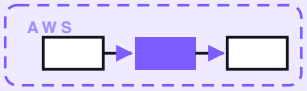
7-PART SERIES · FREE COMPANION



Appointment reminder

A serverless reminder that cuts no-shows. It watches every upcoming appointment, sends each customer the right reminder at the right time, and lets them confirm, reschedule, or cancel with one tap; then it tells your staff who confirmed and who went quiet so gaps get filled early. Nothing pushy; quiet hours respected. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Appointment reminder

PART 1 OF 7

MAY 7, 2026 PART 1 OF 7 · APPOINTMENT REMINDER SERIES ~5 MIN READ

An appointment reminder on AWS for a few dollars a month

Every business that books time loses money to no-shows. The customer who booked a Tuesday cleaning three weeks ago and forgot. The new client who double-booked and isn't sure which one they meant to keep. The patient who meant to call and reschedule but never got around to it, so the chair sits empty during a slot you could have sold. This post walks through the design of a small reminder that watches every upcoming appointment, nudges each customer at the right time, lets them confirm or reschedule or cancel with one tap, and tells your front desk who went quiet so the gap can be filled early.

KEY TAKEAWAYS

- Three sources for booked appointments: a Drive list, an inbox forwarding lane, and a calendar import lane.
- Every appointment ends in one of four moves on each tick: scheduled, first reminder, second reminder, or gap alert.
- Per-service rules: a dental visit might use a 48-hour and 2-hour reminder, a salon 24-hour and 1-hour.
- Reminders go by text first, email as fallback, and respect quiet hours and your holiday calendar.
- Designed on AWS for about $3/month at typical small-business volume.

The whole system on one page

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

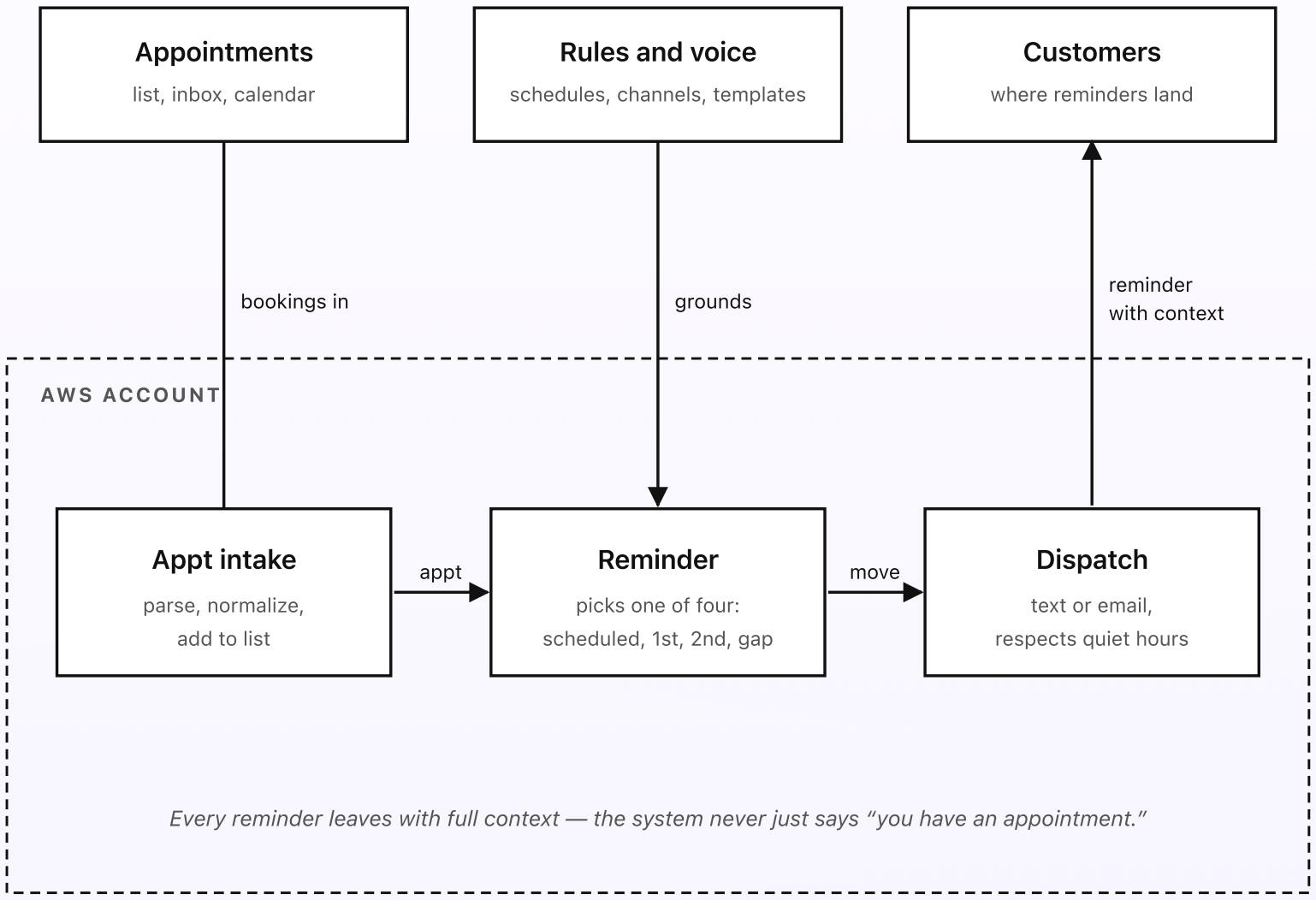

Appointments
list, inbox, calendar
Rules and voice
schedules, channels, templates
Customers
where reminders land
bookings in
grounds
reminder
with context
AWS ACCOUNT
Appt intake
parse, normalize,
add to list
appt
Reminder
picks one of four:
scheduled, 1st, 2nd, gap
move
Dispatch
text or email,
respects quiet hours
Every reminder leaves with full context — the system never just says “you have an appointment.”

Fig 1. Three sources outside, three pieces inside AWS. Bookings flow in from a Drive list, an inbox forwarding lane, and a calendar import lane. The Reminder runs hourly and picks one of four moves. Dispatch sends the right reminder to the right person at the right time.

What you set up once (the outside)

- **Appointments.** A Google Sheet in a Drive folder, one row per booking: customer name, phone, email, service (cleaning, consult, haircut, fitting, follow-up), staff member, date and time, status, and a link to the booking. Most appointments arrive here straight from your booking tool; new ones can also enter via two other lanes covered in Part 2 — an inbox-forwarding lane (forward a booking-confirmation email to a dedicated address and the reminder proposes a row for one-tap approval) and a calendar import lane (events tagged in Google Calendar with `#appt` get pulled in automatically).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the reminder schedule for each service — how many hours ahead the reminder should send, and how many times. A dental visit might get a 48-hour and 2-hour reminder; a salon appointment 24-hour and 1-hour; a longer consult 72-hour, 24-hour, and 3-hour. The doc also lists the channel preference per customer (text or email), the quiet hours, the staff member who handles each service, and any holiday calendars to skip. The *voice* doc holds one reminder message template per service — what the text or email actually says.
- **Customers.** The people with upcoming appointments. Each row has a phone number (so the reminder is a text) or, if no phone is set, an email address. Reminders land with the service, the date and time, the staff member, a link to

the booking, and three one-tap links — “Confirm,” “Reschedule,” and “Cancel” — that stop further reminders or update the booking.

What runs on every tick (the inside)

- **The appointment intake.** Three sources feed the list. The Drive sheet itself is the canonical store. New bookings can also be added via the inbox forwarding lane (forward a confirmation email to `bookings@your-company.com`, the reminder uses Textract to read it and Bedrock Haiku 4.5 to extract customer, service, date, and time, then drops a one-tap approval card in the front-desk Slack to confirm before the row is added) and the calendar import lane (events tagged `#appt` get pulled hourly by a small sync Lambda).
- **The reminder.** Runs once an hour. Reads the list. For each upcoming appointment, computes hours-to-appointment. Compares against the per-service reminder schedule in the rules doc. Picks one of four moves. *Scheduled*: too far out for any reminder window — do nothing. *First reminder*: just crossed the first window — text the customer with full context. *Second reminder*: crossed a later window with no confirm — re-send, closer to the time. *Gap alert*: the customer cancelled, or the appointment is hours away and still unconfirmed — tell staff so the slot can be filled. The reminder itself doesn’t call a model on the hourly tick — the move logic is plain Python.
- **Dispatch.** Reads the voice doc, formats the reminder for the chosen move and service, and sends it. Text messages go through SNS. Email goes through SES outbound. Both honor quiet hours (no reminders between 9pm and 8am local by default) and the holiday calendar (no reminders on configured days). Every dispatch writes a row in DynamoDB so the next tick can tell whether the customer replied. A weekly staff summary writes a short paragraph:

appointments confirmed, customers who went quiet, and the slots that opened up.

In plain words

Mr. Tan booked a 9am dental cleaning three weeks ago. The rules doc says dental cleanings get a 48-hour text and a 2-hour text. Two days out, at a sensible hour, the reminder texts him: “Reminder: cleaning with Dr. Lim, Thursday 9:00am. Reply to *confirm*, *reschedule*, or *cancel*.” Mr. Tan taps Confirm. The reminder marks the appointment confirmed and stops — he won’t get the 2-hour text, because he already said he’s coming. If instead he taps Reschedule, a short page lets him pick a new time; the list updates and a fresh reminder chain starts against the new slot. If he taps Cancel, the slot is freed and the front desk gets a heads-up so they can offer it to someone on the waitlist. And if he simply goes quiet — no tap at all — the 2-hour reminder still goes out, and if he’s still silent the morning of, staff see him flagged on the daily list as “unconfirmed.”

The cost of running this is about $3 a month at SMB volume. The cost of *not* running it is the empty chair on a Thursday morning — the slot you could have sold twice over if you’d known the night before that it was opening up.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every reminder ships with full context — service, time, staff member, link to the booking. The customer never has to dig.
- Four moves, always. Scheduled, first reminder, second reminder, gap alert. There is no fifth.
- Quiet hours and holidays are respected. A 6am text is worse than no text; bad timing burns goodwill.
- Confirm stops further reminders for that appointment. Reschedule updates the list and resets the chain.
- The list lives in Drive. Adding a booking, changing a time, or fixing a phone number doesn't need a deploy.
- Every dispatch and reply is logged. Look back at a no-show next month and you can see every reminder that went out.

Why this shape

Most businesses handle reminders one of three ways: the front desk calls everyone the day before, the booking tool fires a single generic reminder, or nobody reminds anyone and you eat the no-shows. The phone calls work until the desk is busy — which is exactly when the no-shows pile up. The single generic reminder is better, but it has no memory: it can't tell a customer who already confirmed from one who hasn't, so it either over-sends or under-sends. And no reminders at all is just paying for empty chairs.

The setup above keeps the booking list in a doc the team already edits, but adds a small system that *looks at* that list every hour and acts only when something needs acting on. Reminders come at the right time for the service. They carry enough context that the customer doesn't have to call to ask what or when. They let the customer reply in one tap. And they stop the moment somebody confirms. The reminder is invisible most of the time; visible only when it actually saves you a no-show.

The next four posts walk through each piece in turn: how an appointment gets on the list, how the reminder knows when to send, how a reminder reaches the customer, and how a customer confirms, reschedules, or cancels. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 7, 2026 PART 2 OF 7 · APPOINTMENT REMINDER SERIES ~4 MIN READ

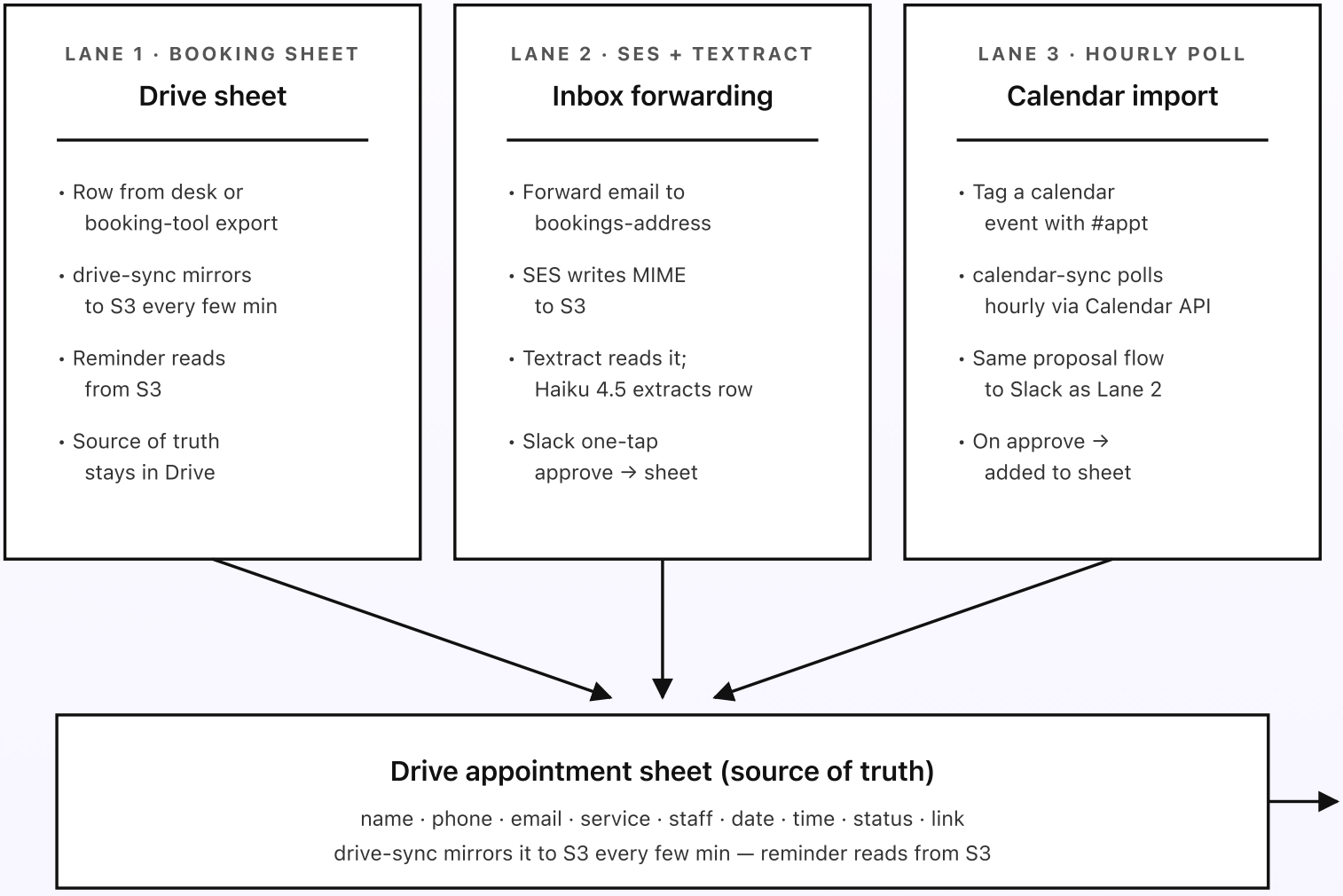
How an appointment gets on the list

The reminder only reminds about what's on the list. So the first job is making sure the list actually reflects what your business has booked. There are three ways an appointment gets on: it lands in the Drive sheet from your booking tool or by hand, somebody forwards a booking-confirmation email to a dedicated address, or somebody puts the appointment on a Google Calendar with a small tag. The first one is obvious. The other two exist because in real life bookings come in by email and on calendars, not as tidy spreadsheet rows.

KEY TAKEAWAYS

- Three intake lanes feed one list: the Drive sheet, an inbox-forwarding lane, and a calendar import.
- Inbound booking emails are parsed by Textract; Bedrock Haiku 4.5 reads the text and proposes a row.
- Every parsed row goes to the front-desk Slack for one-tap approval before it lands on the list.
- Calendar events tagged `#appt` get pulled hourly via the Google Calendar API.
- The list stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one list



The Drive sheet stays the source of truth — the other lanes are conveniences that propose rows for it.

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the inbox lane and the calendar lane are conveniences that propose rows for human approval. The drive-sync Lambda mirrors the sheet to S3 so the reminder can read it without hitting Drive on every tick.

Lane 1: the Drive sheet itself

The simplest lane. The booking lands as a row in the appointment sheet — either typed by the front desk, or pushed in by an export from your booking tool. The columns are short: customer name, phone, email, service, staff member, date, time, status, and a link to the booking. A small Lambda — `drive-sync` — runs every few minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://ar-appointments-source/appointments.csv` if the sheet has changed since the last sync. The reminder reads from S3, not Drive directly. That keeps Drive API calls predictable and gives you S3 versioning for free, so a bad bulk-edit can be rolled back in one click.

This lane covers the bulk of bookings: most businesses already have a tool that knows the appointment, and getting its rows into one sheet is the smallest possible integration. Everything the reminder needs lives in those few columns.

Lane 2: inbox forwarding (for the one-off bookings)

Set up a dedicated inbound address — something like `bookings@your-company.com` — via Amazon SES. Anyone on the team forwards a booking-confirmation email to that address and the reminder takes it from there. SES writes the raw MIME to `s3://ar-raw-mime/`. The S3 PUT triggers a parser Lambda. The Lambda walks the MIME tree to the body text and any attachment,

runs Amazon Textract if there's a PDF or image (Textract reads PDF, PNG, JPEG, and TIFF natively), and gets back the extracted text plus any tables.

Then a Bedrock Haiku 4.5 call reads the text and emits a structured row: customer name, phone, email, service, staff member (if named), and the date and time. The model prompt is short: "Extract a row for the appointment list. Return JSON only. Mark each field with a confidence score. Do not invent a time that isn't in the text." The output goes to a small Slack interactive message that pings the front desk: the proposed row, the confidence per field, and three buttons — *approve*, *edit*, *discard*. On *approve*, a Lambda writes the row to the Drive sheet via the Sheets API. On *edit*, the desk gets a fillable modal pre-populated with the proposal. On *discard*, the message is logged and the email moved to a discarded prefix in S3 for audit.

The reason every parsed row goes to a human first is simple: a wrong appointment time the model misread is worse than a booking that never made it onto the list. The misread one will text the customer to show up at the wrong hour — which annoys them and burns your goodwill.

Lane 3: calendar import

Some teams already keep appointments on a shared calendar. The mobile groomer runs her whole week off a Google Calendar. The physiotherapist blocks slots on a clinic calendar. The tutor keeps sessions on a personal one. Forcing those teams to also type rows in a sheet is a fight you don't need to have on day one.

Lane 3 picks up calendar events tagged with `#appt` in the description. A small `calendar-sync` Lambda runs hourly, iterates through the configured Google

Calendars (using a service-account credential stored in Secrets Manager), and pulls any tagged events whose start time is in the future. Each pulled event becomes a proposal in the same Slack flow as Lane 2 — one-tap approve to add to the list. Once approved, the calendar event can stay where it is; the list now owns the reminder for it.

Calendar import is the most opt-in of the three lanes. A team that doesn't use it loses nothing; a team that does avoids retyping appointments they already put on a calendar.

Why the list stays the source of truth

Three lanes in, but only one place where the reminder actually looks. That's a deliberate constraint. If two lanes both wrote directly to the reminder's state, every "why did this text go out?" question would mean checking three places. Funneling everything through the Drive sheet means there is exactly one row per appointment, and any staff member can read or edit any of it without learning a new tool. The convenience lanes are first-class for getting bookings in, but they always pass through the sheet on the way.

Next post: how the reminder actually reads the list, computes hours-to-appointment, and picks one of four moves.

PART 3 OF 7

MAY 7, 2026 PART 3 OF 7 · APPOINTMENT REMINDER SERIES ~5 MIN READ

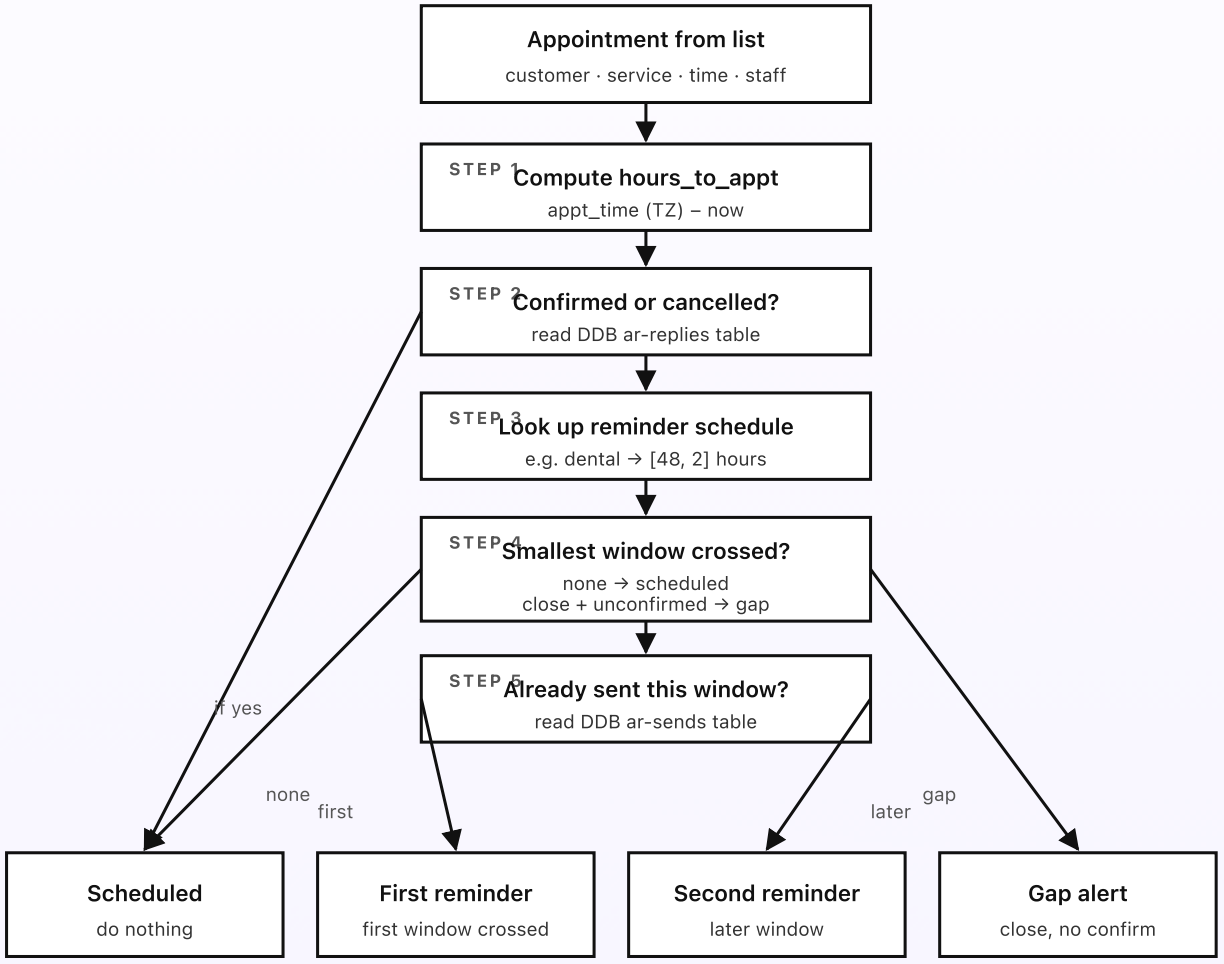
How the reminder knows when to send

Once an hour, an EventBridge Scheduler rule fires the reminder Lambda. The Lambda reads the list, looks at one appointment at a time, computes the hours remaining, and decides whether to do nothing or to send a reminder — and if so, which kind. The whole decision is plain Python. No model. No vector retrieval. Every threshold lives in the rules doc, where the front desk can edit it without a deploy.

KEY TAKEAWAYS

- The reminder runs once an hour via EventBridge Scheduler so it can catch tight 2-hour and 1-hour windows.
- Per-service schedules live in the rules doc — a dental visit gets 48h/2h, a salon gets 24h/1h, a long consult gets 72h/24h/3h.
- Four moves per appointment, every tick: scheduled, first reminder, second reminder, gap alert.
- DynamoDB tracks last-send and confirm per appointment so reminders aren't duplicate spam.
- The reminder itself never calls a model. The decision is entirely deterministic.

The decision flow, per appointment



The rules doc holds every threshold — change a window and the next hourly tick uses it.

Fig 3. The reminder's decision tree, per appointment, per hourly tick. Five steps decide which of four moves applies. The rules doc holds every threshold; the reminder only enforces them.

Reminder schedules: 48/2 isn't magic, it's in the doc

The rules doc has one short section per service. Each section names the schedule in plain prose: "Dental cleaning: remind at 48 and 2 hours. Salon haircut: 24 and 1 hour. New-patient consult: 72, 24, and 3 hours. Quick follow-up: 24 hours only." The numbers are hours remaining when the reminder fires. The first number is the early heads-up. The last number is the same-day nudge — the "see you soon" text that catches the customer who forgot.

The schedules exist for a reason. A 48-hour dental reminder gives the customer time to reschedule if the day no longer works, which turns a no-show into a filled slot. A 1-hour salon nudge is the last-chance reminder for a walk-in-style appointment people book on impulse and forget. A 72-hour consult reminder respects that a longer, more valuable appointment deserves more notice. Different services have different rhythms; the schedules reflect that.

Per-appointment overrides exist too. The list sheet has an optional column called `schedule_override`. Type a comma-separated list of hours there and the reminder uses your numbers instead of the service default for that one row. This is the right escape hatch for the VIP client who asked to be reminded the night before, or the first-timer you want to nudge twice.

Four moves, always

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

- **Scheduled.** The appointment is more than the first window away, or it's already confirmed or cancelled. Do nothing. Most appointments, most hours, are scheduled.
- **First reminder.** The appointment just crossed the first window threshold and there's no reply yet. Send a fresh reminder with full context. Write a row to the `ar-sends` DynamoDB table marking that the first window has fired.
- **Second reminder.** A later window crossed without a confirm. Send a closer-to-the-time nudge that's shorter and friendlier — "See you at 9 this morning." Write the new send to `ar-sends`.
- **Gap alert.** The customer cancelled, or the appointment is only an hour or two away and still completely unconfirmed. Tell the front desk, in their Slack, so they can call the customer or offer the slot to someone on the waitlist. A gap alert goes to staff, not the customer — it's the system's way of saying "this chair might be empty; do something about it now."

State that makes the decision deterministic

The reminder reads two DynamoDB tables every tick. `ar-sends` records every reminder that's gone out: `(appt_id, window_index, sent_at, channel)`. `ar-replies` records every reply: `(appt_id, replied_at, action)`. With those two tables, the move-decision logic is a few dozen lines of Python and zero magic. A given appointment with a given time, a given reminder schedule, and a given

send/reply history always produces the same move. Re-running the tick produces no extra texts (because the state in DDB shows what already fired).

Rescheduling an appointment is an explicit reset of both tables for that booking: rows for the old time are kept for audit, but a new chain starts fresh against the new appointment time. Part 5 covers the reschedule flow in detail.

Why the hourly tick uses no model

The reminder could call a model on the tick to write a smarter message, or to decide whether to send at all. It doesn't. Two reasons. First, the tick should be the one part of the system that is utterly predictable — if the rules doc says remind at 48 hours and there's no confirm, the text fires. A model in that loop introduces variance the front desk can't reason about. Second, model calls cost money, and most hours most appointments are scheduled, so the call would be wasted again and again.

Bedrock fires elsewhere — on the inbound parsing lane in Part 2, and on the weekly staff summary mentioned in Part 6. Not on the hourly tick. The reminder itself is plain Python that reads a doc and writes events.

Next post: how a reminder reaches the customer, how quiet hours and holidays are honored, and what a confirm actually does to the chain.

PART 4 OF 7

MAY 7, 2026 PART 4 OF 7 · APPOINTMENT REMINDER SERIES ~5 MIN READ

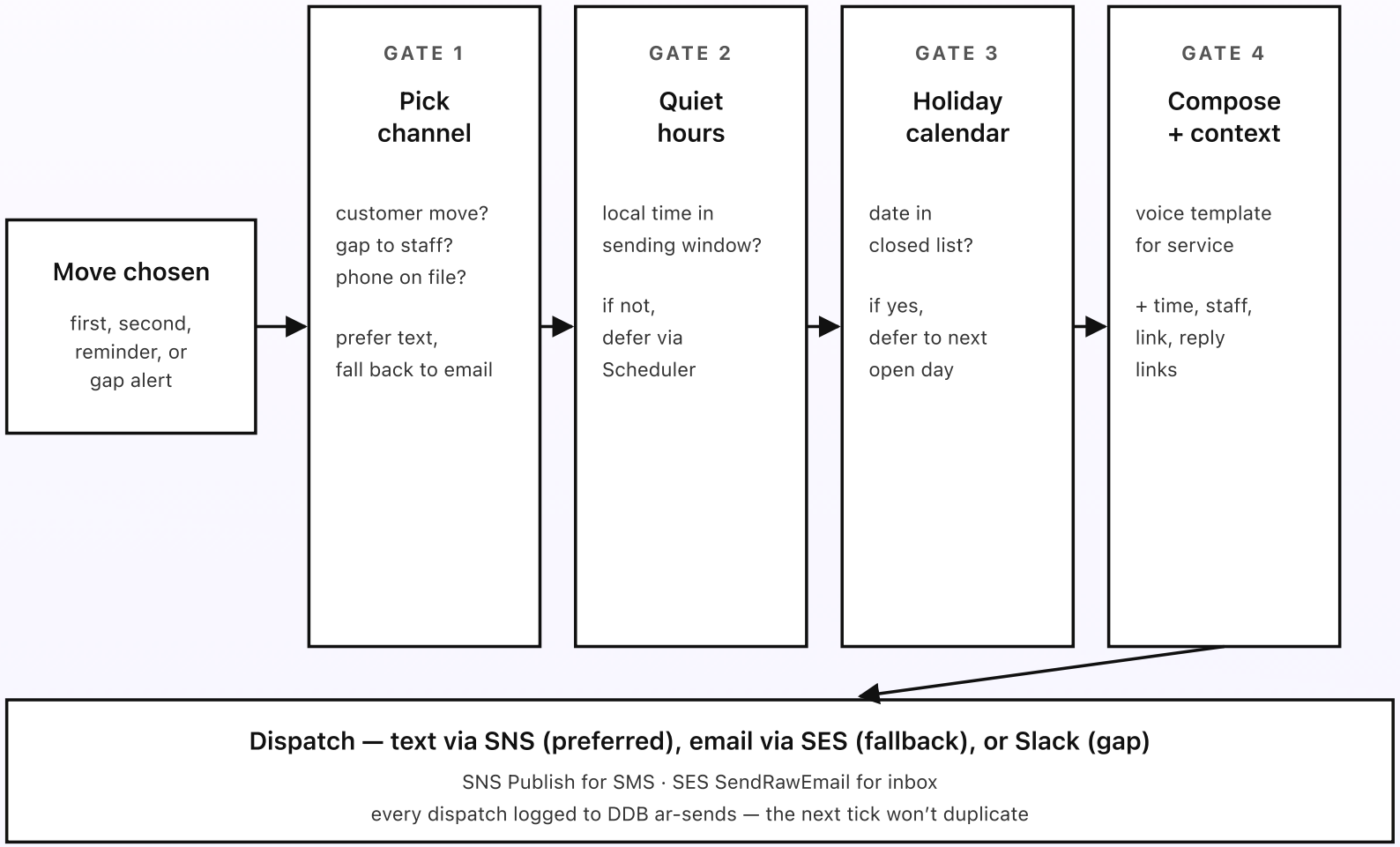
How a reminder reaches the customer

The reminder picked a move — first reminder, second reminder, or gap alert. Now the dispatch Lambda has to figure out who to send it to, on what channel, at what time of day, and with what context attached. Get any of those wrong and the reminder is worse than no reminder: a 6am text, a generic “you have an appointment,” a message to a number the customer changed a year ago. Four small guardrails sit between the move and the actual send.

KEY TAKEAWAYS

- Channel choice: text if a phone is on file, email if not; a gap alert goes to staff in Slack instead.
- Text is the default for customers; email is the fallback when no phone is set.
- Quiet hours and holiday calendars defer reminders to the next sensible hour.
- Every reminder ships with the service, date, time, staff member, link, and one-tap reply links.
- A gap alert is the one move that goes to staff, not the customer — so an opening gets filled.

Four guardrails on every dispatch



Every gate is a deterministic check — no model calls, no surprise 6am text on a holiday.

Fig 4. Four guardrails between the move and the dispatched reminder. Pick the channel. Honor quiet hours. Skip closed days. Compose with full context. Then ship via text, email, or Slack and log the dispatch so the next tick doesn't duplicate.

Gate 1: pick the channel

The first thing the dispatch Lambda decides is who the message is even for. A first or second reminder is for the customer. A gap alert is for the front desk. Those go to completely different places, so the channel choice forks here.

For a customer reminder, the dispatch looks up the row's phone number. If one is set, the reminder is a text — texts get read within minutes and feel personal, which is what you want for an appointment nudge. If no phone is on file, it falls back to email so nobody slips through. For a gap alert, the message goes to a configured Slack channel where staff watch the day's bookings; it never goes to the customer, because a gap alert is an internal "this slot might open" signal, not something the customer should see.

Gate 2: quiet hours

The reminder runs hourly, so a window can come due at any hour — including the middle of the night. A 48-hour window on a 7am appointment comes due at 7am two days before, which is fine. But a 2-hour window on a 10pm appointment would come due at 8pm, and a tight schedule could line up a send for after the customer's bedtime.

Gate 2 reads the rules doc's quiet-hours setting (default 9pm to 8am, configurable per business). If the current local time is in the quiet window, the dispatch creates a one-off EventBridge Scheduler rule that fires at the next allowed minute and exits without sending. The Scheduler invokes the same dispatch Lambda with the same payload at the deferred time, where Gate 2 will let it through. A reminder that can't be sent before the appointment itself (because the appointment is inside quiet hours) is dropped and logged rather than waking the customer.

Gate 3: holiday calendar

The rules doc lists the days you're closed — either a static list ("public holidays, the week between Christmas and New Year…") or a reference to a Google Calendar that holds them. Gate 3 checks the current local date against that list and, if it's a closed day, defers the reminder to the next open day.

The list is on purpose — the reminder won't auto-detect a country's public holidays for you. The failure modes are very different. A closed day you forgot to add sends a reminder for an appointment that can't happen. A day in the list that you're actually open just delays a reminder by a day, which is fine. The trade-off favors keeping the list explicit.

Gate 4: compose with full context, then ship

The voice doc has one message template per service: a short message with placeholders for the service, the date and time, the staff member, and a link to the booking. The dispatch Lambda fills the placeholders, adds three short one-tap

links — Confirm, Reschedule, Cancel — and ships the message via SNS for a text or SES for an email. The reply links point at a Function URL that records the customer's choice; Part 5 covers what each one does.

Text messages are short by nature, so the template is tight: "Hi {name}, reminder: {service} with {staff}, {day} {time}. Confirm: {link} · Reschedule: {link} · Cancel: {link}." Email gives a little more room for the booking details and a clearer set of buttons, but carries the same fields and the same three links.

A gap-alert move composes differently: it's a staff message, so it names the customer, the service, the time, and how long until the appointment, plus a note on why it fired ("cancelled" or "2 hours out, still unconfirmed"). The point is to hand the front desk everything they need to decide whether to call the customer or release the slot.

Every dispatch — text, email, or gap alert — writes a row to `ar-sends` in DynamoDB. The next tick reads that row and knows not to send the same window again.

Why the guardrails exist

None of these gates are exotic. They're the kind of small care a thoughtful receptionist would take if they were sending the reminders by hand — text the people who text, don't message at 6am, skip the days you're closed, include enough detail that the customer doesn't have to call back to ask what time. Putting them in code as four small sequential gates makes them part of the design, not a feature you're trusting the writer of any one message to remember.

Next post: how a customer confirms, reschedules, or cancels once a reminder lands — how the list updates, how the chain resets, and how the front desk sees the change.

PART 5 OF 7

MAY 7, 2026 PART 5 OF 7 · APPOINTMENT REMINDER SERIES ~5 MIN READ

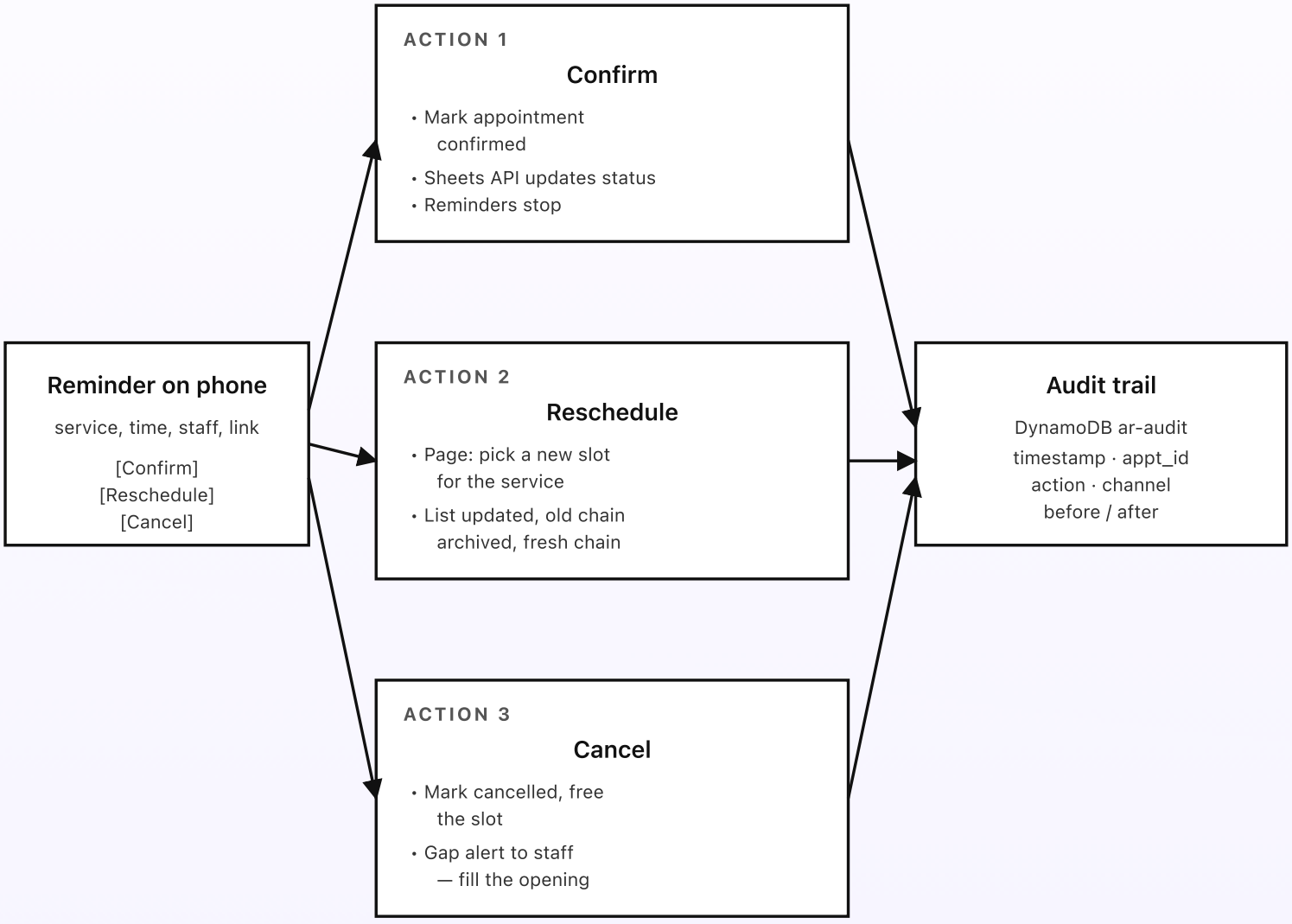
How a customer confirms or reschedules

A reminder text lands on Mr. Tan's phone at 9am, two days before his cleaning. There are three links: Confirm, Reschedule, Cancel. What happens when he taps one? The honest answer is "it depends on which one." This post walks through the three things the reminder can do on a reply — confirm, reschedule, cancel — and how the list, the chain state, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Three actions per reply: *confirm* (stop the reminders), *reschedule* (new time, fresh chain), *cancel* (free the slot).
- Each action updates the list sheet via the Sheets API and writes an audit row.
- A reschedule archives the old chain and starts a clean one against the new time.
- Cancel frees the slot and pings the front desk so the opening can be filled.
- The reply links are short Function URL pages backed by a Lambda — no app to install.

Three actions on a reply



Confirm stops the noise; Reschedule moves the booking; Cancel turns an empty chair into a slot to fill.

Fig 5. Three actions per reply, three different effects. Confirm stops the reminders. Reschedule updates the time and starts a fresh chain. Cancel frees the slot and alerts staff. Every action writes to the audit trail.

Action 1: confirm (the most common)

Mr. Tan taps *Confirm*. The link opens a tiny page — nothing to install, no login — that says "Thanks, you're confirmed for Thursday 9:00am." Behind it, a Function URL Lambda does three small things, in order. First, it writes a `confirmed` row to the `ar-replies` DynamoDB table for this appointment. Second, it updates the `status` column in the list sheet via the Sheets API so the front desk sees a green check next to his name. Third, it writes a `confirmed` row to `ar-audit` with the time, the appointment, and the channel he replied on.

From that point on, the "already confirmed?" check from Part 3 short-circuits the appointment to *scheduled* on every tick. Mr. Tan won't get the 2-hour text, because he already said he's coming. The system only nudges people who haven't replied — confirming is how you opt out of the rest of the chain.

Action 2: reschedule (the save)

Sometimes the day no longer works. Mr. Tan has a meeting that ran long, or a school pickup landed on the same hour. Without a reminder, he'd just not show up. With one, he taps *Reschedule* two days ahead — while there's still time to fill his old slot.

Reschedule opens a short page showing the open slots for his service over the next couple of weeks (read from the same list, filtered to free times). He picks one. On submit, the Function URL Lambda updates the list sheet with the new date and time, copies the old appointment's chain rows from `ar-sends` to `ar-sends-archive` with a chain id, and clears the live chain so a fresh set of reminders starts against the new time. It writes a `rescheduled` row to `ar-audit` with the old time and the new time. And it frees his original slot, so a gap alert can offer it to someone else.

The next tick reads the new time, sees it's more than the first window away, and lands at *scheduled*. The reminder chain begins again against the new appointment — same service, same schedule, clean state.

Action 3: cancel (the honest no)

Sometimes the customer just can't make it and isn't ready to pick a new time. That's fine — a clean cancel two days out is far better than a no-show, because it gives you the slot back while it's still sellable.

Cancel opens a one-line confirmation page. On confirm, the Function URL Lambda marks the appointment `cancelled` in the list, frees the slot, and posts a gap alert to the front-desk Slack: "Cancelled — Mr. Tan, cleaning, Thursday 9:00am. Slot now open." A `cancelled` row goes to `ar-audit`. The reminders stop, of course — there's nothing left to remind about. The waitlist, if you keep one, is where the front desk goes next.

If a cancelled customer later books again through any of the three intake lanes from Part 2, that's simply a new appointment with a new row and a new chain. The

cancelled one stays in the list, marked, so the history is intact.

Every action is logged, every action is reversible

The `ar-audit` table records every confirm, reschedule, and cancel with the time, the appointment, the channel, and a snapshot of the row before and after. If a customer rescheduled to the wrong slot, or tapped Cancel by mistake, the front desk can run an “undo last action” through a small admin command that reads the previous-state snapshot and restores the row. The undo is itself an audit row, so the trail of edits stays clean.

This kind of reversibility matters because the reply links are out in the world, on customers’ phones, tapped by people in a hurry. A misclick shouldn’t cost anyone an appointment. The audit trail is what lets the desk fix a wrong tap in seconds, and it’s the same trail the weekly summary in Part 6 reads to tell you how many no-shows you actually prevented.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the text messages are the only line that really moves.

PART 6 OF 7

MAY 7, 2026 PART 6 OF 7 · APPOINTMENT REMINDER SERIES ~3 MIN READ

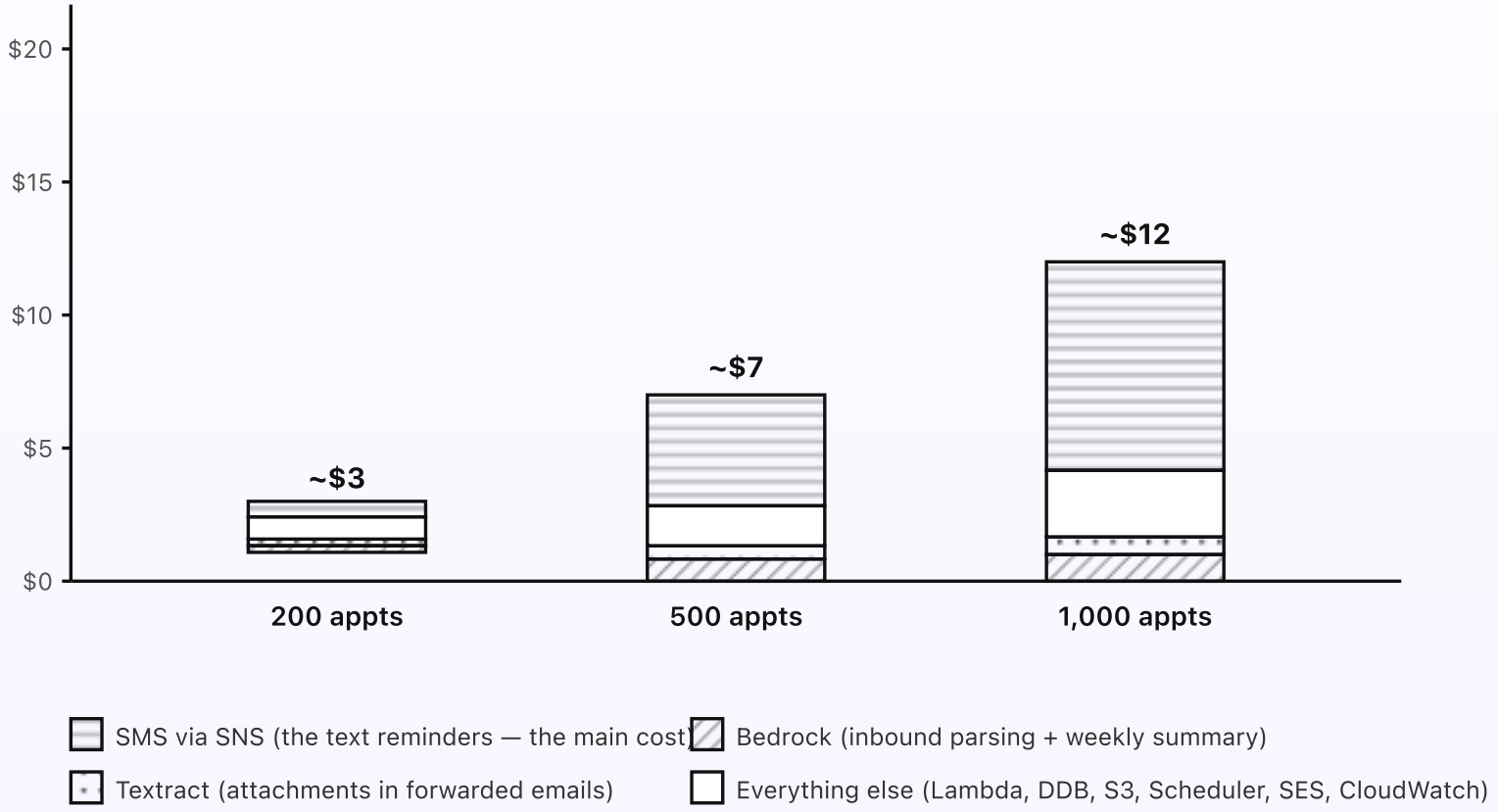
What the appointment reminder costs

The reminder is one of the cheapest systems in this whole series — with one twist. The hourly tick reads a CSV from S3, does some time arithmetic, writes a few rows to DynamoDB, and sends a handful of messages. It calls no models on the tick. The one line that actually moves the bill is the text messages themselves: each SMS carries a small per-message fee. At typical SMB volume the total is a few dollars a month, and most of that few dollars is texts.

KEY TAKEAWAYS

- Around $3/month at typical SMB volume (about 200 appointments a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The dominant cost is text messages — a small per-message fee, a few per appointment.
- Bedrock fires only on inbound email parsing (a few times a month) and the weekly summary.
- At 500 appointments the bill is around $7. At 1,000 it's around $12 — mostly texts.

Cost at three volumes



The text messages are the dominant cost — everything else is fractions of a cent per appointment.

Fig 6. Monthly cost at three appointment volumes. SMS is the dominant slice because every reminder is a paid text. Bedrock and Textract are small because they only fire on the inbound parsing lane and the weekly summary. The everything-else bucket — Lambda, DynamoDB, S3, Scheduler — stays tiny.

Where the dollars actually go

SMS via SNS (the bulk). This is the line that matters. Every reminder sent as a text carries a small per-message fee, and the fee varies by country. A typical appointment gets one or two reminders, plus the odd reschedule confirmation. At 200 appointments a month that's a few hundred messages — a couple of dollars. At 1,000 appointments it's a few thousand messages and the SMS line is most of the bill. Customers who only have email on file cost nothing extra to remind, because email is effectively free; that's one reason the email fallback is worth keeping.

Lambda runtime. The reminder runs once an hour. Each tick reads the list CSV from S3, iterates the rows, computes `hours_to_appt` for each, and decides on a move. At 200 appointments that's a few hundred milliseconds; at 1,000 it's under a second. Add the dispatch Lambda firing per send, the Function URL Lambda for replies, the calendar-sync Lambda hourly, and the drive-sync Lambda every few minutes — the Lambda total still lands under a dollar at all three volumes.

DynamoDB on-demand. Three small tables: `ar-sends`, `ar-replies`, `ar-audit`. Reads are dominant during the hourly tick (one read per appointment per tick). Writes are sends, replies, and audit rows. Pennies a month at any of these volumes.

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

EventBridge Scheduler. The hourly tick rule plus deferred-send rules from quiet-hours and holiday gates. A handful of invocations a day. Pennies.

SES. Inbound for the forwarding lane: $0.10 per thousand received messages (a few cents a year for an SMB). Outbound for email-fallback reminders: $0.10 per thousand sent. Both are negligible at this scale — which is exactly why email-only customers are so cheap to remind.

Bedrock (only when something fires it). The hourly tick uses no Bedrock. The inbound parsing lane fires Haiku 4.5 once per forwarded booking email: a few thousand input tokens (the parsed text) and a few hundred output tokens (the proposed row JSON), so a fraction of a cent per parse. The weekly staff summary is one larger call: write a short paragraph on the week's confirms, no-shows prevented, and slots freed; a fraction of a cent. Bedrock costs cents a month.

Textract (only on forwarded attachments). Many booking emails are plain text and skip Textract entirely. When a confirmation comes as a PDF or image, Textract runs per-page; a booking confirmation is usually one page. A few cents a month at most.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the reply links.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The reminder sleeps almost the whole hour.
- **A Knowledge Base.** The list is structured rows, not free text — deterministic lookup beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.

- **Models on the tick.** The hourly decision is plain Python. Bedrock fires only on the inbound parsing lane and the weekly summary.

How the cost scales

SMS grows roughly linearly with appointment count, because each appointment gets a fixed number of reminder texts. Lambda and DynamoDB grow linearly too, but from a much smaller base. Bedrock and Textract are uncorrelated with appointment count — they only fire when somebody forwards a booking email or it's the weekly summary. So the bill at 3,000 appointments is around $35, at 5,000 around $55 — almost all of it text messages. If the SMS line ever gets uncomfortable, you can lean harder on the free email channel for customers who'll read it, or drop a tight third reminder for low-value services.

Set an AWS Budgets alarm at $30/month so anything unusual — a runaway loop sending duplicate texts — pages you before the bill matters. The reminder's normal-volume bill stays well under that ceiling, and the one no-show it prevents each week more than pays for it.

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

PART 7 OF 7

MAY 7, 2026 PART 7 OF 7 · APPOINTMENT REMINDER SERIES ~8 MIN READ

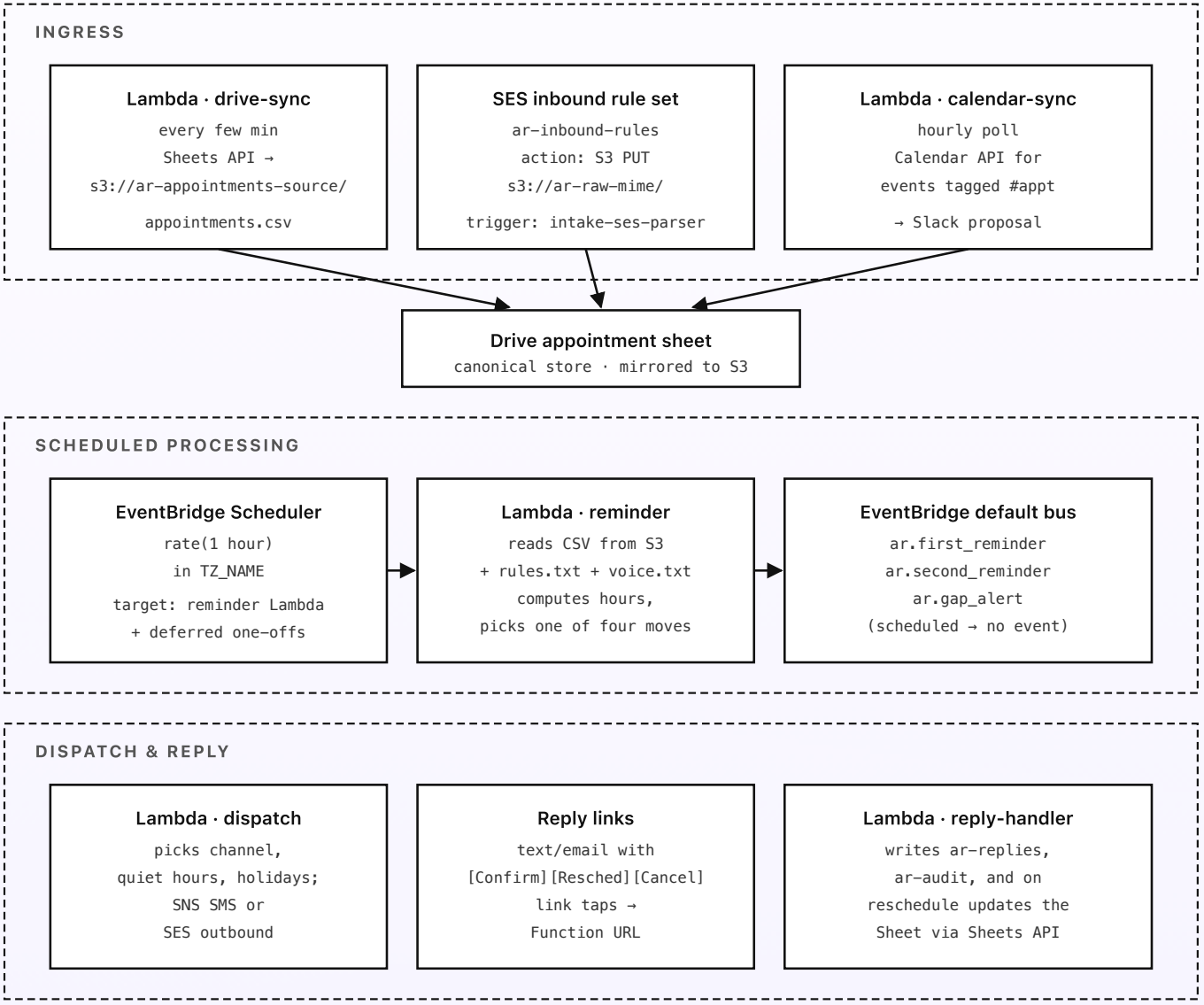
Engineering reference: the appointment reminder architecture

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 SMS reply 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, EventBridge Scheduler, and SNS SMS are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is a reminder that goes out an hour late, not a regional outage. One AWS account dedicated to the reminder (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. SMS origination identity (a sender ID or a long/short code, depending on the destination country's rules) is registered in the SNS console and referenced by the dispatch Lambda.

Topology



Every reminder leaves with full context — and every interaction is logged to ar-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the list), scheduled processing (the hourly reminder tick emitting events), dispatch and reply (the reminder ships and the customer's response is recorded). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

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

- **`drive-sync`** — EventBridge Scheduler target, fires every few minutes (default `rate(5 minutes)`; tighten only if your booking tool writes the sheet faster than the reminder needs). Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `ar/drive/sa`) to export the appointment sheet as CSV and write to `s3://ar-appointments-source/appointments.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://ar-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`calendar-sync`** — EventBridge Scheduler target, hourly. Uses the Google Calendar API `events.list` to scan configured calendars for events with `#appt` in the description; for any new events, creates a Slack interactive proposal message. For lower-latency setups you can switch to `events.watch` and have Calendar push notifications to a Function URL instead of polling, at the cost of renewing the channel before it expires (Calendar push channels have a finite TTL and need a small refresh job). Memory: 256 MB. Timeout: 30 s.

- **`intake-ses-parser`** — S3 PUT trigger on `s3://ar-raw-mime/`. Parses MIME; if the booking confirmation is plain text it goes straight to the model, and if it carries a PDF or image attachment it runs Textract via `StartDocumentTextDetection` (asynchronously to handle multi-page confirmations). On Textract completion (via SNS notification), reads the structured text and calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to propose an appointment row. Posts the proposal to Slack via the bot token with Approve/Edit/Discard buttons. For DOCX attachments (Textract doesn't accept them), falls back to `python-docx`; XLSX uses `openpyxl`. Both packages are stable and widely used in 2026, though their maintenance velocity is light — for a parsing path that only runs a few times a month, that's acceptable. Memory: 512 MB. Timeout: 60 s.
- **`reminder`** — EventBridge Scheduler target, hourly (the schedule expression runs in `TZ_NAME` set to the SMB's timezone, e.g. `Asia/Singapore`). Reads `s3://ar-appointments-source/appointments.csv` and the rules and voice docs. For each row, computes `hours_to_appt`, reads chain state from `ar-sends` and `ar-replies`, decides on a move. Emits one event per row that needs action: `ar.first_reminder`, `ar.second_reminder`, or `ar.gap_alert`, with the appointment context as the event payload. Scheduled (no-action) appointments emit nothing. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`dispatch`** — EventBridge rule on the three move events. Picks the channel, checks quiet hours and holiday calendar, formats the message from the voice template, and ships via SNS `Publish` (SMS, with the registered origination identity), SES `SendRawEmail` (email fallback), or the Slack bot token (gap alert to staff). On quiet-hours or holiday defer, creates a one-off EventBridge

Scheduler rule that re-invokes `dispatch` at the next allowed minute. Writes a row to `ar-sends` after a successful send. Memory: 256 MB. Timeout: 30 s.

- **`reply-handler`** — Lambda Function URL, public with `AuthType: NONE`; each reply link carries a signed, single-use token (HMAC over `appt_id` + action + nonce, secret in Secrets Manager) so a guessed URL can't confirm or cancel someone else's appointment. Triggered by customer taps on the Confirm/Reschedule/Cancel links. Writes to `ar-replies` and `ar-audit`; on reschedule, updates the Drive sheet via the Sheets API and archives the old chain in `ar-sends-archive`; on cancel, frees the slot and posts a gap alert to Slack. Renders a tiny HTML confirmation page. Memory: 256 MB. Timeout: 15 s.
- **`summary`** — EventBridge Scheduler target, weekly Monday 8am. Reads the past week's `ar-sends`, `ar-replies`, and `ar-audit`; calls Bedrock Haiku 4.5 to write a short staff narrative (confirms, no-shows prevented, slots freed, customers who went quiet); posts it to a configured Slack channel and emails it via SES to the owner. Memory: 512 MB.

Storage

- **DynamoDB · `ar-sends`** — one row per dispatch. PK `(appt_id, window_index)`; attributes: `sent_at`, `channel` (sms/email/slack), `recipient`, `move` (first_reminder/second_reminder/gap_alert). On-demand. No TTL.
- **DynamoDB · `ar-replies`** — one row per reply. PK `appt_id`; sort key `replied_at`; attributes: `action` (confirm/reschedule/cancel), `channel`, `old_time`, `new_time` (if action = reschedule). On-demand. The reminder reads this table for the "already confirmed or cancelled?" short-circuit.

- **DynamoDB · `ar-audit`** — one row per write action of any kind. PK `(appt_id, ts)`; attributes: `action`, `channel`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB · `ar-sends-archive`** — archived chains after a reschedule. Same shape as `ar-sends`; PK `(appt_id, chain_id, window_index)`. On-demand.
- **S3 · `ar-appointments-source`** — mirrored CSV from the Drive appointment sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 2 years.
- **S3 · `ar-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3 · `ar-raw-mime`** — raw inbound MIME from forwarded booking emails. Lifecycle to Glacier at 30 days; expiry at 1 year.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `intake-ses-parser` for the inbound booking parsing, and `summary` for the weekly staff narrative. The hourly tick never touches Bedrock; if you ever wanted a model to draft a warmer same-day message, Haiku 4.5 is the right tier and Sonnet 4.6 would be overkill.
- **Embeddings.** Not used. The list is structured rows; deterministic lookup beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at SMB volume. The reminder itself doesn't call Bedrock; the parsing lane fires a few times a month at most.

EventBridge Scheduler config

- `ar-hourly-tick` — `rate(1 hour)` in the SMB's timezone. Target: `reminder` Lambda.
- `ar-drive-sync` — `rate(5 minutes)`. Target: `drive-sync` Lambda.
- `ar-calendar-sync` — `rate(1 hour)`. Target: `calendar-sync` Lambda.
- `ar-weekly-summary` — `cron(0 8 ? * MON *)` in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `dispatch` when a quiet-hours or holiday defer is needed. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SNS SMS, SES inbound and outbound

- **SNS SMS.** Register an origination identity for each destination country in the SNS console (sender ID where allowed, otherwise a long or short code; some countries require pre-registration). Set the SMS type to `Transactional` for delivery priority, and set a monthly SMS spend limit in SNS as a hard backstop under the Budgets alarm. The dispatch Lambda calls `sns:Publish` with the customer's E.164 phone number.
- Set the MX record on a dedicated subdomain (e.g. `bookings.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `ar-inbound-rules`: one rule with recipient `bookings@your-company.com` → spam scan → S3 PUT to `s3://ar-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.

- SES outbound for the email-fallback reminders: verify a sender identity at `appointments@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **reminder role:** `s3:GetObject` on the appointments, rules, and voice keys; `dynamodb:Query` + `GetItem` on `ar-sends`, `ar-replies`; `events:PutEvents` on the default bus. *No* `bedrock:*`.
- **dispatch role:** `events:ListSchedules` + `CreateSchedule` for the deferred-send one-offs; `sns:Publish` for SMS with the registered origination identity; `secretsmanager:GetSecretValue` on the Slack bot-token secret; `ses:SendRawEmail` from the verified sender identity; `dynamodb:PutItem` on `ar-sends`; outbound network access to `hooks.slack.com`.
- **reply-handler role:** `dynamodb:PutItem` on `ar-replies` and `ar-audit`; `secretsmanager:GetSecretValue` on the Sheets-API service-account secret and the reply-token HMAC secret; outbound network access to `sheets.googleapis.com`; `dynamodb:Query` for chain-state lookup; on reschedule, `dynamodb:BatchWriteItem` for archiving the old chain to `ar-sends-archive`.
- **intake-ses-parser role:** `s3:GetObject` on `ar-raw-mime`; `textract:StartDocumentTextDetection`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack bot token.

- **drive-sync and calendar-sync roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the appointments and rules buckets; outbound network to `www.googleapis.com`.

Reply-link token flow

The one-tap links are public URLs, so each one carries a signed, single-use token rather than a raw appointment id. The token is an HMAC (key in Secrets Manager under `ar/reply/hmac`) over `appt_id`, the action (`confirm` / `reschedule` / `cancel`), and a per-send nonce, with a short expiry tied to the appointment time. `reply-handler` verifies the signature and the expiry, checks the nonce hasn't already been spent (a single-use marker in `ar-replies`), then processes the action. A confirm or cancel is one tap and resolves immediately; a reschedule renders a slot-picker page that posts back to the same handler with a second token. This keeps the surface stateless and tamper-evident without standing up an auth system for customers who'll never log in.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch metric for alerting.
- **Alarms:** reminder Lambda failures > 0 in an hour (the tick is the one piece that has to run); SNS SMS delivery-failure rate > 5% in 24h (might mean an origination-identity or registration issue); reply-handler token-verification failures > 10/hour (might mean a leaked or rotated HMAC secret).

- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $30/month threshold, alarm at 80% and 100%, posts to SNS topic `ar-cost-alarm` subscribed to the owner's email and Slack. Back it with the SNS monthly SMS spend limit so a runaway loop can't send unbounded texts.

Config and secrets

Service-account credentials for Drive, Sheets, and Calendar APIs all live in Secrets Manager under `ar/drive/sa` (one service account with scopes for all three APIs). The Slack bot token lives under `ar/slack/bot-token`. The reply-link HMAC key lives under `ar/reply/hmac`. SES sender identity lives in IAM and the verified-domain config; the SNS origination identity lives in the SNS config. The configured timezone, holiday list reference, quiet-hours window, per-service reminder schedules, and gap-alert Slack channel all live in Parameter Store under `/ar/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

Whichever IaC you prefer. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `ar-appointments-source` and `ar-rules-source` so a bad Drive edit can be rolled back in one click, set the SNS monthly SMS spend limit explicitly so a bug can't run up a text bill, and version the EventBridge Scheduler timezone setting so you don't accidentally start running the hourly tick in UTC after a CI rotation. CDK

with a Python stack file works well; SAM also fits. Total deployable surface: around seven Lambdas, four DDB tables, three S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SES rule set, one SNS origination identity, 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.