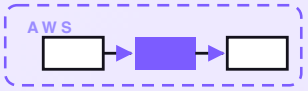
7-PART SERIES · FREE COMPANION



Booking assistant

A serverless booking assistant on AWS that reads appointment requests, checks Google Calendar, proposes slots that respect your service rules, locks in the customer's pick, and sends quiet reminders. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Booking assistant

PART 1 OF 7

APRIL 30, 2026 PART 1 OF 7 · BOOKING ASSISTANT SERIES ~5 MIN READ

A booking assistant on AWS for a few dollars a month

Booking is the part of running a service business that quietly eats your week — a customer asks for “sometime next Tuesday afternoon,” you check your calendar, you write back with three options, they pick one, you write the confirmation, you remember to send a reminder. Here’s how to design a small assistant that does all of that for you, and only escalates the unusual ones.

KEY TAKEAWAYS

- Four outside surfaces, three AWS pieces. The customer, your service rules, your Google Calendar, and your team — with reader, scheduler, and confirmer between them.
- Every request lands in one of four moves: propose 2–3 slots, draft for review, escalate to a human, or politely decline.
- The calendar is the source of truth. Every check goes through Google Calendar — no parallel availability database to drift.
- Slot writes are atomic. The confirmer claims a slot in DynamoDB before the calendar write, so two simultaneous taps can't double-book.
- Runs 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 building.

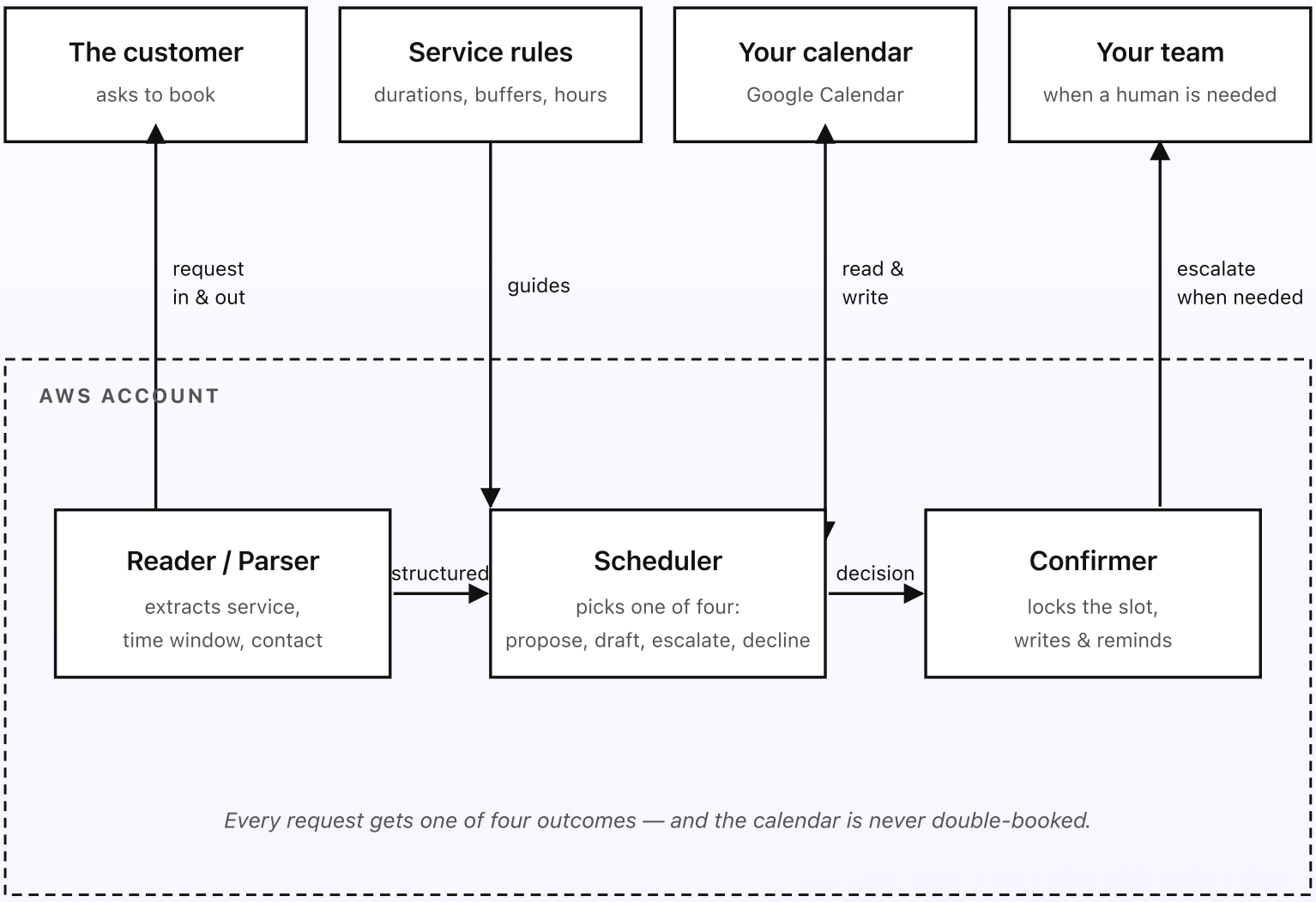

The customer
asks to book
Service rules
durations, buffers, hours
Your calendar
Google Calendar
Your team
when a human is needed
request
in & out
guides
read &
write
escalate
when needed
AWS ACCOUNT
Reader / Parser
extracts service,
time window, contact
structured
Scheduler
picks one of four:
propose, draft, escalate, decline
decision
Confirmer
locks the slot,
writes & reminds
Every request gets one of four outcomes — and the calendar is never double-booked.

Fig 1. Four outside surfaces, three pieces inside AWS. Request in, slot proposal back, calendar updated atomically when the customer picks one.

What you set up once (the outside)

- **A receiving surface** — a small embedded web form on your site (the fast path), or your existing email address (the free-text path). Either drops a request into the assistant in the same shape downstream.
- **A short service-rules file** — the services you offer with their durations and prices, working hours, buffer times between bookings, blackout dates, who's qualified to do what, deposit and cancellation rules, the tone you want for confirmations. Lives in a Google Doc you can edit anytime.
- **Your Google Calendar** — the same one you already use. The assistant reads availability and writes confirmed bookings into it. Nothing else moves.
- **An escalation inbox** — for ambiguous requests, VIP customers, and anything outside what you offer. Your normal inbox or a shared team queue, whichever you already use.

What runs on every request (the inside)

- **The reader / parser** — takes the inbound request, in whatever shape it arrived, and turns it into a structured form: which service, roughly when, who's booking. A web form arrives mostly structured already; an email might say "next Tuesday afternoon for a haircut."
- **The scheduler** — reads the structured request, queries the calendar, applies your service rules (duration, buffers, working hours, blackouts, who provides

what), and picks one of four moves: propose 2–3 viable slots, draft a proposal for your review, escalate to a human, or politely decline.

- **The confirmer** — once the customer picks a slot, atomically locks it on the calendar (so two simultaneous requests can't double-book), writes the event with the right details, sends the confirmation in your tone, and queues the reminder messages.

In plain words

A booking request lands. The cloud reads it — whether it came from a web form or an email like "can we do sometime next week?" A small AI turns the request into a clean shape, queries your calendar, applies your rules, and writes back with two or three slots that actually work. The customer picks one with a single click. The cloud writes the event into your calendar, sends a confirmation in your voice, and quietly sends a reminder the day before. You see only the bookings that are unusual.

Total cost runs in coffee-money territory at typical small-business volume — cents per booking, going up smoothly with how often the inbox rings.

DESIGN RULES THAT SHAPED EVERY DECISION

- The calendar is the source of truth. Every check, every write, goes through Google Calendar — not a parallel database that drifts.
- The assistant proposes from your service-rules file only — never invents durations, prices, or who can do what.
- Slot writes are atomic. The confirmer claims a slot before it confirms; if the calendar moved, it re-proposes.
- Auto-confirm only on clean requests with an explicit slot pick. Anything fuzzy becomes a draft proposal a human approves.
- Configuration lives in a Drive doc you can edit. Updating durations, hours, or blackouts never needs a deploy.
- If Google Calendar isn't your calendar, Microsoft 365 swaps in cleanly — same shape, different SDK.

Why this shape

Booking is one of those problems where most existing tools either over-promise or under-deliver. Per-seat scheduling apps charge the same whether you take ten bookings a month or ten thousand. Generic AI "assistants" happily promise slots that conflict with what's already on your calendar, or worse, invent service durations they don't know. Customer-facing booking widgets force the customer to think in your service taxonomy — "is this a 30-minute consult or a 60-minute first visit?"

The setup above splits the difference. A web form for customers who already know what they want (the easy 80%); a free-text path for the rest, parsed by a small AI; the calendar as the authority on availability; your service rules in plain English in a Drive doc; and atomic writes so the calendar is never wrong even if two requests collide.

The next five posts walk through each piece in turn — how a request reaches the assistant, how the parser turns fuzzy text into structured form, how the scheduler picks slots, how a booking gets confirmed without double-booking, and what the whole thing actually costs. One diagram per post. A final engineering reference at the end gives engineers the dense version with precise service names and model IDs.

PART 2 OF 7

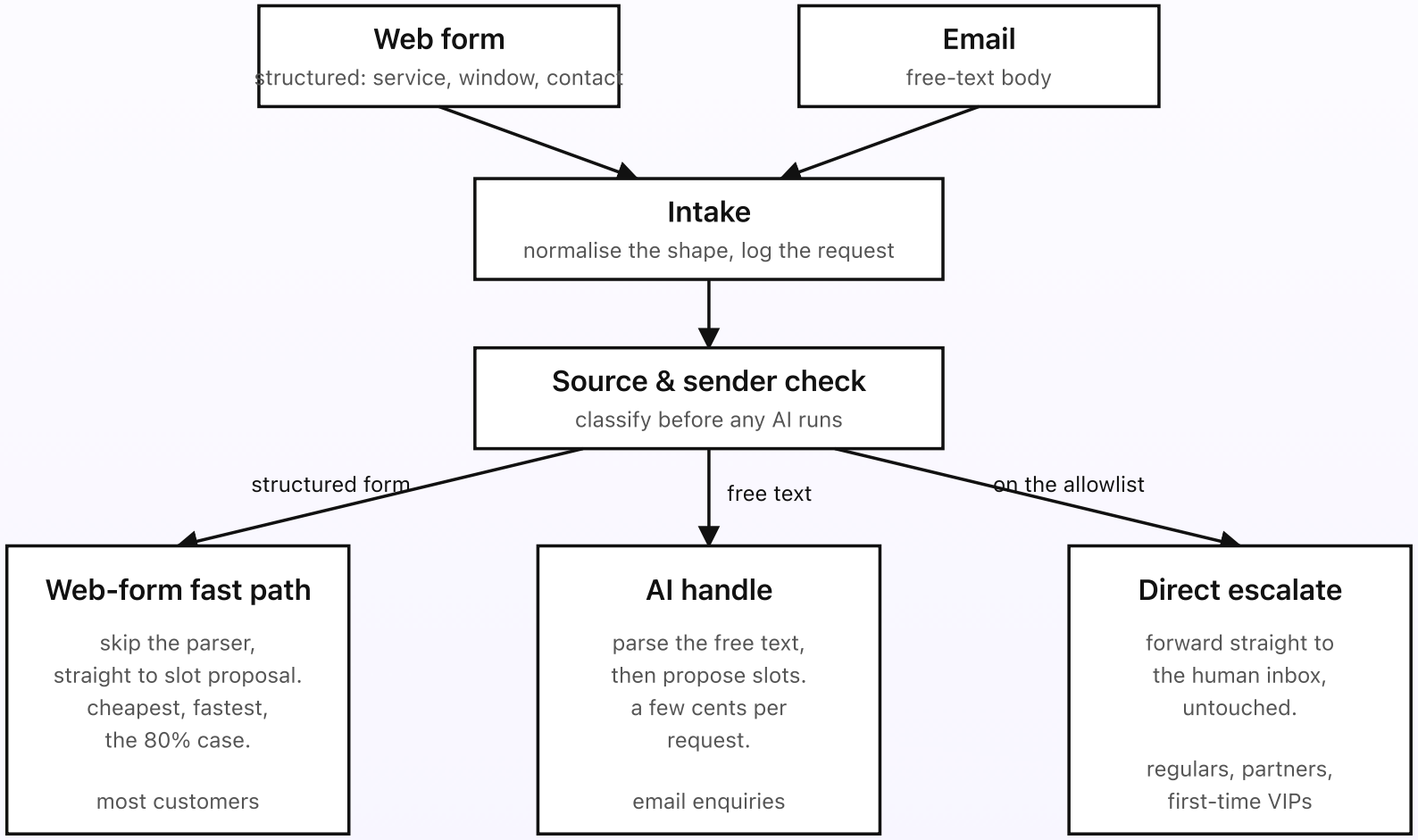
APRIL 30, 2026 PART 2 OF 7 · BOOKING ASSISTANT SERIES ~4 MIN READ

How a booking request reaches the assistant

Most customers know what they want and would happily click a button. Some write long emails because that's how they think. A few are regulars you'd never want to send through an AI in the first place. Three lanes at the door — one for each — and the scheduler only runs on the ones that need it.

KEY TAKEAWAYS

- Two surfaces in, one shape out. The web form and inbound email both drop a request into the same intake pipeline.
- Lane 1 — web-form fast path: structured fields skip the AI entirely and go straight to the scheduler.
- Lane 2 — AI handle: free-text email and partial form notes go through the parser before scheduling.
- Lane 3 — direct escalate: senders on the allowlist (regulars, partners, VIPs) reach a human untouched, with an `X-Assistant-Lane: direct` header you can audit.
- By the time the scheduler runs, the easy 80% is already answered, the VIPs are already through, and only the messy ones cost AI tokens.



By the time the scheduler runs, the easy ones are answered and the VIPs are already through.

Fig 2. Three lanes at the door. The fast path costs nothing; the AI path runs only when there's actually free text to parse.

Two surfaces, one shape

The assistant accepts requests from two places: a web form on your site, and the existing email address you already publish. Both paths drop a request into the same intake pipeline, in the same shape, so everything downstream — the parser, the scheduler, the confirmer — only ever has to handle one kind of object.

The web form is short on purpose. Three fields: which service, when (a date or rough window), and the customer's contact. Submitting it is one click, no account, no widget that takes thirty seconds to load. The form posts directly to a Lambda function URL and returns either "here are your slots" or a polite "we can't do that one, but here's what we can offer."

The email path catches the rest. Long messages, repeat customers who know your name, people who'd never use a form — all land in the same place. The intake step strips quoted threads and signatures the same way the email assistant in the previous series does, then passes the body to the parser.

Lane 1 — Web-form fast path

If the request came from the form and the three required fields are populated, the assistant skips the parser entirely. No AI runs. The structured request goes straight to the scheduler, which queries the calendar and proposes slots in a fraction of a second.

This is the cheapest possible path — just a Lambda invocation and a Google Calendar query. It's also the most common, and that's the point: the AI exists to handle the messy 20%, not to add cost to the easy 80%.

Lane 2 — AI handle

The default for everything that arrives as text. An email saying "hi, can I get a haircut sometime Tuesday afternoon? My name's Maria." A web form where the customer wrote their request in the "notes" field instead of picking a service. A reply on a thread you'd started days ago that now has "actually let's do Friday at 2" in it.

These all need the parser, which is the subject of the next post. The parser pulls out service, time window, and contact identity, and hands the result to the scheduler in the same shape the form uses.

Lane 3 — Direct escalate

The allowlist. A short list of email addresses, domains, or patterns that always skip the AI. Examples that usually belong on it:

- Existing customers booking a follow-up — you probably want to greet them yourself.
- Partners or referral sources sending you work.
- Anyone replying on a thread a human has already taken over.
- Senders whose previous request the assistant got wrong — they get pinned to the allowlist for a few weeks while the rules tighten.

For senders on the list, the intake step forwards the request to your normal inbox unchanged, with a small header tag (`X-Assistant-Lane: direct`) so you know it bypassed the AI. The customer notices nothing — they just get a human reply — and you stay in the loop on the relationships that matter most.

In plain words

Most booking requests don't need an AI to handle, and some shouldn't. A short web form catches the easy 80%; an email path with an AI parser catches the messy 20%; a small allowlist sends regulars and VIPs straight to a human. By the time the scheduler runs, every request has a known shape and a known reason to be there.

PART 3 OF 7

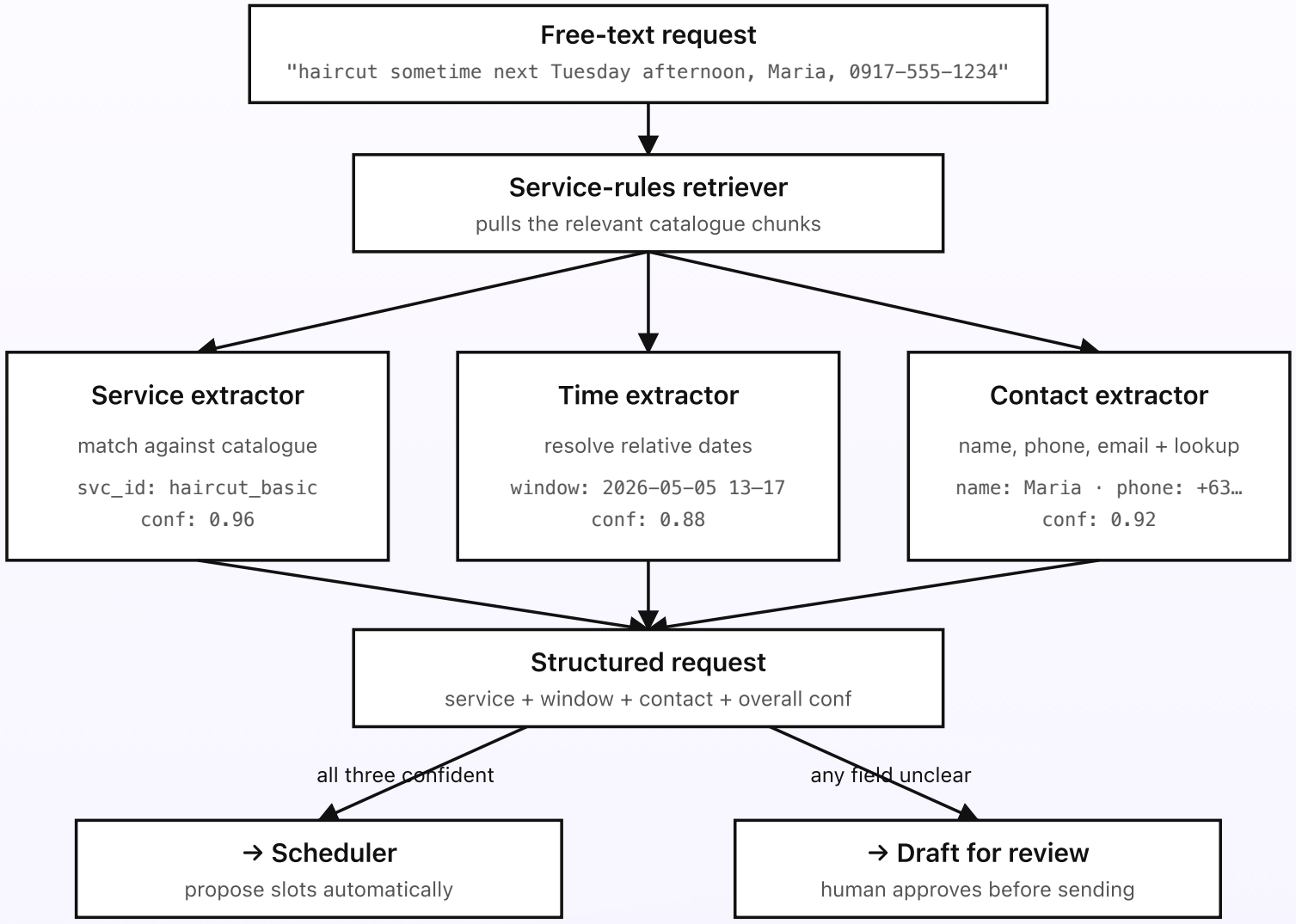
APRIL 30, 2026 PART 3 OF 7 · BOOKING ASSISTANT SERIES ~5 MIN READ

How the assistant understands the request

A booking request is really three questions stacked on top of each other: *what* service do you want, *when* roughly do you want it, and *who* are you. The parser's only job is to pull those three out of whatever the customer wrote and hand them on as a clean object. Anything it can't cleanly extract becomes a draft for human eyes, not a confident guess.

KEY TAKEAWAYS

- Three small extractors run in parallel — service, time window, contact — so each one is small, narrow, and easy to test on its own.
- The retriever pulls catalogue chunks from a vector index before the model runs, so the parser only ever sees services that actually exist.
- The time extractor turns “next Tuesday afternoon” into `{date_range, time_band}` anchored to the request timestamp in your business’s timezone.
- Every field returns a confidence score. Anything below threshold becomes a one-screen draft you approve in seconds — never silently guessed.
- The parser deliberately doesn’t price, doesn’t guess preferences, and doesn’t commit. The calendar write happens later, in the confirmer.



Every field is either confident or asked about — never silently guessed.

Fig 3. Three small extractors run in parallel, then merge. Confidence on every field; anything unclear becomes a draft.

Three things to pull, in parallel

The parser doesn't do natural-language understanding in general. It does three very narrow things in parallel, and that's a deliberate design choice — each extractor is small, has a single job, and is easy to test on its own.

The service

Match the customer's words against the service catalogue from your service-rules file. The catalogue is a small list (`haircut_basic`, `haircut_with_colour`, `blow_dry`, `first_visit`, etc.) with names, durations, prices, and a few aliases each ("trim," "quick cut," "just a haircut").

The retriever pulls the relevant chunk of the catalogue with a vector search before the model runs — so the parser sees only services that exist, not a hypothetical universe of them. The output is a service ID and a confidence score in `[0, 1]`. If the score is below the threshold, the parser doesn't guess; the request becomes a draft ("was this a basic haircut or a colour?") for you to clarify.

The time window

The hardest of the three, and the one most home-grown systems get wrong. "Next Tuesday" is unambiguous if you know what today is. "Tuesday afternoon" needs a definition of afternoon. "ASAP" is a window the size of your business hours. "The week of the 15th" is a different shape of window again.

The time extractor resolves all of these into the same shape: a date range plus a time-of-day band, both anchored to the request's timestamp in your business's timezone. "Next Tuesday afternoon" sent on Friday May 1 in Asia/Manila becomes `{date_range: [2026-05-05, 2026-05-05], time_band: [13:00, 17:00]}`. The scheduler in the next post will use that window directly to query the calendar.

The contact

Name, phone, and email if present, plus a lookup against your existing customer table. If the customer is returning, the extractor attaches their existing ID and any preferences on file ("always books with stylist A," "wheelchair access required"). If they're new, it just records what they wrote, plus a flag so you know to greet them as a first-timer in the confirmation.

This is the cheapest extractor — it's mostly a regex pass for phone numbers and emails, then a database lookup. The AI only runs if the message is unusual enough that the regex misses something obvious.

Confidence, not certainty

Each extractor returns its result *and* a confidence score. The merger combines them: if all three are above threshold, the request goes straight to the scheduler. If any one is below threshold, the request becomes a draft proposal — a one-screen view in your phone where you can correct the field, hit approve, and the assistant carries on from there.

Drafts feel slower the first week. They're actually faster: a five-second tap is cheaper than the ten-minute back-and-forth you'd have had to clarify the request

yourself.

What the parser deliberately doesn’t do

- **It doesn’t guess preferences.** “Maria, my usual” from a returning customer pulls in their last booking’s service if confidence is high; otherwise it asks. The assistant never invents a preference the customer didn’t state.
- **It doesn’t price.** Pricing is in the service catalogue and shown to the customer at the proposal step, not invented in the parser. If the customer asks about pricing in the request itself, that’s passed through to the scheduler so the proposal can include it.
- **It doesn’t commit.** Nothing the parser does writes to the calendar. That happens only in the confirmer, after the customer picks a slot.

In plain words

Three small extractors do one job each — what service, when, who — and each returns a confidence score. If everything’s clear, the assistant proceeds. If anything is borderline, you get a draft to approve in seconds. The parser is allowed to be confident, or to defer; never to silently guess.

PART 4 OF 7

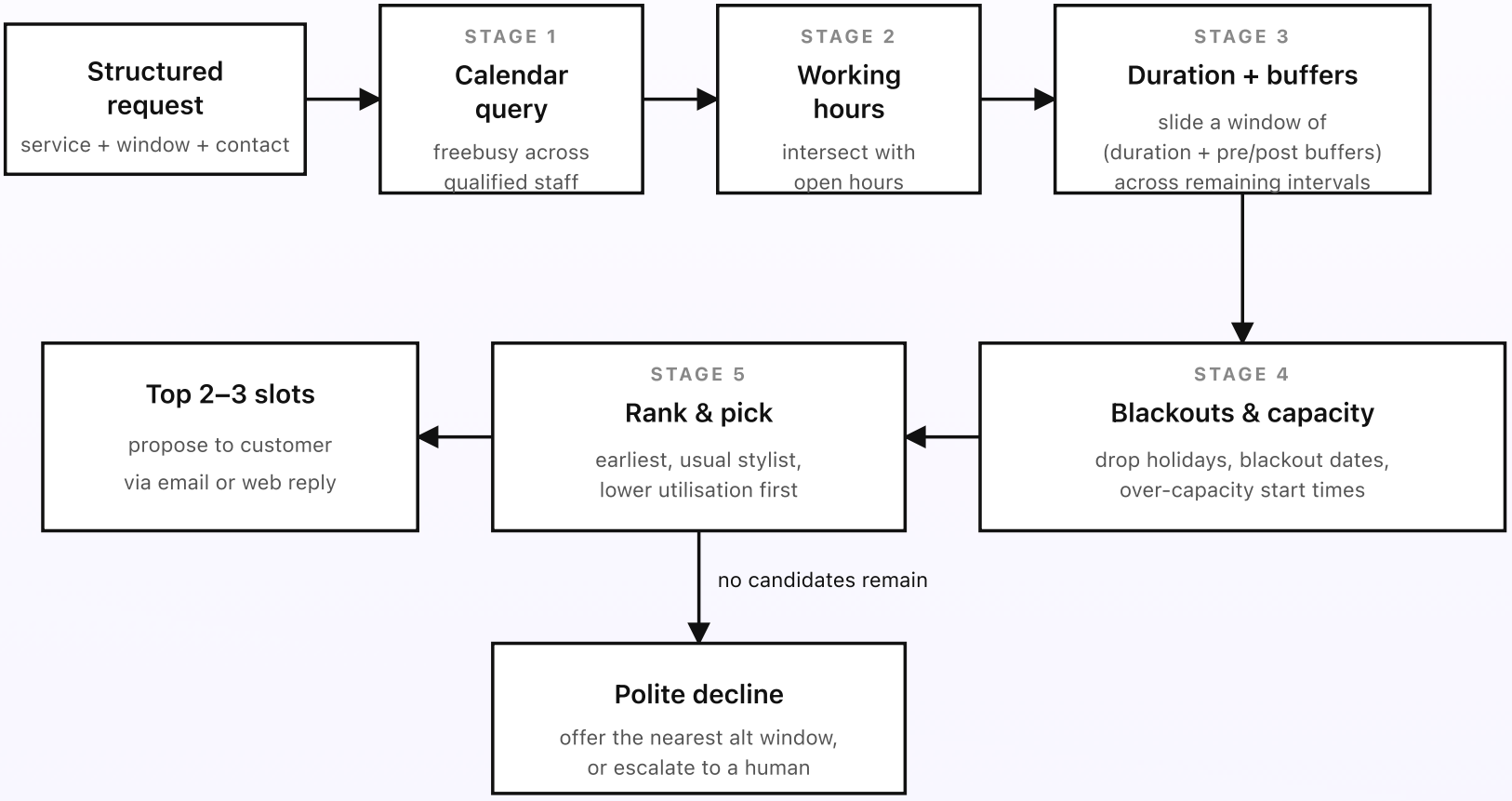
APRIL 30, 2026 PART 4 OF 7 · BOOKING ASSISTANT SERIES ~5 MIN READ

How the assistant picks slots

A structured request walks in. The scheduler walks out with two or three slots that fit the customer's window, your working hours, the service's real duration, the buffer you need before the next appointment, your blackout dates, and which staff member can do which service. The trick is doing this entirely on top of Google Calendar — never a parallel database that drifts.

KEY TAKEAWAYS

- Google Calendar is the only source of truth for availability — freebusy is queried at request time, no parallel database to drift.
- Five filters in strict order: freebusy query → working hours → duration plus pre/post buffers → blackouts and capacity caps → rank and pick.
- Buffers are why this isn't just "is the slot free?" A 10-minute post-buffer means the next bookable start after a 2:00 finish is 2:10, not 2:00.
- Per-staff working hours and capacity caps are honoured per resource, so stylist A and stylist B can have different schedules under one rules file.
- When nothing fits, the scheduler offers the nearest alternative window or escalates — a polite decline is one of the four moves, not an error.



Every constraint comes from the rules file or the calendar — nothing is hard-coded in the assistant.

Fig 4. Five filters in order. Each stage is cheap; the expensive ones (the calendar query, the ranking) only run on what survived the cheap ones.

The calendar is the source of truth

Most booking systems get into trouble by keeping a parallel availability database. The calendar says one thing; the database says another; somebody books over an existing event because the database is stale. The fix here is to never have the parallel database in the first place. Every availability check goes to Google Calendar's freebusy API at request time, and every confirmed booking is written to the calendar atomically (the next post is about that).

The freebusy API is fast (typically under 100 ms for a 7-day window across a small number of calendars) and free at the volumes a small business sees. Caching is tempting but unnecessary; the latency budget for a slot proposal is several seconds, of which the calendar query is a small slice.

Stage 1 — Calendar query

The scheduler asks the calendar for free intervals across every staff member who can perform the requested service. The service-rules file maps each service to a list of qualified resources (people, rooms, machines), and the scheduler queries the calendars of all of them in a single freebusy call. The result is a list of free intervals per resource, in the customer's requested window.

Stage 2 — Working hours

The freebusy API doesn't know your working hours; it just knows what's on the calendar. The scheduler intersects each free interval with the business's open hours from the rules file. A free interval from 11pm to 9am is real on the calendar, but useless to the customer; this stage drops it.

Per-staff working hours are supported here too — the rules file can specify that stylist A only works Tuesday–Saturday, while stylist B works Monday–Friday, and the scheduler intersects against the right set per resource.

Stage 3 — Duration and buffers

The service has a real duration (a basic haircut takes 30 minutes; a colour takes 90; a first visit takes 60 because of intake forms). It also has buffers — the time you need before and after the appointment for setup, cleanup, or breathing room. The scheduler slides a window of `(duration + pre + post)` minutes across the surviving free intervals at a configurable step size (15 min by default), and emits one candidate start time per fit.

Buffers are why this isn't just "is this slot free?" If the previous appointment ends at 2:00 and your post-buffer is 10 min, the next bookable start is 2:10, not 2:00. Skipping this is how you end up with no time to clean up between clients.

Stage 4 — Blackouts and capacity

Two more cuts. The blackout list (public holidays, your annual leave, the day the salon is closed for inventory) drops candidate starts that fall on those dates.

Capacity caps drop candidates that would push the day or the resource over a configured limit — useful for businesses where the constraint isn't just "the staff member is busy" but "we only do four colours per day so we don't run out of dye."

Stage 5 — Rank and pick

What survives gets ranked by preference, and the top 2–3 are returned. The default ranking puts earliest-in-window first (most customers want the soonest slot), then prefers the customer's usual staff member if known, then prefers slots that smooth the day's utilisation rather than concentrate it.

The ranking is configurable in the rules file, not hard-coded. Some businesses want the opposite — pack the morning so afternoons are free; some want premium services to go to the senior stylist by default. One config line, no deploy.

When nothing fits

Sometimes there's nothing in the customer's window. The scheduler doesn't shrug; it offers the nearest alternative window ("Tuesday afternoon is fully booked, but I have Wednesday morning"), or escalates to a human if the alternative would be a substantial change. A polite decline is one of the four moves — not an error.

In plain words

Five filters in order. Get free time from the calendar. Trim it to working hours. Fit the service's actual duration plus buffers. Drop blackouts and over-capacity slots.

Rank what's left and propose the top two or three. The scheduler does no thinking the calendar and the rules file haven't already authorised.

PART 5 OF 7

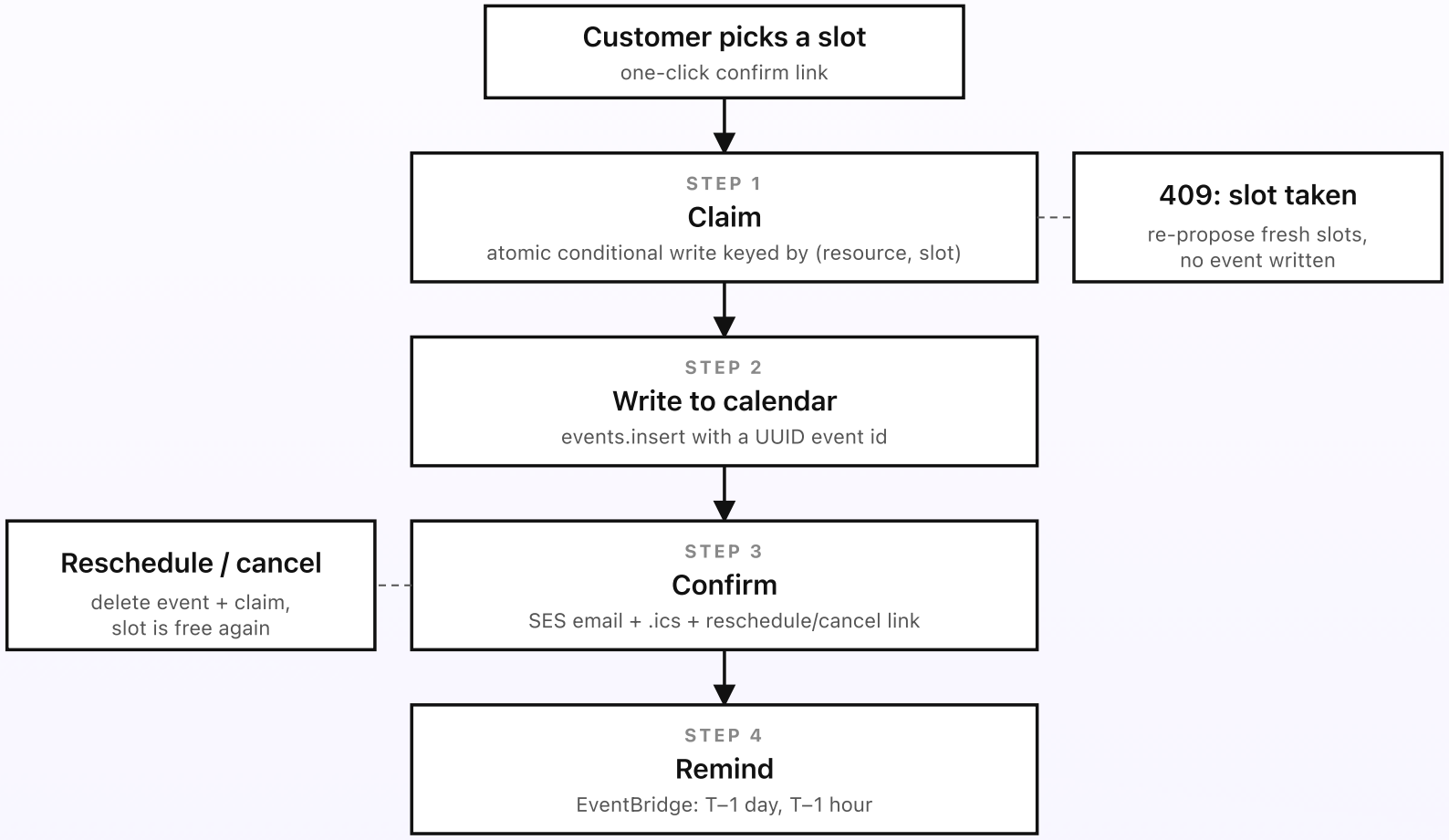
APRIL 30, 2026 PART 5 OF 7 · BOOKING ASSISTANT SERIES ~5 MIN READ

How a booking gets confirmed

The customer taps one of the proposed slots. From there, four things happen in a strict order — claim, write, confirm, remind. Each step is idempotent, the calendar is the source of truth, and a slot can only be claimed once even if two customers click at the same second.

KEY TAKEAWAYS

- Four steps in strict order: claim → write → confirm → remind. Each is idempotent and safe to retry.
- Step 1 claims the slot with a conditional `PutItem` on a DynamoDB table keyed by `(resource_id, slot_start_iso)`. The first request wins; the second gets a 409 and is bounced back to fresh slots.
- Step 2 calls Google Calendar `events.insert` with a stable UUID set as the event id, so retries return 409 duplicate instead of creating a second event.
- Step 3 sends an SES email with an `.ics` attachment plus signed reschedule and cancel links; step 4 schedules T–1 day and T–1 hour reminders via EventBridge Scheduler.
- Reschedule and cancel run the flow in reverse: delete the event, delete the claim row, cancel the pending reminders — the slot is free again.



Claim before write, write before confirm, confirm before remind. Partial failures leave a clean state.

Fig 5. Four steps in strict order. Each one is safe to retry; the claim step is the one that prevents double-booking.

Step 1 — Claim the slot before anything else

The double-booking problem isn't solved by "check the calendar, then write to it." Two requests checking at the same time both see the slot as free, and both write. The fix is a single atomic step that either succeeds or fails — and the easiest place to do it is a small DynamoDB table.

The table is keyed by `(resource_id, slot_start_iso)`. The claim step does a `PutItem` with `ConditionExpression: attribute_not_exists(resource_id)`. The first request for a given slot succeeds; any second request gets a `ConditionalCheckFailedException` and is bounced back to the proposal stage with fresh slots.

The claim row carries a TTL roughly equal to the slot's start time, so abandoned claims (the customer started but never finished the confirmation) auto-expire and free the slot.

Step 2 — Write to the calendar

Only after the claim succeeds does the confirmer call Google Calendar's `events.insert`. The event includes:

- **The right calendar.** The qualified resource's calendar (stylist A's personal calendar, the "chair 2" resource calendar, etc.) — not a global business calendar.

- **The summary and description** in your style: service name, customer name, customer phone, any notes from the request.
- **An attendee entry** for the customer with their email, so they get the calendar invite alongside the email confirmation.
- **A stable event id.** Google Calendar lets you set the `id` field on the event yourself instead of letting the server pick one. We use a UUID derived from the claim row's key, so the same write retried twice always gets the same id. The second try comes back as `409` duplicate — not a second event on the calendar.

If the calendar write fails (Google has a hiccup, the OAuth token rotated mid-flight), the confirmer rolls back the claim row so the slot is free again and the customer sees a clean "something went wrong, here are fresh slots."

Step 3 — Confirm to the customer

Now the customer hears back. The confirmation is a short SES email with:

- The service, the time, the staff member, and the location (or the video link, if it's a remote service).
- An `.ics` attachment so "add to my calendar" is one click on any device.
- A reschedule link and a cancel link, both signed and time-limited so they can't be replayed by anyone but the customer.
- The deposit-payment link if your rules file says this service requires one.

The tone is straight from your rules file — warm, terse, formal, whatever you've set. The same language model that does the parsing fills in the few variable bits;

the rest is a template.

Step 4 — Schedule the reminders

One-time EventBridge schedules trigger one day before and one hour before the appointment. Each reminder is a short message with the same details and the same reschedule/cancel links. If the customer reschedules, the old reminders are cancelled and new ones are scheduled against the new time. If they cancel, both reminders are cancelled.

The hour-before reminder is the one that pays for the system on its own. No-shows drop noticeably when customers get a quiet nudge an hour before they need to leave.

Reschedule and cancel: the same flow, in reverse

Both links lead to a small handler that does three things in order: delete the calendar event, delete the claim row (so the slot is free again), cancel the pending reminders. If it's a reschedule, the customer is then routed back to the slot-proposal stage for the same service, in a new window, and the loop closes when they pick a new slot.

What's deliberately not in this step

- **Payment up front, except where rules say so.** Most small services don't want a deposit gating the booking. The rules file decides per-service whether deposits are required.

- **Customer accounts.** Confirmation links and reschedule/cancel links are signed; no login needed. Adding accounts later is fine, but it's not required for the assistant to work.
- **Two-way calendar sync.** The assistant writes to the calendar; manual edits in Google Calendar (you drag an event to a new time, or block out an unexpected meeting) are picked up the next time the freebusy API is queried. There's no separate sync job to drift.

In plain words

Claim the slot atomically. Write the event with a stable UUID as its id. Send a confirmation with an `.ics` file and a reschedule link. Schedule two reminders. If anything fails, the previous step is the rollback. The customer sees a fast, friendly confirmation; you see a calendar that never lies.

PART 6 OF 7

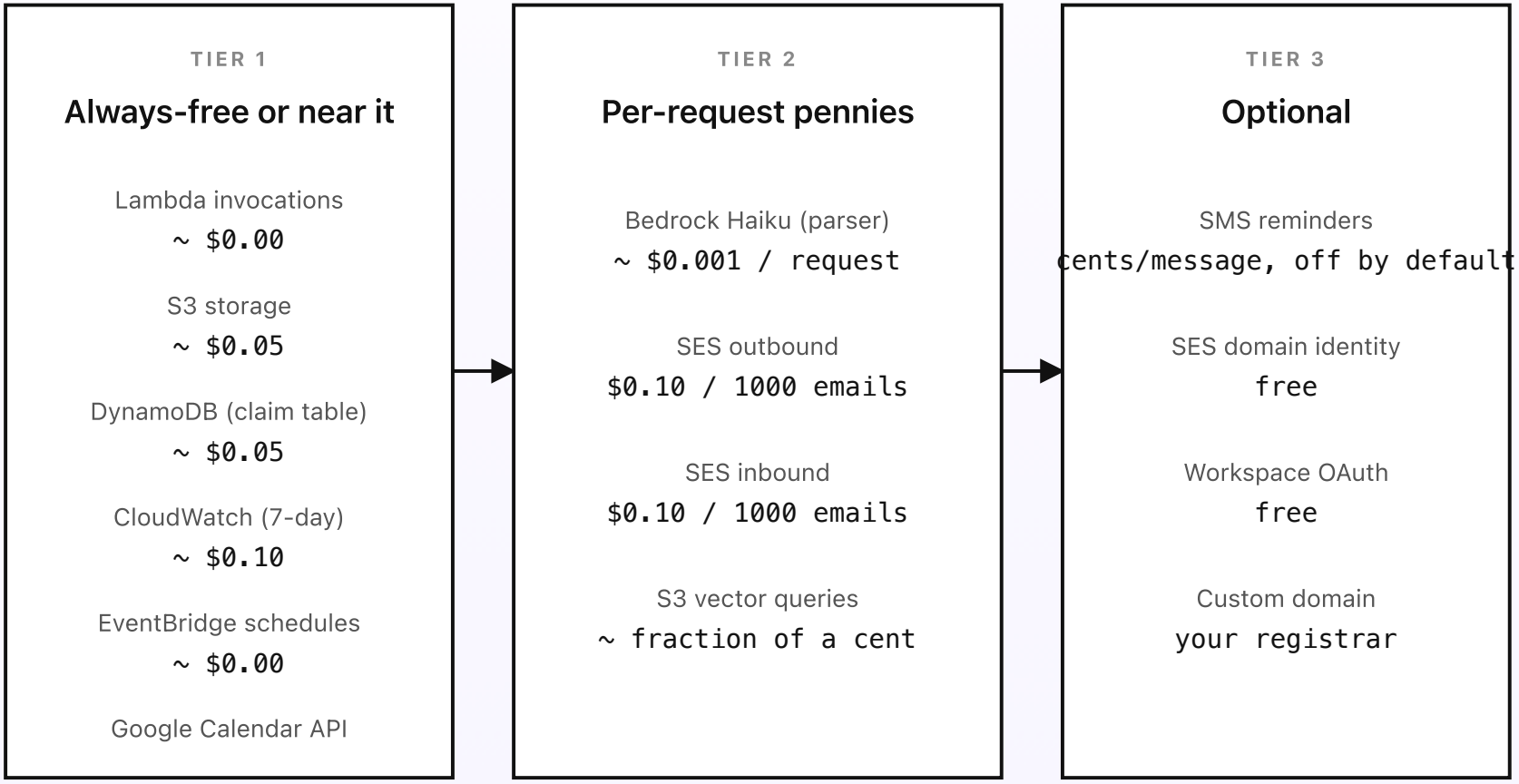
APRIL 30, 2026 PART 6 OF 7 · BOOKING ASSISTANT SERIES ~3 MIN READ

What the booking assistant costs

A coffee a month at typical SMB volume. The fixed cost is roughly nothing. The variable cost is dominated by how many requests need the AI parser — not the easy structured ones the web-form lane already handled.

KEY TAKEAWAYS

- Fixed cost is essentially zero. Lambda, S3, DynamoDB, EventBridge, and CloudWatch sit in or near the always-free tier at SMB volume.
- Google Calendar's freebusy and events.insert APIs are free at SMB volume — no separate calendar product to subscribe to.
- Variable cost is per-request pennies: SES inbound and outbound at $0.10 per 1,000 emails, plus Bedrock Haiku tokens (~$0.001 per parsed email-lane request).
- The web-form lane skips the AI entirely, so most requests cost only a Lambda invocation and a calendar query — not a model call.
- A typical 200-request month lands between $1 and $5 total. Set a $10 monthly AWS Budgets alarm and the bill can't surprise you.



~ 200 requests a month → under five dollars total, often under three.

Fig 6. Three tiers of cost. The bill scales smoothly with request volume; the floor is nearly zero.

The fixed cost is roughly nothing

Like the email assistant in the previous series, the booking assistant has no fixed monthly floor. There's no per-seat licence to pay. If the form is quiet for a week, the bill for that week is basically zero. Lambda, S3, DynamoDB, EventBridge, and CloudWatch all sit in or near the always-free tier at the volumes the assistant uses, and the vector index for the service rules is small enough to round to a few cents a month.

Google Calendar's freebusy and events.insert APIs are free at small-business volumes — the published quotas are far above any normal booking flow. There's no separate calendar product to subscribe to.

The variable cost is per-request pennies

Three things scale with request volume:

- **SES inbound** — only used by the email lane, about $0.10 per 1,000 emails received. (Pricing is technically per 256 KB chunk — a short text email is one chunk; a 700 KB attachment counts as three. Short booking enquiries fit in one chunk each.) The web-form lane doesn't touch SES inbound at all.
- **SES outbound** — about $0.10 per 1,000 emails sent. Each successful booking sends roughly three emails (confirmation + two reminders), so a hundred bookings is around three cents.

- **Bedrock Haiku tokens** — only the email lane runs the parser, and only the parser uses the model. A 100-request month with 30% on the email lane lands at a few cents of model spend.

Add it up: most small businesses end up between $1 and $5 a month total. The assistant pays for itself the first weekend you don't spend an hour playing email tag to confirm Saturday's appointments.

Three traps you're avoiding

- **Per-seat scheduling apps.** Most "booking app" products charge $20–$50 per inbox or per location per month, no matter how few bookings you take. You're trading a flat subscription for pay-per-use that mostly comes in pennies.
- **Running the parser on web-form requests.** The fast-path lane skips the AI entirely when the form is filled in. A simple setup that doesn't do this pays the model to read fields it could just have read directly; you don't.
- **Caching freebusy in a separate database.** The calendar is queried at request time. No drift, no two-sources-of-truth bugs, no reconciliation job.

When this stops being cheap

The math changes at high volume on the email lane. A busy operations inbox at thousands of free-text booking requests a month might land at $20–$40 in model spend — still cheaper than the per-seat products, but no longer coffee money. At that point, nudging more customers to the web form (which is also faster for

them) takes the bill back down. The form is the cheapest path for both you and the customer.

For everyone below that — and that's most small businesses — the bill is small enough that a $10 monthly AWS Budget alarm catches anything strange before you'd notice on the credit card.

In plain words

The fixed bill is nearly zero. The variable bill is cents per request, mostly only on the email lane. A typical small-business setup runs at coffee-money for the whole month. Set a budget alarm that fits your expected volume and the bill can't surprise you.

PART 7 OF 7

APRIL 30, 2026 PART 7 OF 7 · BOOKING ASSISTANT SERIES ~4 MIN READ

Engineering reference: the booking assistant architecture

Same system as the rest of the series, drawn purely for engineers. Service names, resource identifiers, region, Bedrock model IDs, Google Calendar API choices, and the actual flow operations — everything you'd need to recreate this in your own AWS account.

KEY TAKEAWAYS · VERIFIED MAY 2026

- Single AWS account in `ap-southeast-1` (Singapore); Bedrock via Global cross-Region inference.
- Five subsystems: Build & Deploy, Config Sync, Reader & Parser (intake), Scheduler (per request), Confirmer (per pick), with a Cross-cutting strip for audit, logs, alarms, and archive.
- Models: `global.anthropic.claude-haiku-4-5-20251001-v1:0` + `amazon.titan-embed-text-v2:0`; vector index is S3 Vectors (`vec-services`) for the service catalogue.
- Google Workspace auth: service account with domain-wide delegation, exactly three scopes — `calendar.events`, `calendar.events.freebusy`, `drive.readonly`.
- Atomic claim is in DynamoDB `tbl-claims` keyed by `(resource_id, slot_start_iso)` with `attribute_not_exists` condition expression; calendar event id is a stable UUID so retries return 409 duplicate.

Posts 1–6 walk through the system in plain language. This page is the dense version — nothing softened, just the architecture as you'd sketch it on a whiteboard during a design review.

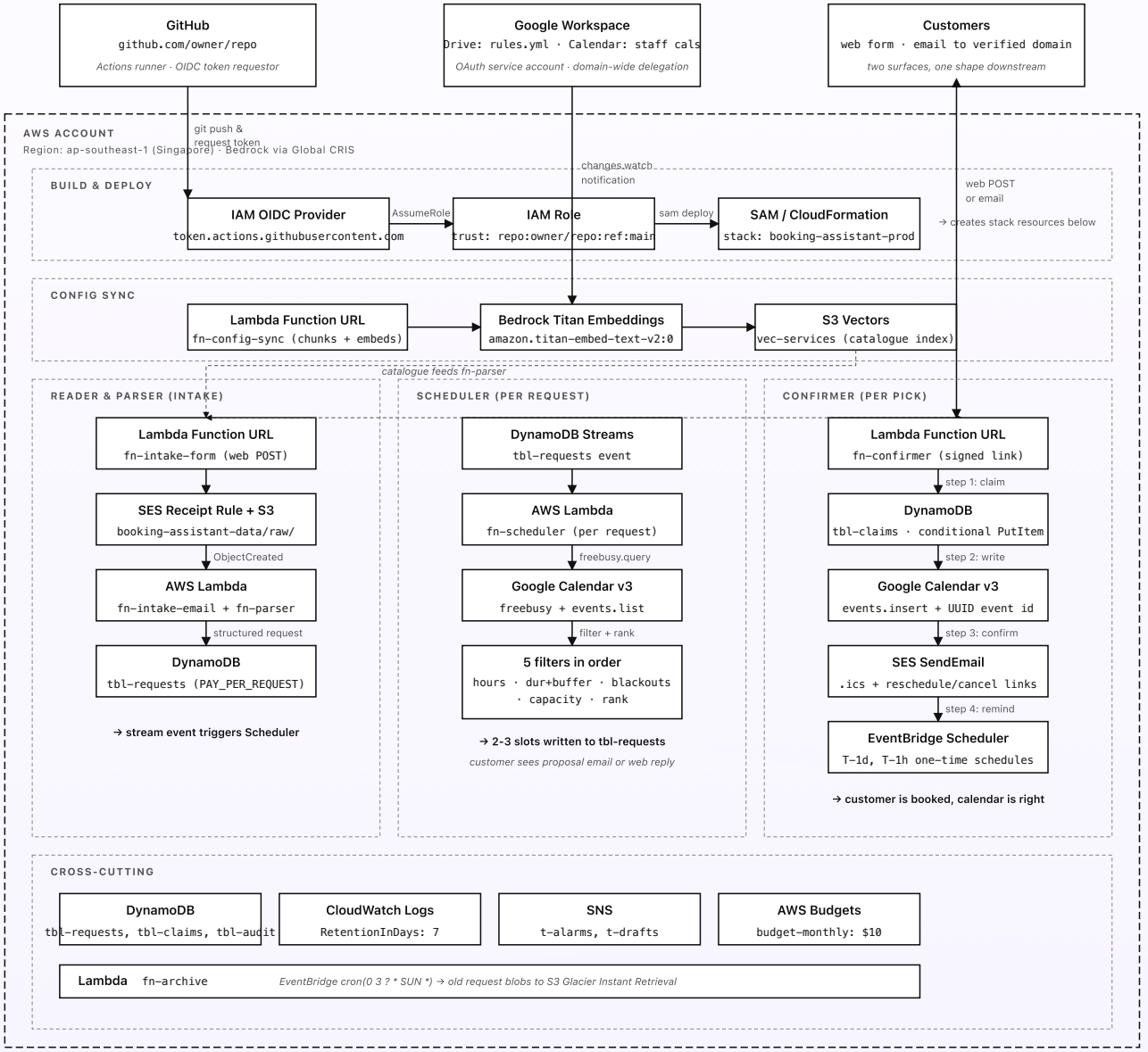

GitHub
github.com/owner/repo
Actions runner · OIDC token requestor
Google Workspace
Drive: rules.yml · Calendar: staff cals
OAuth service account · domain-wide delegation
Customers
web form · email to verified domain
two surfaces, one shape downstream
AWS ACCOUNT
Region: ap-southeast-1 (Singapore) · Bedrock via Global CRIS
git push & request token
changes watch notification
web POST or email
BUILD & DEPLOY
IAM OIDC Provider
token.actions.githubusercontent.com
AssumeRole
IAM Role
trust: repo:owner/repo:ref:main
sam deploy
SAM / CloudFormation
stack: booking-assistant-prod
→ creates stack resources below
CONFIG SYNC
Lambda Function URL
fn-config-sync (chunks + embeds)
Bedrock Titan Embeddings
amazon.titan-embed-text-v2:0
S3 Vectors
vec-services (catalogue index)
catalogue feeds fn-parser
READER & PARSER (INTAKE)
Lambda Function URL
fn-intake-form (web POST)
SES Receipt Rule + S3
booking-assistant-data/raw/
ObjectCreated
AWS Lambda
fn-intake-email + fn-parser
structured request
DynamoDB
tbl-requests (PAY_PER_REQUEST)
→ stream event triggers Scheduler
SCHEDULER (PER REQUEST)
DynamoDB Streams
tbl-requests event
AWS Lambda
fn-scheduler (per request)
freebusy.query
Google Calendar v3
freebusy + events.list
filter + rank
5 filters in order
hours · dur+buffer · blackouts · capacity · rank
→ 2-3 slots written to tbl-requests
customer sees proposal email or web reply
CONFIRMER (PER PICK)
Lambda Function URL
fn-confirmer (signed link)
step 1: claim
DynamoDB
tbl-claims · conditional PutItem
step 2: write
Google Calendar v3
events.insert + UUID event id
step 3: confirm
SES SendEmail
.ics + reschedule/cancel links
step 4: remind
EventBridge Scheduler
T-1d, T-1h one-time schedules
→ customer is booked, calendar is right
CROSS-CUTTING
DynamoDB
tbl-requests, tbl-claims, tbl-audit
CloudWatch Logs
RetentionInDays: 7
SNS
t-alarms, t-drafts
AWS Budgets
budget-monthly: $10
Lambda fn-archive
EventBridge cron(0 3 ? * SUN *) → old request blobs to S3 Glacier Instant Retrieval

Fig 7. Full architecture, ap-southeast-1. White boxes = AWS resources; dashed AWS container; dashed grey boxes = subsystem groupings; dashed grey arrows = config feed and side branches.

Read this top-down, then column-by-column

Top row is the three external surfaces. Below it, the AWS account contains five subsystems: Build & Deploy across the top, then Config Sync, then three runtime columns (Reader & Parser, Scheduler, Confirmer), with a Cross-cutting strip at the bottom. A request arrives either via the form (POST to `fn-intake-form`) or via email (SES → S3 → `fn-intake-email`), is parsed if needed, and is written as a structured row into `tbl-requests`. The DynamoDB stream then triggers `fn-scheduler`, which queries Google Calendar's freebusy API and applies the five filters from part 4. The proposal goes back to the customer with a one-click confirm link; tapping it invokes `fn-confirmer`, which runs the four-step claim/write/confirm/remind sequence from part 5.

Naming conventions used in the diagram

- **Lambda functions:** `fn-<purpose>` — `fn-intake-form`, `fn-intake-email`, `fn-parser`, `fn-scheduler`, `fn-confirmer`, `fn-config-sync`, `fn-archive`.
- **DynamoDB tables:** `tbl-requests` (every inbound request, with parsed shape, slots, and outcome), `tbl-claims` (the atomic-claim ledger keyed by `(resource_id, slot_start_iso)`, with TTL), `tbl-audit` (every action ever taken).

- **SNS topics:** `t-alarms` for general failures, `t-drafts` for parser-low-confidence drafts that need a human approval.
- **S3 layout:** single bucket `booking-assistant-data` with prefixes `raw/{date}/`, `config/`, `archive/`.
- **S3 Vectors index:** `vec-services` — chunked + embedded service catalogue for parser retrieval. The index lives inside an S3 *vector bucket*; the bucket is the parent resource, the index is what you query.

Region, model access, and Google Workspace auth

Everything runs in `ap-southeast-1` (Singapore) for low latency from the Philippines. Bedrock model invocations use the **Global cross-Region inference** profile (model IDs prefixed with `global.`) — data at rest stays in Singapore; inference may route to other regions for capacity. Pricing is the same as on-demand Singapore pricing.

Google Workspace authentication uses a service account with **domain-wide delegation**, granted exactly three scopes: `https://www.googleapis.com/auth/calendar.events` to write events, `https://www.googleapis.com/auth/calendar.events.freebusy` to read freebusy on those calendars (the broader `calendar.events` scope alone is not authorised for freebusy lookups), and `https://www.googleapis.com/auth/drive.readonly` for the rules file. The service account's private key is stored in AWS Secrets Manager and never leaves the runtime; rotation is a single CloudFormation parameter update.

The parser uses **strict tool_use**: three tool definitions (`extract_service`, `extract_window`, `extract_contact`) with required parameter schemas including a `confidence_score` in `[0, 1]` per field. The model can only emit structured tool calls — not a free-text reply. Free text would let it invent service IDs or hallucinate phone numbers; tool_use makes that mathematically impossible.

What's deliberately not on the diagram

- IAM policy details — per-Lambda execution role inline policies are minimal (one bucket prefix, one or two tables, SES on a single sending identity, Bedrock invoke on one model, Secrets Manager read on the Google service-account secret).
- Per-business rules schema — `rules.yml` is a single Drive doc with sections for services, working hours, blackouts, capacity caps, ranking preferences, and tone. Updating sections updates the assistant's behaviour without a deploy.
- X-Ray tracing — on for `fn-parser`, `fn-scheduler`, and `fn-confirmer`, sampling 100% during tuning, 10% in steady state.
- **Microsoft 365 swap-in** — if you're on Outlook/Exchange, replace the Google Calendar v3 calls with Microsoft Graph `getSchedule` and `events.create`. Same shape, different SDK, same idempotency story.
- **Bedrock Knowledge Bases** — managed retrieval that can replace the explicit S3 Vectors + Titan Embeddings setup with a single connector. Worth picking when the bring-your-own-rules-file pattern isn't strictly required; the catalogue is small enough that the explicit path is also fine.

- **Bedrock Guardrails contextual grounding check** — managed grounding-and-relevance scoring. The custom `confidence_score` per field in `fn-parser` is roughly the same idea hand-rolled; swapping to Guardrails moves the thresholds into console configuration and adds PII redaction on every model call. Worth turning on once the in-code thresholds are stable.
- **SMS lane** — an SMS surface (Twilio number → Lambda Function URL) slots in next to the email and form intakes with no other architectural change. Keeping it off the default diagram so the per-request cost stays in the always-free band.

IF YOU'RE RECREATING THIS

Start with Build & Deploy alone (a single Lambda, no triggers). Once `git push` reliably updates an empty stack, get a Google service-account credential into Secrets Manager and verify a hard-coded freebusy call from a Lambda. Then the form intake and a stub scheduler that always returns three made-up slots. Then the real five-filter scheduler. Then the claim/write/confirm/remind confirmer (one step at a time — the claim's atomic write is the part most worth integration-testing with parallel invocations). Then the email lane and the parser. Cross-cutting (audit, logs, alarms, budget, archive) goes in from day one.