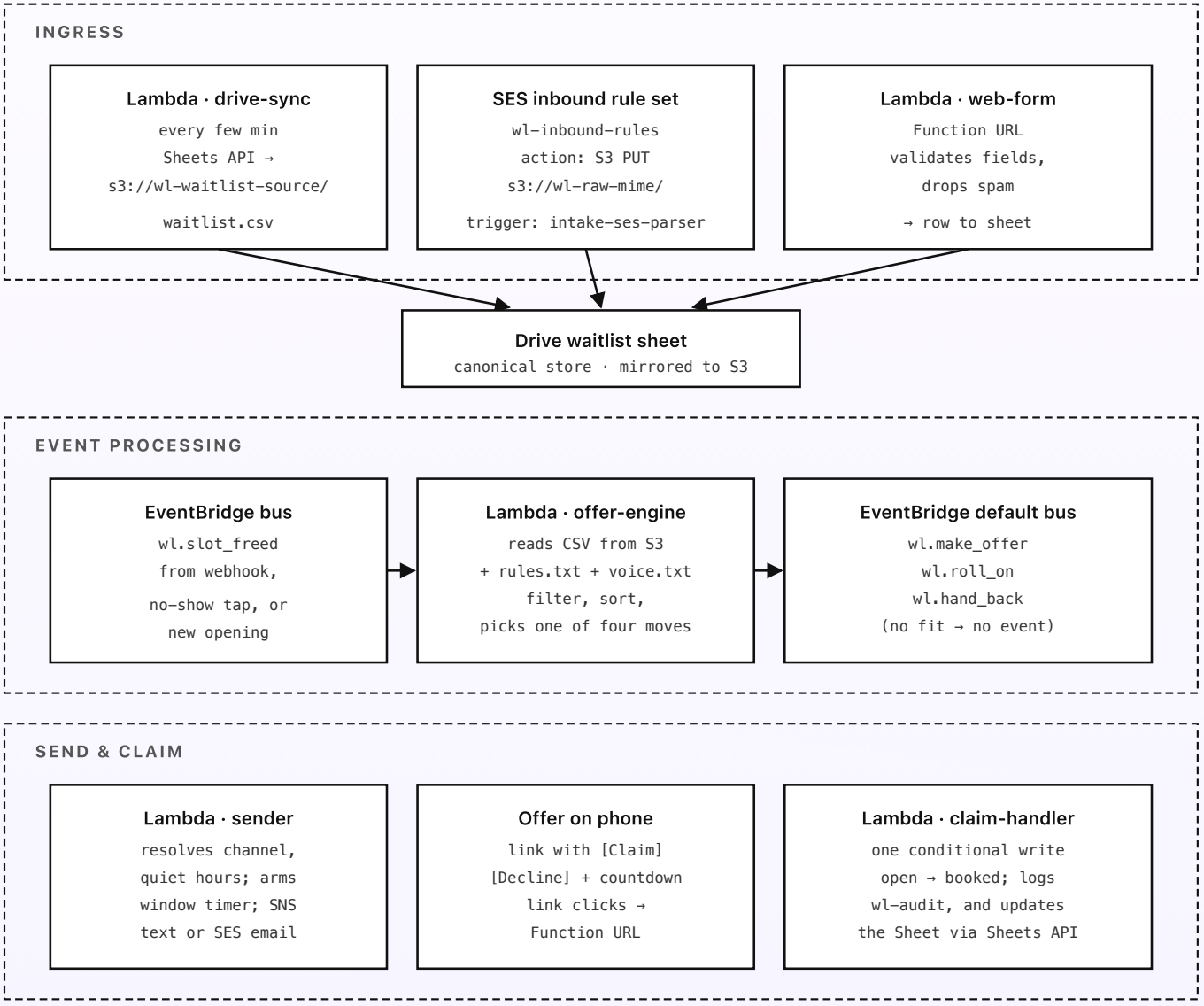
Engineering reference: the waitlist manager architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, EventBridge Scheduler config, the DynamoDB schemas, and the conditional-write claim that guarantees no double-booking. 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, SNS SMS, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is one freed slot going unfilled, not a regional outage. One AWS account dedicated to the waitlist manager (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. SMS in some regions requires registering a sender ID or a 10DLC number; budget a day for that with your carrier.

Topology



One conditional write guarantees no double-booking — and every step is logged to wl-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the waitlist), event processing (the offer engine reacting to a freed slot and emitting move events), send and claim (the offer ships and the claim is resolved by one conditional write). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

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

- **`drive-sync`** — EventBridge Scheduler target, fires every few minutes (default `rate(3 minutes)`). Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `wl/drive/sa`) to export the waitlist sheet as CSV and write to `s3://wl-waitlist-source/waitlist.csv` only if the sheet has changed. The same pattern syncs the rules and voice docs to `s3://wl-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`web-form`** — Lambda Function URL, public with `AuthType: NONE`. Backs the hosted join-the-waitlist page. Validates fields, runs a lightweight spam/honeypot check and a per-IP rate limit (token bucket in `wl-ratelimit` with a short TTL), and writes a clean row to the Drive sheet via the Sheets API. Memory: 256 MB. Timeout: 15 s.
- **`intake-ses-parser`** — S3 PUT trigger on `s3://wl-raw-mime/`. Parses MIME; if there's a PDF or image attachment, runs Textract via `StartDocumentTextDetection` (async via SNS completion); otherwise reads

the body text. Calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to propose a waitlist row, and posts an Approve/Edit/Discard card to the staff channel. For DOCX attachments (Textract doesn't accept them), falls back to `python-docx`. Memory: 512 MB. Timeout: 60 s.

- **`offer-engine`** — EventBridge rule on `wl.slot_freed` (and on the internal `wl.roll_on` re-entry). Reads `s3://wl-waitlist-source/waitlist.csv` and the rules and voice docs. Filters by eligibility (service, party size, date window, staff preference), sorts by the rules-doc order (join time, lifted by priority), reads `wl-offers` for live/tried state, decides on a move. Emits one event per slot that needs action: `wl.make_offer`, `wl.roll_on`, or `wl.hand_back`. *No Bedrock calls.* Memory: 512 MB. Timeout: 60 s.
- **`sender`** — EventBridge rule on the offer events. Resolves channel (mobile → SNS, else email → SES), checks quiet hours, writes the live offer to `wl-offers` with `window_ends_at`, mints a claim token (HMAC over `slot_id|offer_seq|exp` with a secret in `wl/claim/signing-key`), arms a one-off EventBridge Scheduler roll-on timer, formats the message from the voice template, and ships via SNS `Publish` or SES `SendRawEmail`. On quiet-hours defer, creates a Scheduler one-off that re-invokes `sender` at the next business minute (window armed only on actual send). Memory: 256 MB. Timeout: 30 s.
- **`claim-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies the HMAC claim token and its expiry. On *claim*, runs one DynamoDB `UpdateItem` on `wl-slots` with a condition expression `status = :open AND live_offer_seq = :seq`, setting `status = booked, booked_by = entry_id`; on success, confirms to the customer, deletes the roll-on Scheduler one-off,

and updates the Drive sheet via the Sheets API. On `ConditionalCheckFailedException`, returns the "just filled" page. On *decline*, marks the offer declined in `wl-offers` and puts a `wl.roll_on` event. Writes `wl-audit` on every path. Memory: 256 MB. Timeout: 15 s.

- **`roll-on-timer`** — target of each one-off claim-window Scheduler rule. Re-reads `wl-slots`; if the slot is still `open`, marks the live offer `timed_out` in `wl-offers` and puts a `wl.roll_on` event; if booked, no-op. Idempotent on `(slot_id, offer_seq)`. Memory: 256 MB. Timeout: 15 s.
- **`housekeeping`** — EventBridge Scheduler target, daily. Expires stale entries past their latest date, removes booked customers from the active list, and reconciles any slot left in an in-between state (e.g. a Scheduler rule that failed to fire). No Bedrock. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `wl-offers`, `wl-slots`, and `wl-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph owner narrative (slots freed, slots filled, average time-to-fill, estimated recovered revenue); emails it via SES. Memory: 512 MB.

Storage

- **DynamoDB · `wl-slots`** — one row per freed slot. PK `slot_id`; attributes: `service`, `slot_datetime`, `party_capacity`, `staff`, `status` (open/booked/handed_back), `live_offer_seq`, `booked_by`. The conditional-write target. On-demand.
- **DynamoDB · `wl-offers`** — one row per offer attempt. PK `(slot_id, offer_seq)`; attributes: `entry_id`, `channel`, `sent_at`, `window_ends_at`

(epoch), `status` (live/claimed/declined/timed_out). GSI on `entry_id` for per-customer history. On-demand.

- **DynamoDB** · `wl-audit` — one row per action of any kind. PK `(slot_id, ts)`; attributes: `entry_id`, `outcome` (offered/claimed/declined/timed_out/handed_back), `by`, `notes`. No TTL — long-term audit trail. On-demand.
- **DynamoDB** · `wl-ratelimit` — per-IP token bucket for the web form. PK `ip`; short TTL on each item. On-demand.
- **S3** · `wl-waitlist-source` — mirrored CSV from the Drive waitlist sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 3 years.
- **S3** · `wl-rules-source` — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3** · `wl-raw-mime` — raw inbound MIME from forwarded requests. Lifecycle to Glacier at 30 days; expiry at 1 year.

The no-double-booking guarantee

The whole safety property rests on a single DynamoDB conditional write. A slot lives in `wl-slots` with `status = open` and a `live_offer_seq` set by the sender. `claim-handler` issues `UpdateItem` with `ConditionExpression: status = :open AND live_offer_seq = :seq`. DynamoDB guarantees that condition is evaluated atomically against the current item, so for any number of concurrent claims only one can satisfy it — the rest get `ConditionalCheckFailedException`. The token binds a link to one `offer_seq`, so a stale link from a rolled-on offer fails the `live_offer_seq` check even before the status check. The roll-on timer

re-reads status and is a no-op on a booked slot. No locks, no transactions across tables, no read-then-write race — one write decides it.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `intake-ses-parser` for inbound request parsing, and `summary` for the monthly owner narrative. Sonnet 4.6 (`anthropic.claude-sonnet-4-6-...`) is available as a swap on the summary if richer analysis is ever wanted, but Haiku is plenty for a single paragraph.
- **Embeddings.** Not used. The waitlist is structured rows; deterministic filter-and-sort beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The offer path doesn't call Bedrock; the parsing lane fires a few times a month at most.

EventBridge Scheduler config

- **`wl-drive-sync`** — `rate(3 minutes)`. Target: `drive-sync` Lambda.
- **`wl-housekeeping`** — `cron(0 3 * * ? *)` in TZ. Target: `housekeeping` Lambda.
- **`wl-monthly-summary`** — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **Claim-window one-offs** — created by `sender` per live offer. Use `at(YYYY-MM-DDTHH:MM:SS)` at `window_ends_at` with `--action-after-completion DELETE`

so the rule self-cleans; target `roll-on-timer`. Deleted early by `claim-handler` on a successful claim.

- **Quiet-hours defer one-offs** — created by `sender` when an offer is held; `at(...)` at the next business minute, target `sender`.

SES, SNS, inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `waitlist.your-business.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `wl-inbound-rules`: one rule with recipient `waitlist@your-business.com` → spam scan → S3 PUT to `s3://wl-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for email-fallback offers and the monthly summary: verify a sender identity at `waitlist@your-business.com` with DKIM and SPF on the parent domain. Out of sandbox by request.
- SNS for the offer texts: an origination identity (sender ID or 10DLC long code) registered for the destination country; per-message `Publish` with an SMS attribute set to `Transactional` for delivery priority.

IAM (least privilege per Lambda)

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

- **offer-engine role:** `s3:GetObject` on the waitlist, rules, and voice keys; `dynamodb:Query` + `GetItem` on `wl-offers`, `wl-slots`; `events:PutEvents` on the default bus. *No* `bedrock:*`.

- **sender role:** `scheduler:CreateSchedule` + `DeleteSchedule` for the window and defer one-offs; `secretsmanager:GetSecretValue` on the claim signing key; `sns:Publish`; `ses:SendRawEmail` from the verified identity; `dynamodb:PutItem` on `wl-offers`.
- **claim-handler role:** `dynamodb:UpdateItem` on `wl-slots` (the conditional write); `dynamodb:PutItem` on `wl-offers` and `wl-audit`; `scheduler:DeleteSchedule` for the window one-off; `events:PutEvents` for roll-on; `secretsmanager:GetSecretValue` on the claim signing key and the Sheets service-account secret; outbound network to `sheets.googleapis.com`.
- **intake-ses-parser role:** `s3:GetObject` on `wl-raw-mime`; `textract:StartDocumentTextDetection`; `bedrock:InvokeModel` on the Haiku ARN; outbound network for the staff-channel post.
- **drive-sync and web-form roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the waitlist and rules buckets (drive-sync); `dynamodb:UpdateItem` on `wl-ratelimit` (web-form); outbound network to `www.googleapis.com`.

Freed-slot sources

Three producers put a `wl.slot_freed` event on the bus. A **booking-tool webhook** hits a thin Function URL (`slot-webhook`, signature-verified) on a cancellation. A **no-show tap** from the front-desk view hits the same Function URL with a no-show action. A **calendar-opening sync** (optional, if availability lives in Google Calendar) runs on the housekeeping tick and emits a freed-slot event for any newly added gap. All three normalize to the same event shape — `service`,

`slot_datetime`, `party_capacity`, `staff` — so the engine has one input contract.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a metric for alerting.
- **Alarms:** `claim-handler` 5xx > 0 (a customer can't claim); `roll-on-timer` failures > 0 (a slot could stall); SNS SMS delivery-failure rate > 5% (carrier or origination-id issue); offer-engine errors > 0.
- **SQS + DLQ:** the EventBridge targets use an SQS dead-letter queue so a failed roll-on or send can be replayed instead of lost.
- **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 `wl-cost-alarm` subscribed to the on-call admin's email.

Config and secrets

Service-account credentials for Drive and Sheets live in Secrets Manager under `wl/drive/sa`. The claim-token signing key is `wl/claim/signing-key`; the booking-webhook signing secret is `wl/webhook/secret`. The configured timezone, quiet-hours window, claim-window length, per-slot try cap, and down-fit policy all live in Parameter Store under `/wl/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment. The rules and

voice docs are read fresh from S3 per invocation so staff edits take effect on the next freed slot without a deploy.

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 both `wl-waitlist-source` and `wl-rules-source` so a bad Drive edit rolls back in one click, and pin the EventBridge Scheduler timezone so a CI rotation can't silently start the housekeeping tick in UTC. Total deployable surface: around nine Lambdas, four DynamoDB tables, three S3 buckets, one EventBridge rule set on the default bus (plus the Scheduler rules), one SES rule set, one SNS origination identity, one SQS DLQ, 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.