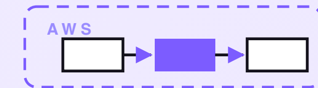
7-PART SERIES · FREE COMPANION



Event RSVP manager

A serverless manager that runs the guest list for a small event. People register; it confirms each RSVP, sends timely reminders before the date, handles cancellations and a waitlist that auto-offers a freed spot to the next person, caps capacity so the event never oversells, and gives the host a live headcount. Guests can confirm, cancel, or claim an offered spot right from the email. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

Workflow guide $19 · Deployable AWS CDK starter $79 · Bundle $89

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/event-rsvp-manager

CONTENTS

Event RSVP manager

01 An event RSVP manager on AWS for a few dollars a month

02 How an RSVP gets confirmed

03 How an RSVP reminder gets scheduled

04 How a cancellation frees a spot

05 How a freed spot finds the next guest

06 What the event RSVP manager costs

07 Engineering reference: the event RSVP manager architecture

PART 1 OF 7

JUNE 3, 2026 PART 1 OF 7 · EVENT RSVP MANAGER SERIES ~5 MIN READ

An event RSVP manager on AWS for a few dollars a month

Running the guest list for a small event is more fiddly than it looks. The room holds 80 and you must not let 95 people show up. Three people RSVP for the last two seats in the same minute. Two days out, half the “yes” crowd has forgotten the time. Someone cancels the night before and the seat just sits empty while ten people would have gladly taken it. This post walks through the design of a small manager that runs all of that: it confirms each RSVP, reminds people at the right time, handles cancellations cleanly, offers freed seats to a waitlist, and never lets the event oversell.

KEY TAKEAWAYS

- Three outside pieces: a register form, the event setup, and the guests who get emails.
- Every guest moves through a few clear states: pending, confirmed, waitlisted, cancelled.
- The capacity cap is enforced by the database, so the event can never oversell.
- Reminders and waitlist offers respect quiet hours and the host's settings.
- Designed on AWS for about $2.40 for a typical 200-guest event.

The whole system on one page

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

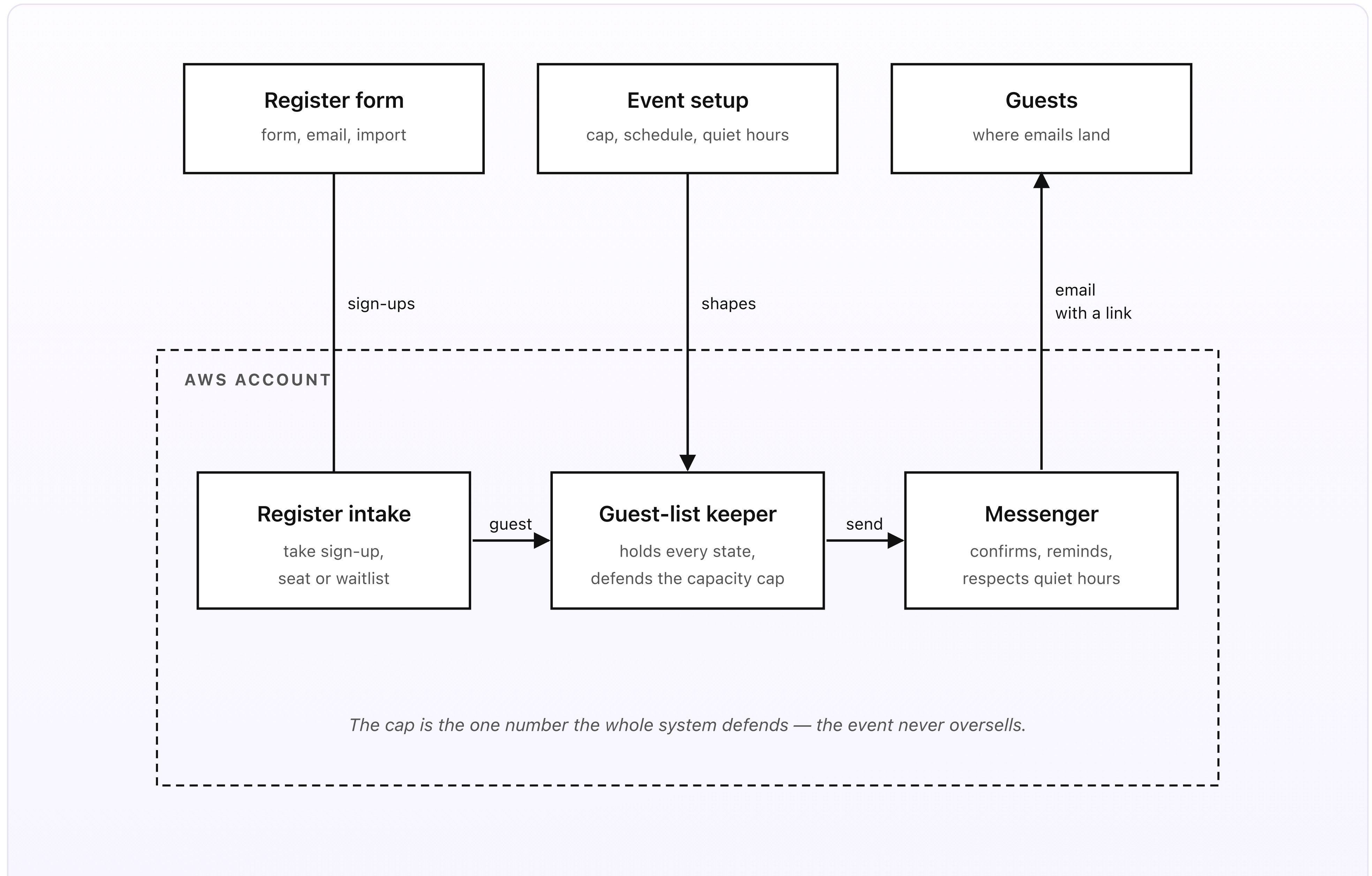

Register form
form, email, import
Event setup
cap, schedule, quiet hours
Guests
where emails land
sign-ups
shapes
email
with a link
AWS ACCOUNT
Register intake
take sign-up,
seat or waitlist
guest
Guest-list keeper
holds every state,
defends the capacity cap
send
Messenger
confirms, reminds,
respects quiet hours
The cap is the one number the whole system defends — the event never oversells.

Fig 1. Three pieces outside, three pieces inside AWS. Sign-ups flow in from a register form, an email reply lane, and a host import. The Guest-list keeper holds every guest's state and defends the cap. The Messenger sends the right email to the right person at the right time.

What you set up once (the outside)

- **The register form.** A simple web page where people enter their name and email to ask for a seat. It can live anywhere — a page on your site, a link in a flyer, a QR code at the door. The form posts to the system, which then either confirms a seat (if there's room) or places the person on the waitlist (if the event is full). Two other lanes can also add people, covered in Part 2 — an email reply lane (someone replies "yes" to your invite and the system reads it) and a host import lane (the host pastes a list of names the system signs up in bulk).
- **The event setup.** A short set of host settings. The event name, date and time, and venue. The *capacity cap* — the single most important number, the most confirmed guests the room can hold. The reminder schedule — how far ahead to remind people (a week out, a day out, the morning of). The quiet-hours window so no reminder lands at 2am. And the waitlist claim window — how long a freed seat is held for the next person before it rolls on. None of these need a developer to change; they live in a settings record the host can edit.
- **The guests.** The people on the list. Each one has a name, an email, and a state: pending (signed up, not yet confirmed), confirmed (holds a seat), waitlisted (in line for a seat), or cancelled. Emails land with the event details, a confirm or cancel link, and — for a waitlist offer — a claim link with a clear deadline. No app to install; everything happens in the inbox.

What runs inside (the system)

- **The register intake.** Takes a sign-up from any of the three lanes. Checks whether this email is already on the list (so nobody double-books). Then asks the guest-list keeper for a seat. If a seat is granted, the messenger sends a confirmation. If the event is full, the person is placed on the waitlist and told their position. Simple, deterministic, no guesswork.
- **The guest-list keeper.** Holds the state of every guest and the live confirmed count. This is where the capacity cap is enforced: a seat is claimed by a single conditional write — a database update that only succeeds if the confirmed count is still below the cap. Two people racing for the last seat can't both win; the database lets exactly one through and refuses the other. The keeper also decides what happens on a cancel (release the seat, start the waitlist offer) and on a claim (move a waitlisted guest into a confirmed seat).
- **The messenger.** Sends every email: the confirmation when a seat is granted, the reminders on the host's schedule, the waitlist offer when a seat frees up, and the post-event thank-you. It reads the quiet-hours window and never sends a reminder overnight — a message that would land in the quiet window is held until the next morning. Every email is recorded so the system knows what already went out and never sends a duplicate.

In plain words

Your workshop seats 40. The form goes live and 38 people confirm over a week; each one gets a clean confirmation email the moment they click. Two days out, everyone who confirmed gets a friendly reminder with the time and address — sent at 9am, never overnight. The night before, two people cancel with one click.

Each freed seat is immediately offered to the first person on the waitlist with a link that holds the seat for six hours. The first waitlister claims theirs in twenty minutes; the second never opens the email, so at the six-hour mark the offer rolls on to the next person in line, who claims it the next morning. On the day, the room has exactly 40 confirmed guests — not 38, not 42 — and the host never touched a spreadsheet.

The cost of running this is about $2.40 for a 200-guest event. The cost of *not* running it is the overbooked room you have to turn people away from, or the half-empty room because cancellations were never back-filled, or the no-shows who simply forgot the time.

DESIGN RULES THAT SHAPED EVERY DECISION

- The cap is sacred. A seat is claimed by one conditional write; the event can never oversell, even under a race.
- A freed seat is offered, not given. The waitlist gets a timed claim link; a stale offer rolls on, never blocks the seat.
- Quiet hours are respected. A reminder that would land overnight waits until morning.
- Every email is logged. The system never sends the same reminder twice.
- The host settings live in one record — change the cap or the schedule without a deploy.
- Every state change is recorded. After the event, the guest list is fully reconstructable.

Why this shape

Most small events are run on one of three things: a spreadsheet someone updates by hand, a free form tool that emails you raw responses, or a calendar invite with a headcount nobody trusts. The spreadsheet oversells the moment two people fill the last seat before anyone refreshes. The form tool has no concept of a cap, a waitlist, or a reminder — it just collects rows. And the calendar invite tells you who said yes three weeks ago, not who’s actually coming.

The setup above keeps the simple parts simple — a form, an inbox, a settings record — but adds a small system that *defends the cap*, *fills cancellations automatically*, and *reminds people at the right time*. It never oversells because the database won’t let it. It never leaves a freed seat empty because the waitlist offer fires the moment a cancellation lands. And it never pings someone at midnight. The host sees a live headcount and otherwise leaves it alone.

The next four posts walk through each piece in turn: how an RSVP gets confirmed, how a reminder gets scheduled, how a cancellation frees a spot, and how that freed spot finds the next guest. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 3, 2026 PART 2 OF 7 · EVENT RSVP MANAGER SERIES ~5 MIN READ

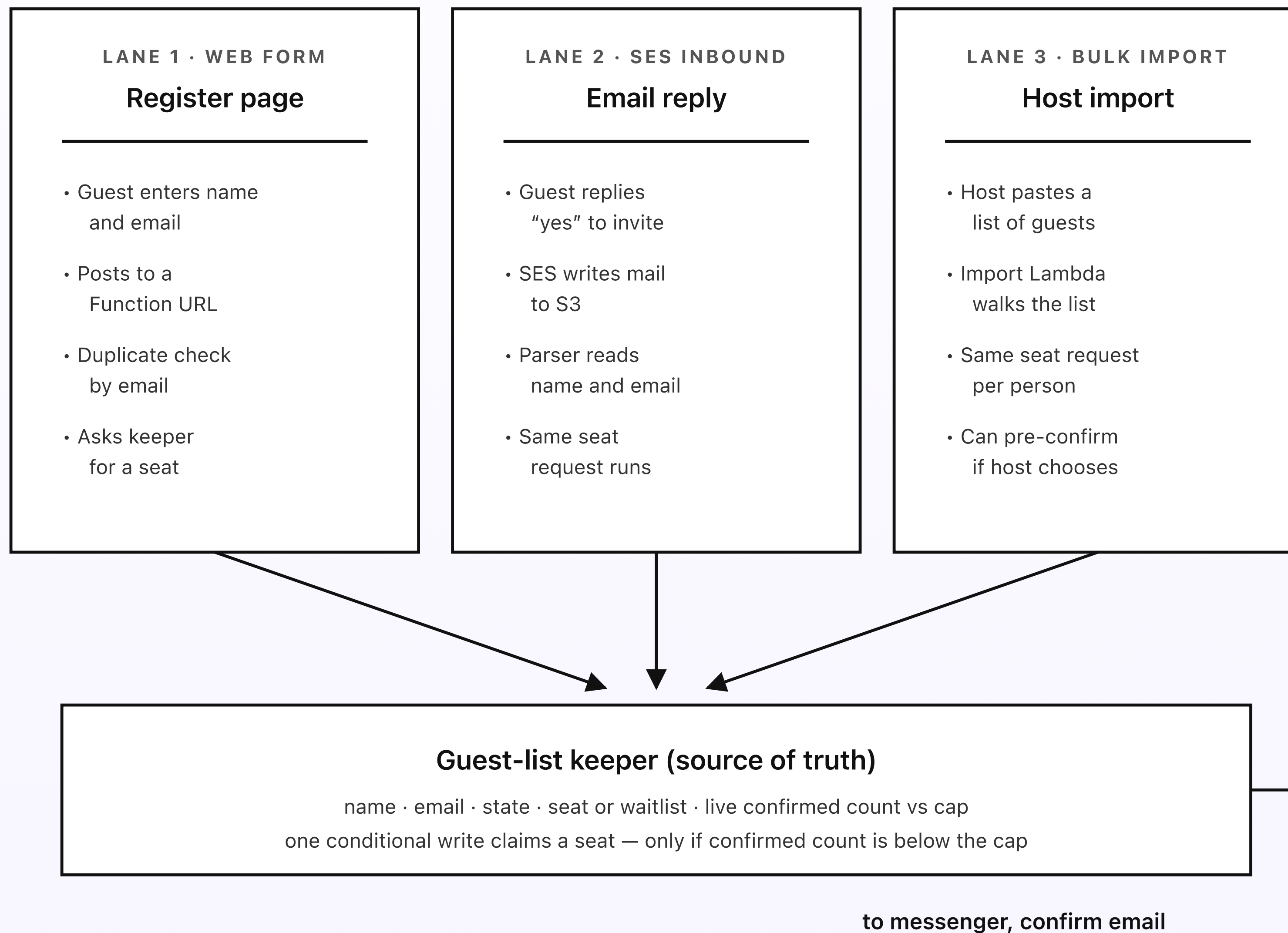
How an RSVP gets confirmed

A guest goes from “I’d like to come” to “I have a seat” in one step, and that one step has to be airtight. There are three ways a guest gets onto the list: they fill the web form, they reply to your invite email, or the host imports a batch of names. All three end at the same place — a request for a seat. And that request is granted by a single, careful database write that makes overselling impossible, even when two people reach for the last seat in the same second.

KEY TAKEAWAYS

- Three sign-up lanes feed one list: the web form, an email reply, and a host import.
- A new sign-up is checked for duplicates so nobody double-books a seat.
- A seat is claimed by a single conditional write — it only succeeds if the event isn't full.
- If the event is full, the person is placed on the waitlist and told their position.
- The capacity cap is defended by the database, not by hopeful counting in code.

Three lanes onto one list



The keeper is the source of truth — the three lanes are just different doors into the same seat request.

Fig 2. Three lanes converge on one guest-list keeper. The web form, the email reply, and the host import all end at the same seat request. The keeper grants a seat with a single conditional write that only succeeds when the event isn't full.

Lane 1: the web form

The simplest lane. The register page asks for a name and email and posts to a Function URL — a plain web address that runs a small Lambda, with no API Gateway in front of it. The Lambda first checks whether this email is already on the list: if it is, the guest just gets their existing status back ("you're already confirmed" or "you're number 4 on the waitlist") instead of a second row. If it's a new email, the Lambda asks the guest-list keeper for a seat. Most guests come in this way.

The form itself is static — it can sit on any page you already have. There's no login, no account to create. The whole interaction is: type two fields, click, get an email. That low friction is the point; the easier it is to RSVP, the more accurate your headcount.

Lane 2: the email reply

Plenty of people will just reply "yes, count me in" to your invite rather than click a form. Lane 2 catches them. Set up a dedicated inbound address through Amazon SES — something like `rsvp@your-event.com`. When a reply lands, SES writes the raw message to S3, which triggers a small parser Lambda. The parser reads the sender's name and email from the message headers and runs the same seat request the form would. A short confirmation (or waitlist notice) goes back to the same address.

The parser keeps it deliberately simple: it only needs the name and email, both of which are in the message envelope. It does not try to interpret the body for clever intent — a reply to the RSVP address *is* the intent. If someone replies to cancel instead, the body is scanned for a clear “cancel” or “can’t make it,” and on a match the guest is routed to the cancel flow from Part 4 rather than a sign-up.

Lane 3: host import

Sometimes the host already has the list — a team roster, last year’s attendees, a sheet of names collected at a previous event. Forcing all of them to fill a form would be silly. Lane 3 lets the host paste a list of names and emails. An import Lambda walks the list and runs the same seat request for each person. The host chooses whether the imported guests are placed as *pending* (they still get an invite and have to confirm) or *pre-confirmed* (they’re straight onto the seat, useful for a guaranteed VIP list).

Even on a bulk import, every single person still goes through the same conditional write. If the import would push past the cap, the people beyond the cap land on the waitlist in list order — the import can’t sneak past the capacity limit any more than a single sign-up can.

The one write that defends the cap

Here is the heart of the whole system. Granting a seat is not “read the count, check it’s under the cap, then add one.” That two-step approach has a gap: two sign-ups can both read “39 of 40” at the same instant, both decide there’s room, and both write — and now you have 41 confirmed for a 40-seat room. Instead, the keeper claims a seat with a single *conditional write*: one database update that

says “increase the confirmed count by one, but only if it’s currently below 40.” The database runs that check and the increase as one indivisible step. If two writes race, exactly one succeeds and the other is refused. The loser isn’t an error — they’re simply placed on the waitlist instead.

This is why the cap can be trusted absolutely. There is no moment where the count is read separately from being changed. The number of confirmed guests is never wrong, never even briefly over the cap, no matter how many people click at once.

What the guest sees

From the guest’s side, none of this is visible. They sign up and within a second or two they get one of two emails. *Confirmed*: “You’re in — here are the details, and a link to cancel if your plans change.” *Waitlisted*: “The event is full, but you’re number 3 in line. If a seat frees up we’ll email you a link to claim it.” Both are clear, both set the right expectation, and both contain the links the rest of the system needs.

Next post: how the system schedules each confirmed guest’s reminders — a week out, a day out, the morning of — and how quiet hours keep any of them from landing in the middle of the night.

PART 3 OF 7

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

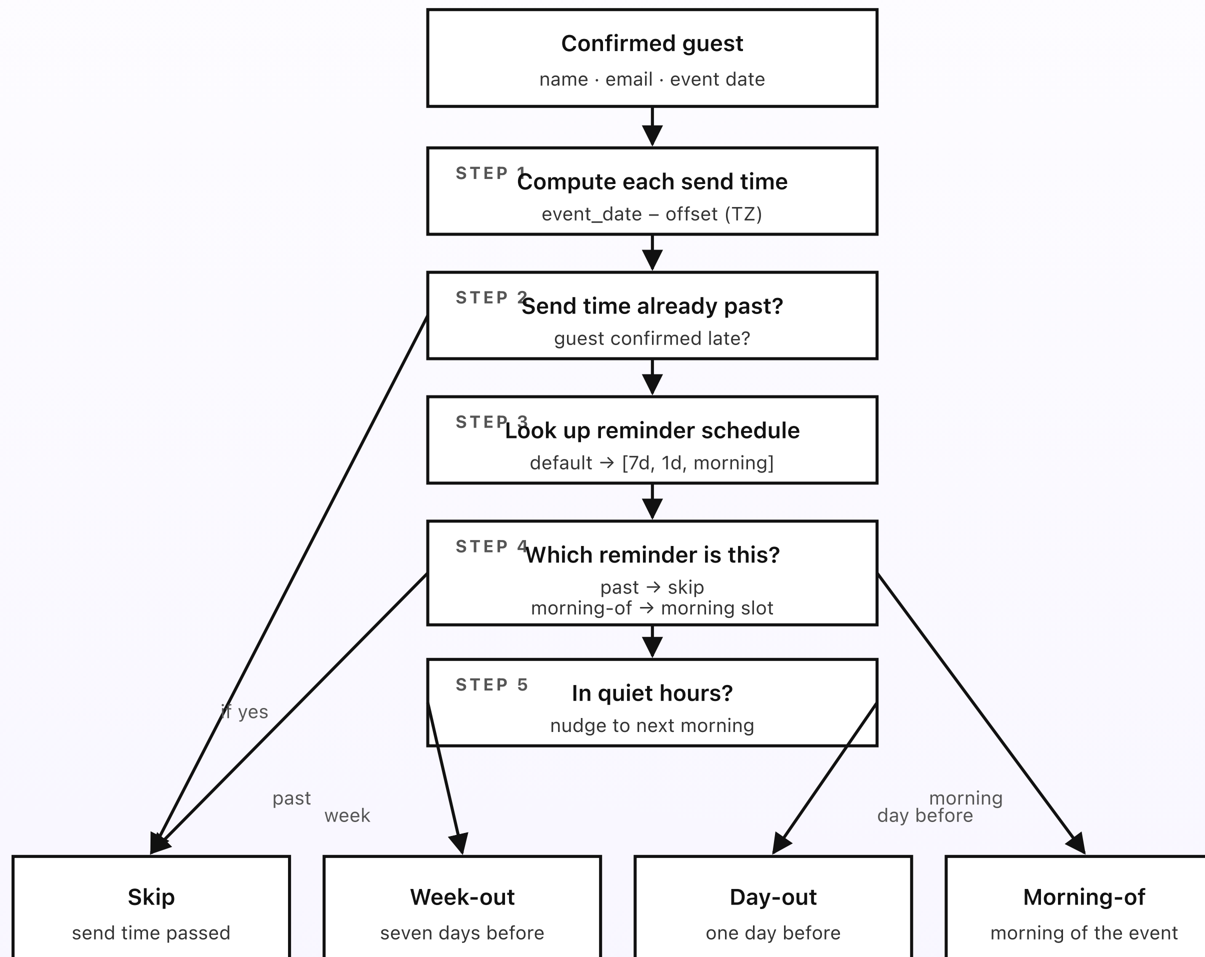
How an RSVP reminder gets scheduled

The moment a guest confirms, the system works out exactly when to remind them — a week before, a day before, the morning of — and books each one as its own timed job. It doesn’t poll a list every hour looking for who’s due; it sets an alarm for each reminder and walks away. The whole decision is plain date arithmetic. The reminder schedule lives in the host’s settings, where it can be changed without a developer.

KEY TAKEAWAYS

- When a guest confirms, the system books their reminders as one-off EventBridge Scheduler rules.
- The schedule lives in the host settings — the default is a week out, a day out, and the morning of.
- Each send time is computed from the event date, then nudged out of quiet hours.
- A reminder already past (the guest confirmed late) is simply skipped, not sent stale.
- No always-on poller. Each reminder is its own alarm; the system sleeps in between.

The scheduling decision, per reminder



The schedule lives in the host settings — change an offset and new confirmations use it.

Fig 3. The scheduling decision, per reminder, the moment a guest confirms. Five steps decide which timed jobs to book. The host settings hold the schedule; the system only computes the times and books the alarms.

Why one-off alarms instead of a poller

There are two ways to send reminders. One is a poller: every hour, wake up, read the whole guest list, and ask “is anyone due a reminder right now?” That works, but it runs constantly whether anything is due or not, and it has to be careful not to send a reminder twice. The other way — the one used here — is to set an alarm for each reminder at the moment the guest confirms. When the guest clicks confirm, the system computes the three send times and books three one-off jobs through EventBridge Scheduler, each pointed at the exact minute it should fire.

The alarm approach is cheaper (nothing runs in between) and simpler to reason about (each reminder is one job that fires once). When the job fires, it invokes the messenger to send that one email, then the job deletes itself. There’s no list to scan, no “did I already send this” bookkeeping on a poll loop — the alarm either fired or it didn’t.

Computing the send times

The schedule is a short list of offsets in the host settings. The default reads, in plain words: “Remind seven days before, one day before, and the morning of.” For each offset, the system subtracts it from the event date and time, in the event’s configured timezone, to get a target send time. A 6pm event on the 20th gives a

week-out reminder on the 13th, a day-out on the 19th, and a morning-of at, say, 9am on the 20th.

Two small adjustments keep the times sensible. First, the *morning-of* reminder is pinned to a friendly hour (9am by default) rather than literally “the event time minus zero,” because a reminder at the exact start time is useless. Second, any computed send time that lands inside quiet hours is nudged forward to the next morning — covered just below.

Quiet hours: never a 2am reminder

The host settings include a quiet-hours window, 9pm to 8am local by default. Before a reminder job is booked, its send time is checked against that window. If the time falls inside quiet hours — for instance, a day-out reminder for a late-night event computing to 11:30pm — the send time is moved to the start of the next morning’s business window. The guest gets their reminder at 8am instead of nearly midnight. The job is booked at the nudged time, so quiet hours are enforced once, at scheduling, and never need rechecking when the job fires.

Quiet hours apply to reminders and waitlist offers, not to the immediate confirmation email — that one is a direct response to the guest’s own click, so it always goes out right away regardless of the hour.

Late confirmations and skips

Not every reminder applies to every guest. Someone who confirms two days before the event has already missed the week-out window. Rather than fire a stale

“your event is in seven days” email a week too late, the system checks each computed send time against the current time and simply skips any that are already in the past. The late-confirming guest still gets the day-out and morning-of reminders; they just never get the week-out one. No stale emails, no awkward timing.

Each booked reminder is also recorded against the guest in DynamoDB, so if the guest later cancels, the system knows exactly which pending reminder jobs to delete — the subject of the next post. A cancelled guest must never get a “see you tomorrow!” reminder.

Next post: how a cancellation frees a spot — releasing the seat, cleaning up the guest’s pending reminders, and handing the freed seat to the waitlist.

PART 4 OF 7

JUNE 3, 2026 PART 4 OF 7 · EVENT RSVP MANAGER SERIES ~5 MIN READ

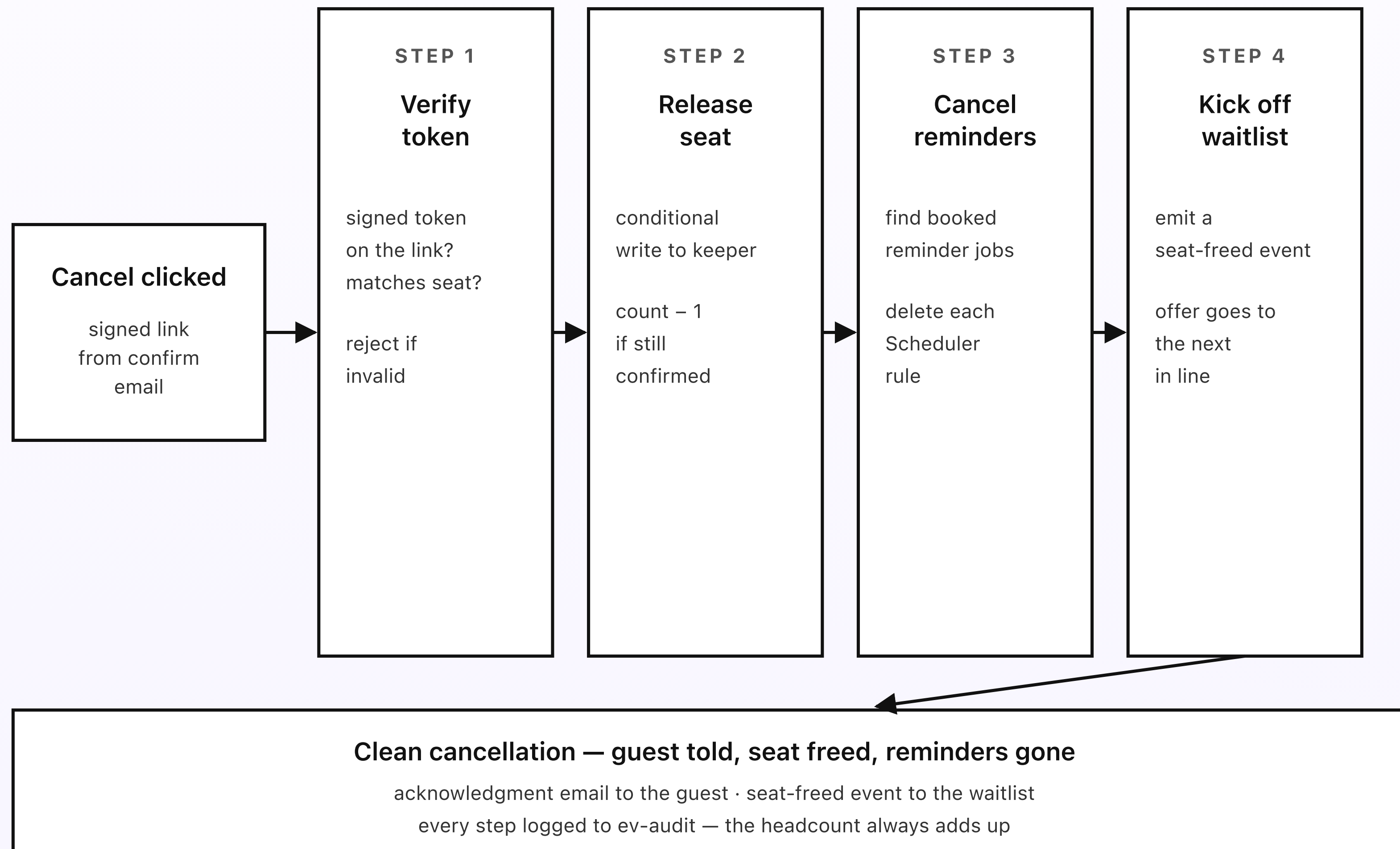
How a cancellation frees a spot

A guest can’t make it, so they click “cancel” in their confirmation email. That one click has to do four things in the right order: prove it’s really them, hand back the seat, stop the reminders they no longer need, and offer the freed seat to the waitlist. Do any of them wrong and you get an empty seat that should have been filled, a reminder to someone who isn’t coming, or a headcount that no longer adds up. Four small steps sit between the click and a clean cancellation.

KEY TAKEAWAYS

- The cancel link carries a signed token, so only the real guest can cancel their seat.
- Releasing the seat is a conditional write that lowers the confirmed count by one.
- The guest’s pending reminder jobs are deleted so they never get a stale reminder.
- A freed seat immediately kicks off the waitlist offer covered in the next post.
- Every step is logged, so the live headcount is always exactly right.

Four steps on every cancellation



Every step is a deterministic check — no model calls, no guessing who freed which seat.

Fig 4. Four steps between a cancel click and a clean cancellation. Verify the token. Release the seat. Cancel the reminders. Kick off the waitlist. Then tell the guest and log every step so the headcount always adds up.

Step 1: verify the token

The cancel link in every confirmation email isn't a plain web address — it carries a signed token, a short string the system created and signed when the guest confirmed. The token identifies exactly which seat the link belongs to, and the signature proves the system made it. When the link is clicked, the cancel Lambda checks the signature before doing anything else. A tampered or guessed link fails the check and is rejected. This keeps one guest from cancelling another's seat just by editing a web address, and it means the cancel flow needs no login — the signed link is the proof of identity.

Tokens are scoped to the event and have a sensible lifetime (they stop working once the event has passed). A guest who lost their email can ask the host to re-send the link, which mints a fresh token for the same seat.

Step 2: release the seat

Handing the seat back is the mirror image of claiming it. Just as a seat was granted by a conditional write that only succeeded if the event wasn't full, releasing it is a conditional write that only succeeds if the guest is *actually still confirmed*. That condition matters: if the guest clicks the cancel link twice (or clicks it, then the page reloads and fires again), the second attempt finds the

guest already cancelled and does nothing. The confirmed count drops by exactly one, never two. The cap defends itself going down just as it does going up.

The guest's state moves to *cancelled*. Their row stays on the list — nothing is deleted — so the audit trail and the headcount history stay intact. They simply no longer hold a seat.

Step 3: cancel the reminders

Back in Part 3, the moment this guest confirmed, the system booked their reminders as one-off Scheduler jobs and recorded the job names against the guest. Step 3 reads those job names and deletes each one. This is the step people forget, and it's the one guests notice most: nothing looks worse than getting a cheerful “See you tomorrow!” reminder the day after you cancelled. Because each reminder is its own named job, cleanup is exact — the system deletes precisely this guest's jobs and touches no one else's.

If a reminder job has already fired (the week-out reminder went out before the guest cancelled), there's nothing to delete for that one — it's already done. Only the still-pending jobs are removed.

Step 4: kick off the waitlist

A freed seat is no good if it just sits empty. The final step emits a small “seat freed” event that starts the waitlist offer — the subject of the next post. The cancel flow itself doesn't pick the next guest or send the offer; it just announces that a seat is open and lets the waitlist logic take over. Keeping the two flows

separate means a cancellation is fast and simple, and the waitlist offer (which has its own timing and claim window) runs on its own.

One small but important detail: the seat-freed event fires only if a seat was genuinely released in Step 2. A double-clicked cancel that did nothing the second time doesn't fire a second waitlist offer. One freed seat, one offer.

Why this order, and why it's logged

The order is deliberate: verify before changing anything, release the seat before announcing it's free, clean up reminders so the cancelled guest goes quiet, and only then hand the seat onward. Every step writes a row to the `ev-audit` table — who cancelled, when, which seat, and what the count went from and to. After the event, the host can reconstruct the entire guest list: who confirmed, who cancelled, when each seat changed hands. The live headcount the host watches is just the current state of this same data, so it's never out of sync with reality.

Next post: how that freed seat finds the next guest — the timed waitlist offer, the claim window, and what happens when an offer goes unclaimed.

PART 5 OF 7

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

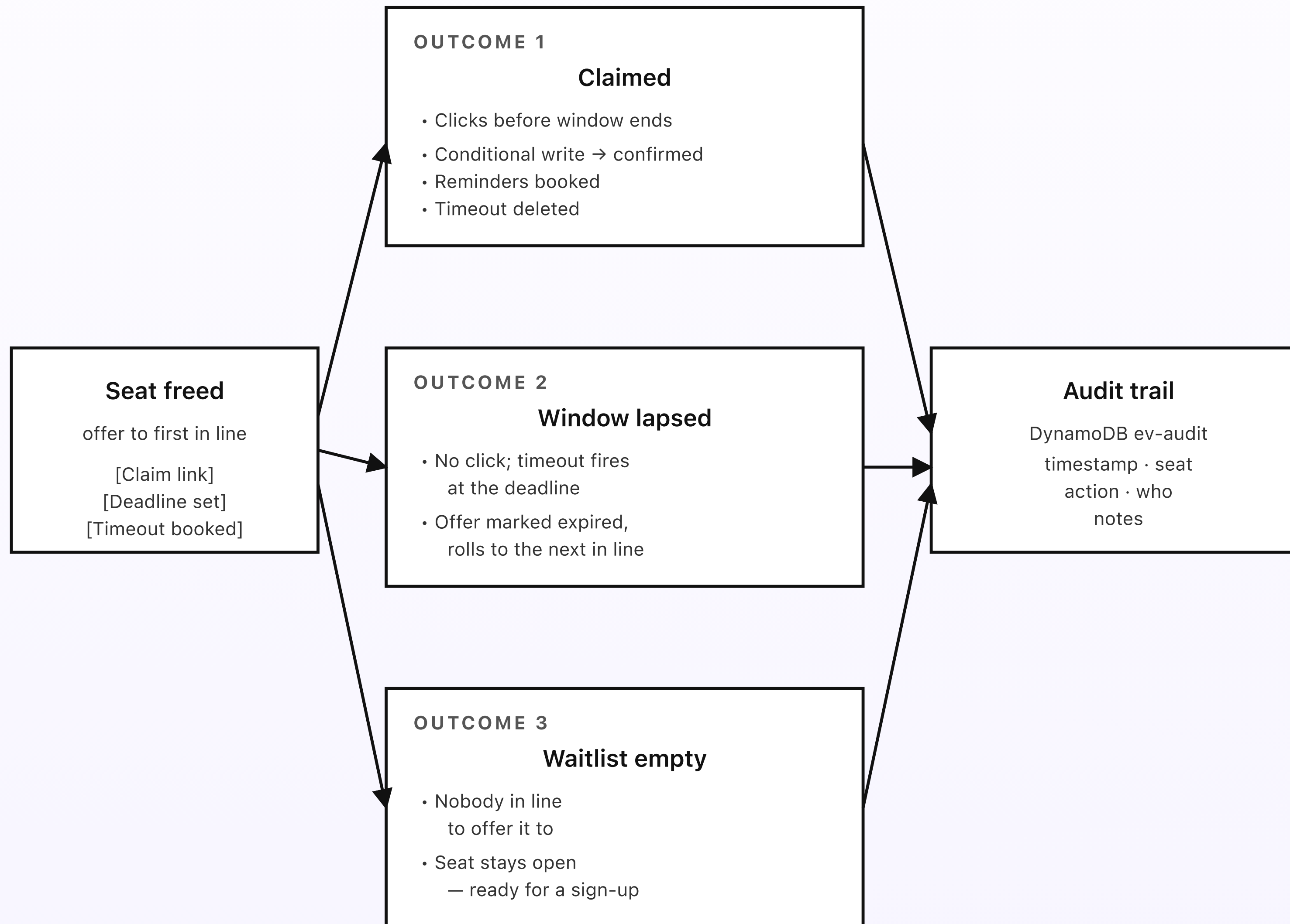
How a freed spot finds the next guest

A seat just opened and three people are waiting for it. The fair, safe way to fill it isn’t to grab the first waitlister and confirm them — they might have made other plans. It’s to *offer* the seat, hold it for them for a set window, and move on if they don’t take it. This post walks through the waitlist offer: who gets it, how long the seat is held, and the three things that can happen — the offer is claimed, the window lapses and the offer rolls on, or the waitlist is empty and the seat waits.

KEY TAKEAWAYS

- A freed seat is *offered* to the first person on the waitlist, not silently given to them.
- The offer holds the seat for a claim window (default 6 hours) via a one-off Scheduler timeout.
- If the offer is claimed, a conditional write moves the guest into the confirmed seat.
- If the window lapses, the offer rolls to the next person automatically — nobody is skipped.
- If the waitlist is empty, the seat simply stays open for a new sign-up.

Three outcomes of a waitlist offer



A freed seat is offered, not given — a stale offer rolls on, and the seat is never blocked forever.

Fig 5. Three outcomes of a waitlist offer. Claimed moves the guest into the seat. A lapsed window rolls the offer to the next person. An empty waitlist leaves the seat open. Every outcome writes to the audit trail.

The offer, not the grant

The most important design choice in the whole waitlist is that a freed seat is *offered*, not handed over. If the system simply confirmed the first waitlister the moment a seat opened, you'd end up confirming people who've already made other plans — and then chasing *their* cancellation. Instead, the first person in line gets an email: "A seat just opened. Click here to claim it — this offer holds your seat until 4pm today." The seat is reserved for them and only them until the deadline. They're in control, and the host doesn't end up with a confirmed guest who never wanted the seat.

When the offer goes out, the system books a one-off timeout job through EventBridge Scheduler set to the end of the claim window (six hours later by default). That timeout is the safety net that guarantees the seat never gets stuck on someone who isn't going to respond.

Outcome 1: the offer is claimed

The happy path. The waitlister clicks the claim link before the window closes. A Function URL Lambda runs a conditional write that moves them from *waitlisted* to *confirmed* — the same careful, indivisible update that grants a seat in the first place, so even here the cap can't be exceeded. Their reminders are booked exactly as a normal confirmation's would be (the flow from Part 3). The pending

timeout job is deleted, since the offer resolved before it needed to fire. And a confirmation email goes out: “You’re in — here are the details.” From the guest’s side, claiming a waitlist seat feels exactly like a normal confirmation.

Outcome 2: the window lapses

The waitlister never opens the email, or opens it too late. At the deadline, the timeout job fires. It first checks whether the offer was already claimed — if it was, there’s nothing to do. If it’s still outstanding, the offer is marked *expired* and the system re-emits the same “seat freed” event that started everything. That re-emission hands the seat to the *next* person on the waitlist, who gets their own fresh offer with their own six-hour window. The seat rolls down the line, one person at a time, until someone claims it or the list runs out. Nobody is silently skipped; everyone gets a real chance at the seat in turn.

This rolling behavior is why the offer-not-grant model is safe: a waitlister who’s gone quiet can’t block the seat forever. After their window, it simply moves on.

Outcome 3: the waitlist is empty

Sometimes a seat frees up and there’s nobody waiting — the event filled, then someone cancelled, but no one ever joined the waitlist. In that case there’s no one to offer the seat to, so the system does the simplest thing: it leaves the seat open. The confirmed count is already lowered (that happened when the seat was released), so the next person to hit the register form will find room and get confirmed straight away through the normal flow from Part 2. The host’s live headcount just shows one open seat until someone takes it.

Quiet hours and the one edge case

Waitlist offers respect quiet hours like any other message — an offer that would land at midnight is normally held until morning. But there's a deliberate exception. If holding the offer until morning would mean the claim window closes before a reasonable person could act — the event is tomorrow and the seat needs filling tonight — a host setting can let time-critical offers through the quiet window. The reasoning is simple: a late-night email is a small annoyance, but a seat that goes empty because the offer waited politely until 8am is a real loss. The host chooses which matters more for their event.

Every offer, claim, expiry, and roll-on writes a row to `ev-audit`. After the event, the host can see the full story of any seat: who held it, who cancelled, who was offered it, who let it lapse, and who finally took it.

Next post: the cost breakdown. The whole system above runs in coffee-money territory per event; Part 6 explains exactly where the dollars go and why most of the bill is just the emails.

PART 6 OF 7

JUNE 3, 2026 PART 6 OF 7 · EVENT RSVP MANAGER SERIES ~3 MIN READ

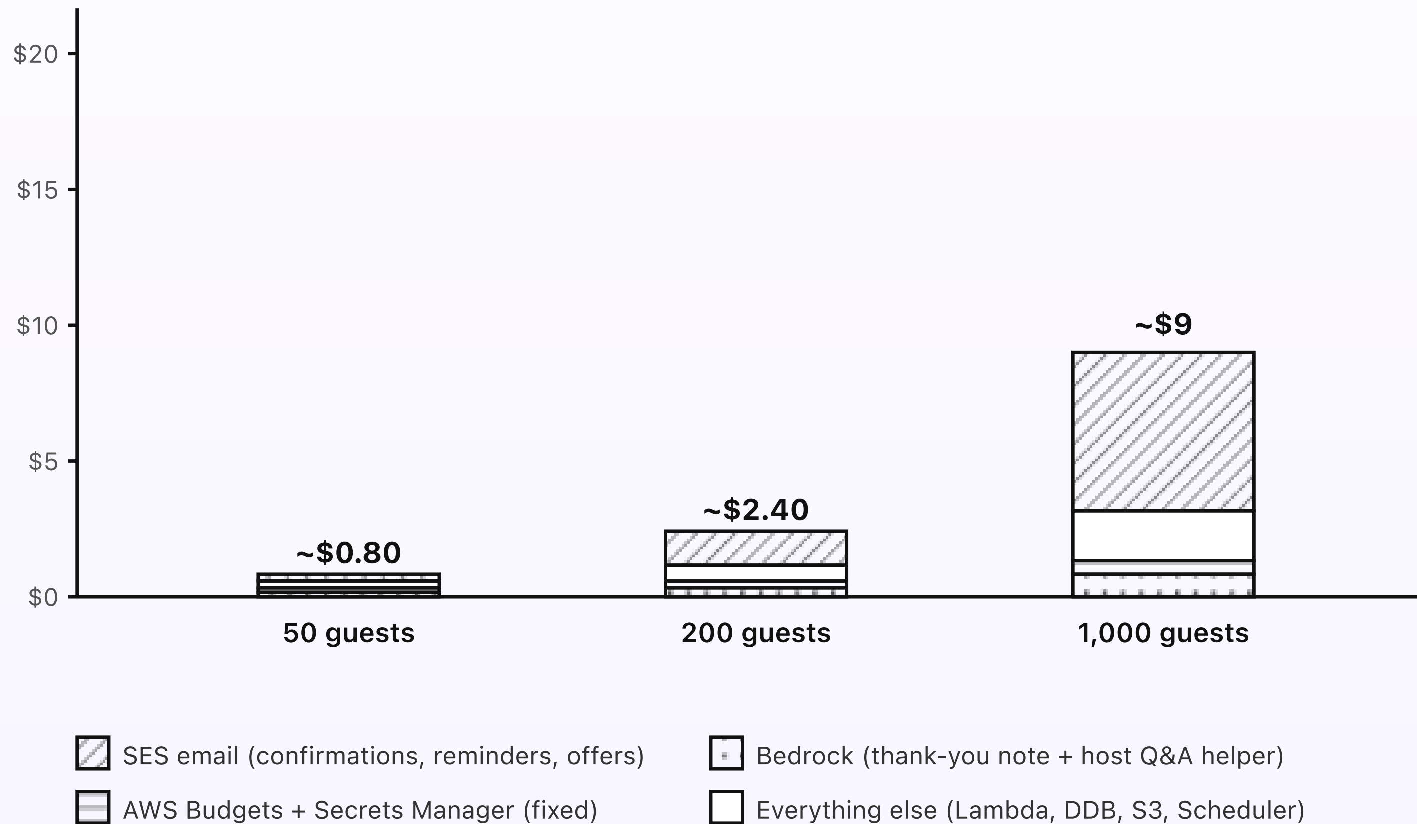
What the event RSVP manager costs

This is one of the cheapest systems in the whole series, and almost all of the bill is the emails it sends. A confirmation per guest, a couple of reminders each, the odd waitlist offer — that's the cost. The Lambda runs, the database writes, and the timed jobs are all pennies. Bedrock only fires twice in the life of an event: the post-event thank-you note and a small host Q&A helper. For a typical 200-guest event, the whole thing lands around $2.40.

KEY TAKEAWAYS

- Around $2.40 for a typical event of about 200 guests.
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- Most of the bill is SES email — $0.10 per thousand messages sent.
- Bedrock fires only on the thank-you note and the host Q&A helper — cents per event.
- A 50-guest event lands near $0.80. A 1,000-guest conference lands around $9.

Cost at three event sizes



The emails are the dominant cost — and even those are a tenth of a cent each.

Fig 6. Cost per event at three guest counts. SES email is the dominant slice because every guest gets a confirmation plus a couple of reminders. Bedrock and the fixed amounts stay tiny because they fire at most twice per event.

Where the dollars actually go

SES email (the bulk). Every guest gets one confirmation and, on the default schedule, up to three reminders. Add waitlist offers and the odd cancellation acknowledgment, and a 200-guest event sends on the order of 700 to 900 emails. At SES's $0.10 per thousand, that's under ten cents in raw send fees — but the email slice on the chart also folds in the small per-message Lambda and template-rendering work, which is why it's the dominant band. Either way, the whole email lane is a couple of dollars at most at this size.

Lambda runtime. A register click, a confirm, a cancel, a claim, each reminder fire — every one is a short Lambda invocation measured in a few hundred milliseconds. Even a 1,000-guest event is a few thousand invocations spread over weeks. Pennies a month at every size.

DynamoDB on-demand. Three small tables hold the guest list, the live count, and the audit trail. The conditional write that defends the cap is a single update per seat change. Reads are the host headcount and the duplicate checks. Pennies at any of these volumes.

EventBridge Scheduler. Each confirmed guest books up to three reminder jobs, plus a waitlist timeout per offer. A 200-guest event books a few hundred one-off jobs over its life. Scheduler is fractions of a cent per job. Negligible.

S3 + storage. The inbound email lane writes raw replies to S3; the register page assets and a small settings record round it out. A few hundred KB total. Effectively free.

Bedrock (only twice). The system uses no model on any hot path — not on register, confirm, cancel, claim, or reminders. Bedrock Haiku 4.5 fires only on the post-event thank-you note (one call that writes a warm wrap-up the host can edit and send to the confirmed list) and the host Q&A helper (a small assistant that answers the host's plain-English questions about their own guest list, like "how many seats are still open?"). A couple of cents per event.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for register, confirm, cancel, and claim.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate, no reminder poller. Each reminder is its own alarm; the system sleeps in between.
- **A database server.** DynamoDB on-demand means no provisioned capacity to pay for between events.
- **Models on the hot path.** The cap, the waitlist, and the reminders are plain code. Bedrock fires only on the thank-you note and the host Q&A helper.

How the cost scales

The bill scales with guest count, because more guests means more emails, more conditional writes, and more reminder jobs — all roughly linear. Bedrock doesn't scale with guests at all; it's a flat couple of cents whether the event has 50 people or 1,000. So a 2,000-guest event lands around $18, and a busy host running four 200-guest events a month spends under $10 across all of them. Past a few

thousand guests per event you’d batch the reminder sends rather than book one job per guest, but that’s an optimization for large conferences, not a redesign.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. A normal small-event host stays comfortably under that.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, the conditional-write capacity logic, DynamoDB schemas, and the EventBridge Scheduler config.

PART 7 OF 7

JUNE 3, 2026 PART 7 OF 7 · EVENT RSVP MANAGER SERIES ~8 MIN READ

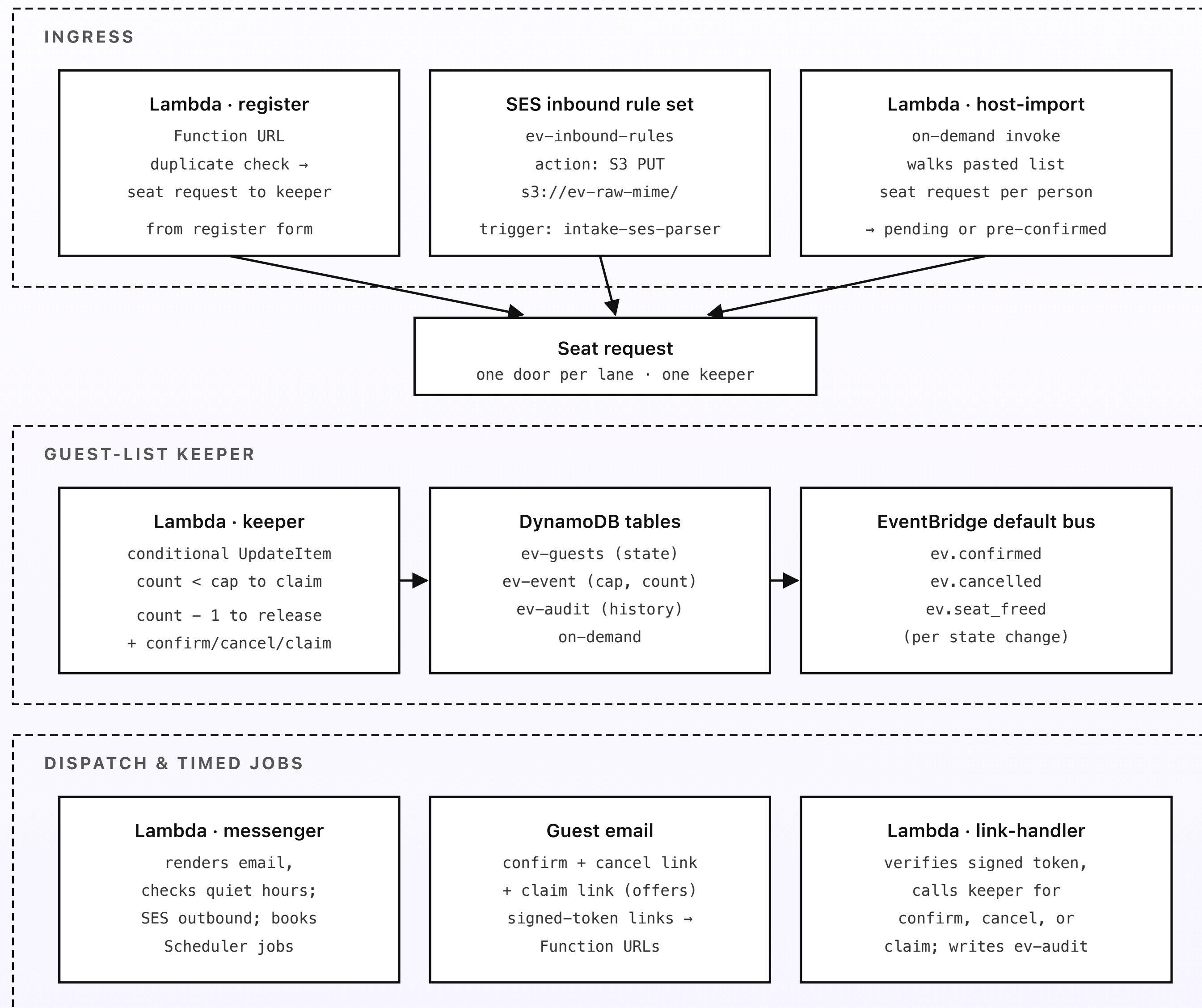
Engineering reference: the event RSVP manager architecture

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

Region and account shape

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

Topology



The cap is defended by one conditional write — and every interaction is logged to ev-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into one seat request), the guest-list keeper (conditional writes defend the cap and emit events), dispatch and timed jobs (the email ships and the signed links resolve). Every Lambda is event- or invoke-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.

- **`register`** — Lambda Function URL (`AuthType: NONE`), invoked by the static register form. Validates input, runs a duplicate check by email against `ev-guests`, and calls into the keeper for a seat request. Returns the resulting state (confirmed or waitlist position) as JSON for the form to render. Memory: 256 MB. Timeout: 15 s.
- **`intake-ses-parser`** — S3 PUT trigger on `s3://ev-raw-mime/`. Parses the MIME, reads the sender display name and address from the headers, and scans the body for a clear cancel intent ("cancel", "can't make it"). On a sign-up it runs a seat request via the keeper; on a cancel intent it routes to the keeper's release path. Sends a short SES reply with the result. Memory: 256 MB. Timeout: 30 s.
- **`host-import`** — on-demand invoke from a small host admin action. Walks a pasted list of `(name, email)` rows and runs a seat request per row, honoring a

`pre_confirmed` flag. People past the cap land on the waitlist in list order. Memory: 256 MB. Timeout: 60 s.

- **`keeper`** — the core state machine. Invoked by the three ingress lanes and by `link-handler`. Claims a seat with a single conditional `UpdateItem` on the `ev-event` item: `SET confirmed_count = confirmed_count + 1` with a condition expression `confirmed_count < cap`. Releases a seat with the mirror update guarded by `attribute_exists` and a guest-state condition so a double-click can't double-decrement. Moves a waitlisted guest to confirmed on a claim with the same capacity condition. Emits `ev.confirmed`, `ev.cancelled`, or `ev.seat_freed` to the default bus. Writes every change to `ev-audit`. *No Bedrock calls.* Memory: 256 MB. Timeout: 15 s.
- **`messenger`** — EventBridge rule on the three state events. Renders the right template (confirmation, reminder, waitlist offer, cancellation ack), checks the quiet-hours window from `ev-event.settings`, and ships via SES `SendRawEmail`. On `ev.confirmed` it books the guest's one-off reminder rules through EventBridge Scheduler; on `ev.seat_freed` it books a one-off claim-timeout rule at the end of the claim window. A quiet-hours deferral re-books the message at the next morning's business minute. Memory: 256 MB. Timeout: 30 s.
- **`link-handler`** — Lambda Function URL (`AuthType: NONE`), serving the confirm, cancel, and claim links. Verifies the HMAC-signed token on the query string (signing key in Secrets Manager under `ev/links/signing-key`), checks it hasn't expired, and calls the matching keeper path. Renders a small HTML confirmation page back to the guest. Memory: 256 MB. Timeout: 15 s.

- **`reminder-fire`** — EventBridge Scheduler target for each booked reminder one-off. Re-checks that the guest is still confirmed (a cancelled guest's jobs are deleted, but this guards a race), then invokes `messenger` to send that reminder. Self-cleans via `--action-after-completion DELETE`. Memory: 256 MB. Timeout: 15 s.
- **`offer-timeout`** — EventBridge Scheduler target for each waitlist claim window. On fire, checks whether the offer was already claimed; if not, marks it expired and re-emits `ev.seat_freed` so the offer rolls to the next waitlister. Self-cleans. Memory: 256 MB. Timeout: 15 s.
- **`host-summary`** — EventBridge Scheduler target, daily in the run-up to the event and once after it. The daily run sends the host a short headcount email (no Bedrock). The post-event run calls Bedrock Haiku 4.5 to draft a thank-you note for the confirmed list. Also backs the host Q&A helper (an on-demand invoke that answers plain-English questions about the guest list via Haiku 4.5). Memory: 512 MB.

Storage

- **DynamoDB** · **`ev-guests`** — one row per guest. PK `(event_id, guest_id)`; GSI on `(event_id, email)` for the duplicate check and on `(event_id, waitlist_pos)` for the front-of-line lookup. Attributes: `name`, `email`, `state` (pending/confirmed/waitlisted/cancelled), `reminder_jobs` (list), `offer_expires`. On-demand.
- **DynamoDB** · **`ev-event`** — one item per event holding the contended counter. PK `event_id`; attributes: `cap`, `confirmed_count`, `waitlist_len`, and a `settings` map (timezone, reminder offsets, quiet-hours window, claim-window

length). The conditional write that defends the cap targets this item. On-demand.

- **DynamoDB** · `ev-audit` — one row per state change of any kind. PK `(event_id, ts)`; attributes: `guest_id`, `action` (confirm/cancel/offer/claim/expire/import), `by`, `before`, `after`. On-demand. No TTL — this is the long-term record so the guest list is reconstructable.
- **S3** · `ev-raw-mime` — raw inbound MIME from the email reply lane. Versioning enabled. Lifecycle to Glacier at 30 days; expiry at 1 year.
- **S3** · `ev-assets` — the static register page and email template fragments. Versioning enabled.

The capacity cap, exactly

The cap is enforced by one DynamoDB `UpdateItem` against the single `ev-event` item. To claim a seat: `UpdateExpression: "SET confirmed_count = confirmed_count + :one"` with `ConditionExpression: "confirmed_count < cap"`. If the condition fails, the SDK raises `ConditionalCheckFailedException`; the keeper catches it and routes the guest to the waitlist instead of erroring. Because `UpdateItem` is atomic, a burst of concurrent claims on the last seat resolves to exactly one success and the rest fall through to the waitlist — no read-then-write gap, no optimistic-lock retry loop. Releasing a seat is the mirror: `confirmed_count = confirmed_count - :one` guarded by a guest-state condition so a double-click can't decrement twice. This single-item counter is the reason the system can promise it never oversells.

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: the post-event thank-you draft and the host Q&A helper, both in `host-summary`. Heavier reasoning isn't needed here, so Sonnet 4.6 is not wired in.
- **Embeddings.** Not used. The guest list is structured rows; deterministic lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough. No hot path calls Bedrock; the model fires at most twice per event plus the occasional host question.

EventBridge Scheduler config

- **Reminder one-offs** — `at(YYYY-MM-DDTHH:MM:SS)` in the event timezone, target `reminder-fire`, one per scheduled reminder per confirmed guest. `--action-after-completion DELETE`.
- **Offer-timeout one-offs** — `at(...)` at the end of each waitlist claim window, target `offer-timeout`. Self-deleting.
- **Quiet-hours deferrals** — created on the fly by `messenger` when a send would land in the quiet window; `at(...)` at the next business minute, target `messenger`. Self-deleting.
- **`ev-host-daily`** — `cron(0 9 * * ? *)` in the event timezone during the run-up window. Target: `host-summary`.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `rsvp.your-event.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `ev-inbound-rules`: one rule with recipient `rsvp@your-event.com` → spam scan → S3 PUT to `s3://ev-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for confirmations, reminders, and offers: verify a sender identity at `events@your-event.com` with DKIM and SPF on the parent domain. Out of sandbox by request. A configuration set with open/click tracking off keeps the emails plain and the bounce/complaint topics wired to SNS.

IAM (least privilege per Lambda)

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

- **keeper role:** `dynamodb:UpdateItem` + `GetItem` on `ev-event` and `ev-guests`; `dynamodb:Query` on the GSIs; `dynamodb:PutItem` on `ev-audit`; `events:PutEvents` on the default bus. *No* `bedrock:*`, *no* `ses:*`.
- **messenger role:** `ses:SendRawEmail` from the verified identity; `scheduler:CreateSchedule` + `DeleteSchedule` for reminder and timeout one-offs; `dynamodb:GetItem` on `ev-event` + `ev-guests` for templating; `secretsmanager:GetSecretValue` on the link signing key (to mint signed links).
- **link-handler role:** `secretsmanager:GetSecretValue` on `ev/links/signing-key`; `lambda:InvokeFunction` on `keeper` (or direct DDB if collapsed); `dynamodb:PutItem` on `ev-audit`.

- **intake-ses-parser role:** `s3:GetObject` on `ev-raw-mime`; `lambda:InvokeFunction` on `keeper`; `ses:SendRawEmail` for the reply.
- **host-summary role:** `dynamodb:Query` on `ev-guests` + `ev-audit`; `bedrock:InvokeModel` on the Haiku ARN; `ses:SendRawEmail` from the verified identity.

Signed-link flow

Every confirm, cancel, and claim link is a Function URL with a query string carrying `event_id`, `guest_id`, an `action`, an `expiry` timestamp, and an HMAC-SHA256 signature over those fields keyed by `ev/links/signing-key`. `link-handler` recomputes the HMAC and rejects any mismatch or past-expiry link before touching state. This gives passwordless, per-seat authorization: a guest can act only on their own seat, and only until the event passes. Claim links additionally carry an `offer_id` so a stale offer link (rolled on to the next person) is recognized and rejected with a friendly “this offer has expired” page.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"ConditionalCheckFailed"` + `"timeout"` to a metric for alerting (note: a `ConditionalCheckFailed` on a claim is normal — a full event — so alert only on unexpected rates).
- **Alarms:** keeper failures > 0 excluding capacity-condition failures; messenger send failure rate > 1% in 24h; link-handler signature-verification failures > 5/hour (might mean a tampered link campaign or a rotated key).

- **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 `ev-cost-alarm` subscribed to the host admin's email.

Config and secrets

The link signing key lives in Secrets Manager under `ev/links/signing-key`; the SES sender identity lives in IAM and the verified-domain config. Per-event settings — cap, timezone, reminder offsets, quiet-hours window, claim-window length, and the time-critical-offer override — live in the `settings` map on the `ev-event` item so the host can change them without a deploy. Cross-event defaults live in Parameter Store under `/ev/config/`. Lambdas fetch config on cold start and cache it for the lifetime of the execution environment.

Deploy

Deploy with GitHub Actions + OIDC + AWS SAM — no long-lived keys. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), keep the cap and counter on a single `ev-event` item so the conditional write stays atomic (don't shard the counter unless you actually outgrow a single partition), and pin the EventBridge Scheduler timezone to the event timezone so reminders don't silently start firing in UTC after a CI change. Total deployable surface: around nine Lambdas, three DDB tables (plus two GSIs), two S3 buckets, one EventBridge rule on the default bus (plus the on-the-fly Scheduler one-offs), 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 event, see **Work with me**.