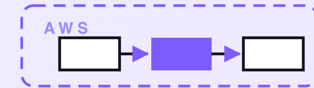
7-PART SERIES · FREE COMPANION



Booking deposit collector

A no-show is the quietest way an appointment business loses money: the slot was booked, nobody came, and it was too late to sell it to anyone else. This is the design of a small serverless system that asks for a deposit the moment a booking is made — it puts a PENDING hold on the slot so it can't be sold twice, sends one payment link in the business's voice, and waits. Pay the deposit and a signature-verified webhook flips the booking to CONFIRMED exactly once; ignore it and a scheduled sweep quietly releases the hold so the slot can be re-sold. On the day, the deposit is applied to the bill or forfeited on a no-show, per policy. It holds a slot once, reminds once before releasing, and never double-books. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/booking-deposit-collector

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

CONTENTS

Booking deposit collector

01 A booking deposit collector on AWS for a few dollars a month
02 How a booking requests a deposit
03 How a deposit payment gets confirmed
04 How an unpaid hold gets released
05 How a no-show gets handled
06 What the booking deposit collector costs
07 Engineering reference: the booking deposit collector architecture

PART 1 OF 7

JULY 5, 2026 PART 1 OF 7 · BOOKING DEPOSIT COLLECTOR SERIES ~10 MIN READ

A booking deposit collector on AWS for a few dollars a month

A no-show costs an appointment business twice: the slot sat empty and it was too late to fill. This post walks through the design of a small serverless system that asks for a deposit the moment a booking is made — holding the slot, sending one payment link, confirming the payment exactly once, and quietly freeing the slot if the deposit never comes.

KEY TAKEAWAYS

- A booking fires an event; within seconds the slot is held PENDING and the customer gets one message with a deposit link.
- When the deposit clears, the payment provider's signature-verified webhook flips the booking to CONFIRMED — exactly once.
- If the deposit isn't paid by a deadline, a scheduled sweep releases the hold so the slot is free to re-sell.
- One Bedrock call phrases each message in your voice. Every decision — the hold, the confirm, the release — is plain Python.
- Designed on AWS for about $2.40/month at roughly 120 bookings. It holds a slot once, reminds once, and never double-books.

The whole system on one page

Before any code, here's the shape of what we're designing. Every business that runs on appointments — a tattoo studio, a fine-dining restaurant, a dog groomer — loses money to the same quiet failure: the slot was booked, nobody turned up, and by the time anyone noticed it was far too late to sell it to somebody else. A deposit fixes it, but only if it's asked for at the right moment and chased without nagging. The system below does exactly that: a booking comes in, the slot is held, a deposit link goes out, and the calendar stays honest whether or not the money arrives.

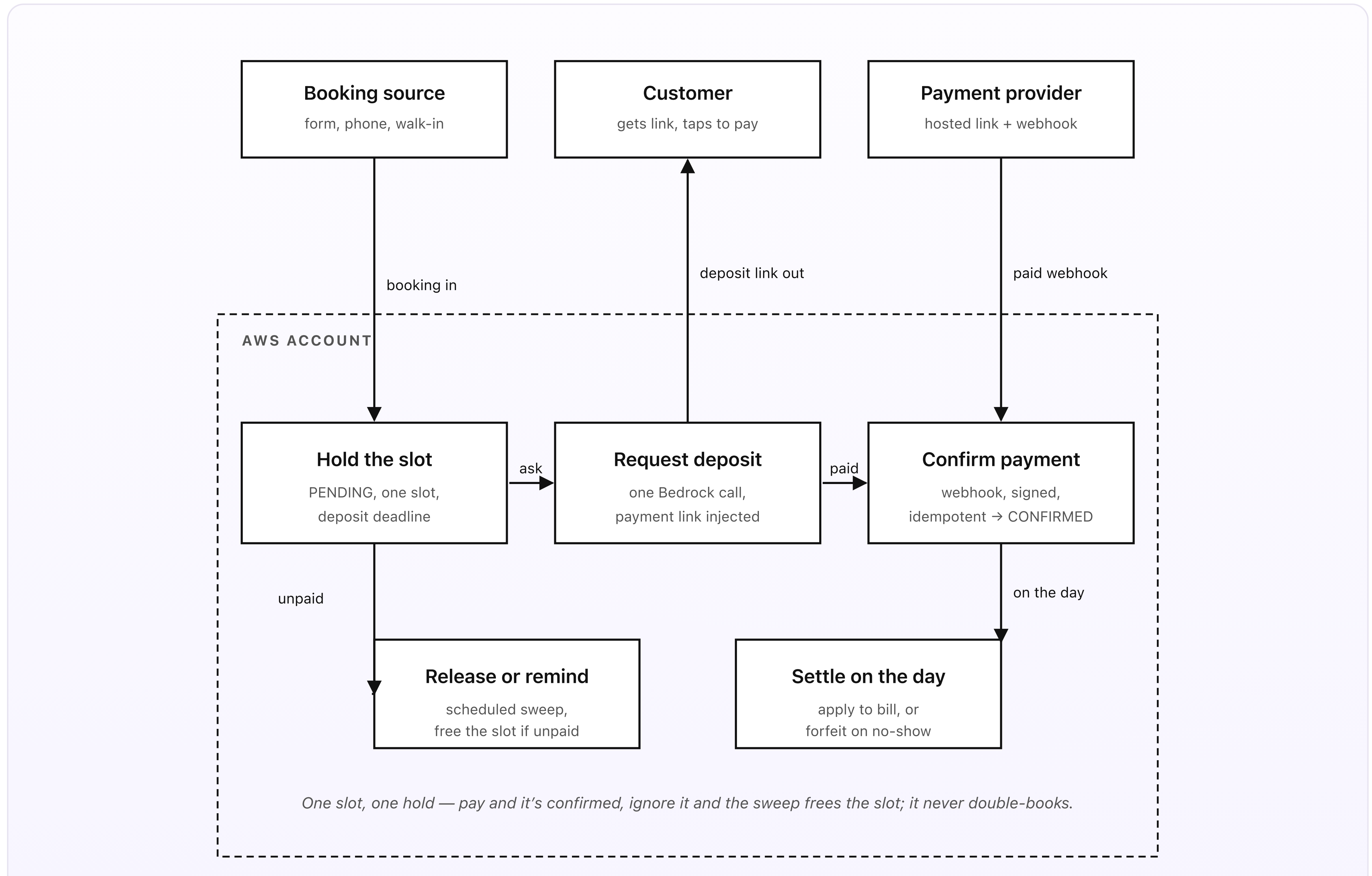

Booking source
form, phone, walk-in
Customer
gets link, taps to pay
Payment provider
hosted link + webhook
booking in
deposit link out
paid webhook
AWS ACCOUNT
Hold the slot
PENDING, one slot,
deposit deadline
ask
Request deposit
one Bedrock call,
payment link injected
paid
Confirm payment
webhook, signed,
idempotent → CONFIRMED
unpaid
on the day
Release or remind
scheduled sweep,
free the slot if unpaid
Settle on the day
apply to bill, or
forfeit on no-show
One slot, one hold — pay and it's confirmed, ignore it and the sweep frees the slot; it never double-books.

Fig 1. Three things outside, five pieces inside AWS. A booking comes in and Hold the slot places a PENDING hold; Request deposit sends one payment link; Confirm payment flips the booking to CONFIRMED when the webhook arrives. Release or remind frees an unpaid slot, and Settle on the day applies or forfeits the deposit.

What you set up once (the outside)

- **A booking source.** Wherever bookings already come from — your website's booking form, a reservation widget, or a member of staff typing one in from a phone call. It needs to do one thing: post the booking (who, when, which table or chair or slot) to one AWS URL. This is the trigger for everything, and it's covered in Part 2. You don't replace your booking flow; you tap into the moment a booking is created.
- **A payment provider.** A hosted checkout — Stripe, or similar — that can take a small deposit on a card and, crucially, fire a webhook the instant the payment clears. You never touch card details; the provider hosts the payment page, and your system only ever sees a signed "this deposit was paid" message. Its signing secret and API key live in Secrets Manager. This is Part 3, and it's where the money-handling care goes.
- **Your voice, your policy, and a place for handovers.** A small settings doc holds the business name and tone for the messages, the deposit amount and deadline, the quiet hours, and the no-show policy — whether a deposit is applied to the final bill, held against the table, or forfeited. Alongside it, an address (an inbox or a shared queue) where the system drops anything a person needs to handle: a disputed forfeit, a failed payment, an odd booking. The system decides the routine cases; a human owns the exceptions.

What runs on every booking (the inside)

- **Hold the slot.** The booking source posts to one Lambda Function URL. The function writes a PENDING hold on that exact slot with a conditional write — so a second booking for the same slot can never take it — and stamps a deposit deadline on the record. Only a genuinely free slot gets held. This is Part 2.
- **Request deposit.** One Bedrock Haiku 4.5 call takes the handful of facts it's allowed — the customer's name, the slot, the deposit amount — and phrases a single warm message. Code injects the real payment link afterwards, so it can never be mangled, and sends it. This is the end of Part 2.
- **Confirm payment.** When the deposit clears, the payment provider posts to a *second* Function URL. The function verifies the signature, then flips the booking from PENDING to CONFIRMED with a conditional write — so even if the same event arrives twice, the booking is confirmed exactly once. This is Part 3.
- **Release or remind.** A scheduled sweep sends one reminder before the deadline and, if the deposit is still unpaid when the deadline passes, releases the hold so the slot returns to sale. Each booking is reminded once and released once. This is Part 4.
- **Settle on the day.** For a confirmed booking, the deposit is applied to the final bill if the customer turns up, or forfeited if they don't — decided by the policy, recorded, and handed to a person for any dispute. This is Part 5.

In plain words

It's Tuesday and a table for six books online at The Copper Table for Saturday at 8pm — their busiest, most over-subscribed slot. The booking hits the system, which writes a PENDING hold on that table and time so no one else can grab it,

and within about ten seconds the customer’s phone buzzes: “Thanks for booking with The Copper Table, Marcus! To hold your table for six on Saturday, we ask for a £30 deposit — it comes straight off your bill on the night. Pay here: coppertable.uk/d/7f3a.” He taps, pays with Apple Pay, and the provider’s webhook lands a second later. The booking flips to CONFIRMED, the table is locked in, and nobody at the restaurant lifted a finger.

Two doors down, a different table books the same Saturday but never pays. The reminder goes out the next morning — “just a nudge, your table’s held until 6pm today” — and still nothing. At 6pm the sweep runs, sees the deadline has passed with the deposit unpaid, and releases the hold. The 8pm Saturday table is back on the booking page within the minute, ready for someone who’ll actually turn up. No awkward phone call, no table sitting empty on the busiest night of the week. The deposit did its job by never being paid: it filtered out the booking that was never real.

DESIGN RULES THAT SHAPED EVERY DECISION

- One slot, one hold. A conditional write on the slot means a booking is held once and never double-booked, however many requests arrive.
- One payment, one confirmation. The webhook is idempotent — the same event delivered twice still confirms the booking exactly once.
- Verify before you trust money. The payment webhook's signature is checked before a booking is ever moved to confirmed.
- An unpaid hold always expires. A slot is never held forever; the sweep reminds once, then releases it so it can be re-sold.
- The model only writes words. The hold, the confirm, the release, and the settlement are deterministic; Bedrock just phrases the messages.
- Policy decides, a person handles disputes. Apply-or-forfeit is a plain rule; anything contested goes to a human with the full record.

Why this shape

Most appointment businesses handle no-shows one of three ways: they eat the loss and grumble, they take a deposit by hand over the phone (slow, awkward, and easy to forget), or they buy into a heavy booking platform that bundles deposits with a monthly fee and a cut of every transaction. The first is the real cost — a busy Saturday with two no-shows can wipe out a night's margin. The second doesn't scale past a handful of bookings a day. The third works but is expensive and takes over your whole front-of-house. The gap is a light-touch collector that

clips onto the booking flow you already have and asks for a deposit the instant a booking lands.

The shape above fills exactly that gap and nothing more. It leans on the payment provider you'd use anyway to actually move the money, keeps your booking flow where it is, and adds a small system that holds the slot, asks once, chases once, and frees the slot if the answer is silence. The common case — a real customer who pays — is a tap on a link and a confirmed table. The few that never intended to come are quietly filtered out before they cost you the slot, and the genuinely awkward cases (a disputed forfeit, a failed card) are handed to a person with the whole story attached.

The next four posts walk through each piece in turn: how a booking becomes a held slot and a deposit request, how a paid deposit gets confirmed, how an unpaid hold gets released, and how a no-show gets settled. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JULY 5, 2026 PART 2 OF 7 · BOOKING DEPOSIT COLLECTOR SERIES ~8 MIN READ

How a booking requests a deposit

Before a deposit can be paid, the slot has to be safely held and the customer has to be asked. This post is about that first move: how a raw booking becomes a PENDING hold that can't be sold twice, with a deadline attached and one well-phrased message carrying the payment link out the door.

KEY TAKEAWAYS

- The trigger is a booking posted to one Lambda Function URL — the intake surface — separate from the payment webhook, and with no API Gateway.
- The slot is held with a conditional write on the slot key, so the first booking wins and a second one can never take the same slot.
- A booking that loses the race for a slot is told so and offered an alternative — it is never silently double-booked.
- The hold is stamped with a deposit deadline, the clock the release sweep in Part 4 later reads.
- Only after the slot is safely held does one Bedrock call phrase the deposit-request message, with the payment link injected by code.

From a booking to a held slot

Everything starts with a booking. However it's made — a form on the website, a reservation widget, or a member of staff keying in a phone call — the booking source posts a small HTTP request to a Lambda Function URL: who booked, when, and which resource (a table, a chair, a grooming slot). That POST lands on the intake function. There's no API Gateway in front of it; a Function URL is a plain HTTPS endpoint on the function itself, which is all a booking post needs and the cheapest way to receive one. This is one of two Function URLs in the whole system — the other, for the payment provider's webhook, is the subject of Part 3, and keeping them separate keeps their jobs and their permissions cleanly apart.

The intake function's job is not to send a deposit request. Its job is to *safely take the slot* — because everything downstream, the deposit link, the confirmation, the release, is meaningless if two customers can both be holding the same table. Most of this post is about how that hold is made exactly once, and only then how the ask goes out.

Holding one slot, and only one

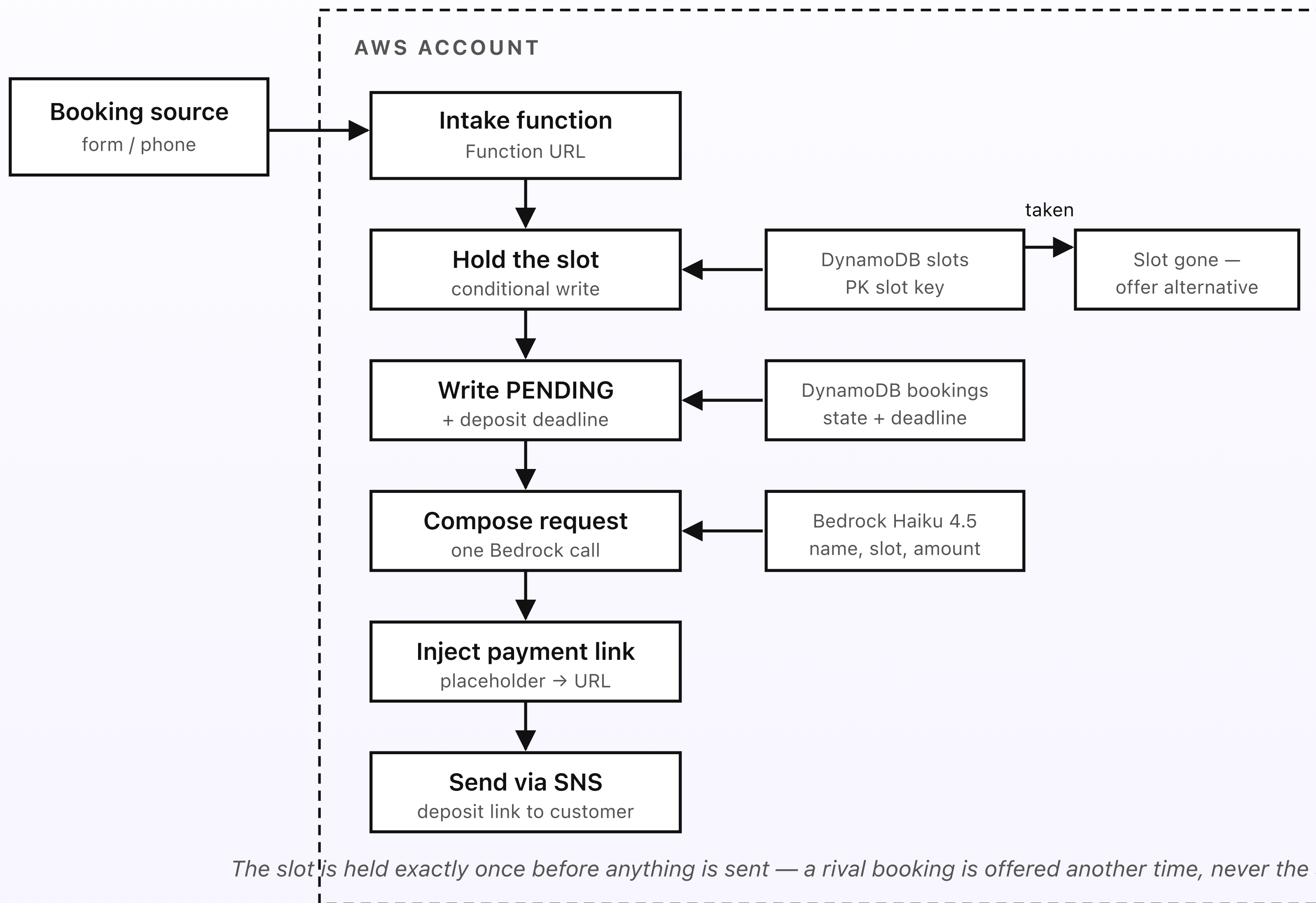
A slot is a scarce thing: one 8pm table for six, one 2pm chair with a particular tattoo artist, one Saturday grooming appointment. Two bookings can arrive for it within the same second — two customers on the website at once, or a form submit that fires twice. The system must let exactly one of them hold it. So the function doesn't just check whether the slot is free and then write; a check-then-write has a gap where both bookings see "free" and both proceed. Instead it writes the hold with a **conditional write** to DynamoDB, keyed on the slot itself, that only succeeds

if no hold already exists for that slot. The first write wins and creates the PENDING hold; any second write for the same slot fails the condition and is rejected. There is no window in which both can succeed, because the database, not the application, arbitrates.

A booking that loses that race isn't dropped on the floor. It's told, honestly, that the slot has just gone, and offered the nearest alternatives — the 8.30 table, the next chair, tomorrow's slot. What it is never given is a silent double-booking, because a double-booked Saturday night is a far worse customer experience than "that one just went, how about this?". The same conditional write is what makes the whole system safe to run at a busy business, where slots are contended and timing is tight.

Stamping the deadline

When the hold is written, the function stamps two things on the booking record: the state (PENDING) and a **deposit deadline** — the moment by which the deposit must be paid or the hold is forfeit. The deadline comes from the settings doc and suits the business: a restaurant might give until 6pm on the day, a tattoo studio 48 hours, a groomer 24. That timestamp is the single most important field for what comes later, because it's the clock the release sweep in Part 4 reads: no deadline, no way to know when an unpaid hold has gone stale. Setting it here, at the moment the slot is taken, means the countdown starts the instant the customer's intent is freshest.



The slot is held exactly once before anything is sent — a rival booking is offered another time, never the same one.

Fig 2. The intake gate. A booking hits the intake Function URL; the slot is held with a conditional write so only the first booking can take it, the booking is written PENDING with a deposit deadline, and only then does one Bedrock call phrase the request before code injects the payment link and sends it.

Asking once, and asking well

With the slot safely held, the ask goes out — and this is the one place in the intake path where a model runs. A single Bedrock Haiku 4.5 call is handed only the facts it may use: the customer's first name, the slot they booked, the deposit amount, and the business's voice notes. It writes one short, warm message asking for the deposit — friendly, not threatening, because a deposit request that reads like a debt collector loses the booking it was meant to secure. Haiku is right for the job precisely because the task is small: a sentence or two, in a set tone, running on every booking, for a fraction of a penny.

The payment link itself is never written by the model. The model emits a placeholder like `{{pay}}`, and the code substitutes the real hosted-checkout URL afterwards. This matters more here than almost anywhere: a mistyped link on a deposit request doesn't just read badly, it takes the customer's money to nowhere or, worse, somewhere wrong. By keeping the link out of the model's hands and injecting it deterministically — and validating that the finished message carries exactly one link, the amount, and a note that the deposit comes off the bill — the ask is always correct. If the model is slow or the draft fails a check, a fixed template goes out instead: plain, but always right, always on time. The model gives the request its warmth; the code guarantees the link and the number are exactly what they should be.

DESIGN RULES THAT SHAPED THE INTAKE STEP

- Two surfaces, kept apart. The booking intake has its own Function URL, separate from the payment webhook, with its own narrow permissions.
- The database arbitrates the slot. A conditional write on the slot key holds it exactly once; a check-then-write would leave a double-booking gap.
- Losing the race is handled. A booking that can't take the slot is offered an alternative, never silently double-booked.
- Set the clock at the hold. The deposit deadline is stamped the moment the slot is taken — it's what the release sweep reads later.
- Ask only after the slot is safe. Nothing is sent to the customer until the hold is written and the booking is PENDING.
- The link is the code's job. The model phrases the ask; the payment URL and the amount are injected and validated deterministically.

PART 3 OF 7

JULY 5, 2026 PART 3 OF 7 · BOOKING DEPOSIT COLLECTOR SERIES ~8 MIN READ

How a deposit payment gets confirmed

Money makes this the step that has to be exactly right. When the deposit clears, the payment provider posts a webhook — and this post is about trusting it: verifying its signature so a stranger can't fake a payment, and flipping the booking to confirmed once and only once, however many times the event is delivered.

KEY TAKEAWAYS

- The payment provider posts a `payment.succeeded` webhook to a second Lambda Function URL — the money surface, kept separate from the booking intake.
- The webhook's signature is verified against a secret in Secrets Manager before anything runs, so a stranger can't fake a paid deposit.
- The event is recorded idempotently: providers re-deliver webhooks, so the same event id is only ever processed once.
- The booking is flipped PENDING → CONFIRMED with a conditional write, so a duplicate delivery can never confirm twice or fire a second confirmation.
- The system reads the money from the signed webhook, never from the customer's browser — the redirect back to your site is a courtesy, not the source of truth.

The step that has to be exactly right

When the customer pays their deposit, the booking has to become real — and because money is involved, this is the step with the least room for error. Two failures matter here and neither is theoretical. If the system could be tricked into believing a deposit was paid when it wasn't, someone could hold your busiest slots for free. And if a single genuine payment could confirm a booking twice — or refund twice, or email the customer twice — the business's books and inbox both go wrong. So the confirmation path is built around two guarantees: only the real

payment provider can trigger it, and a given payment confirms the booking exactly once.

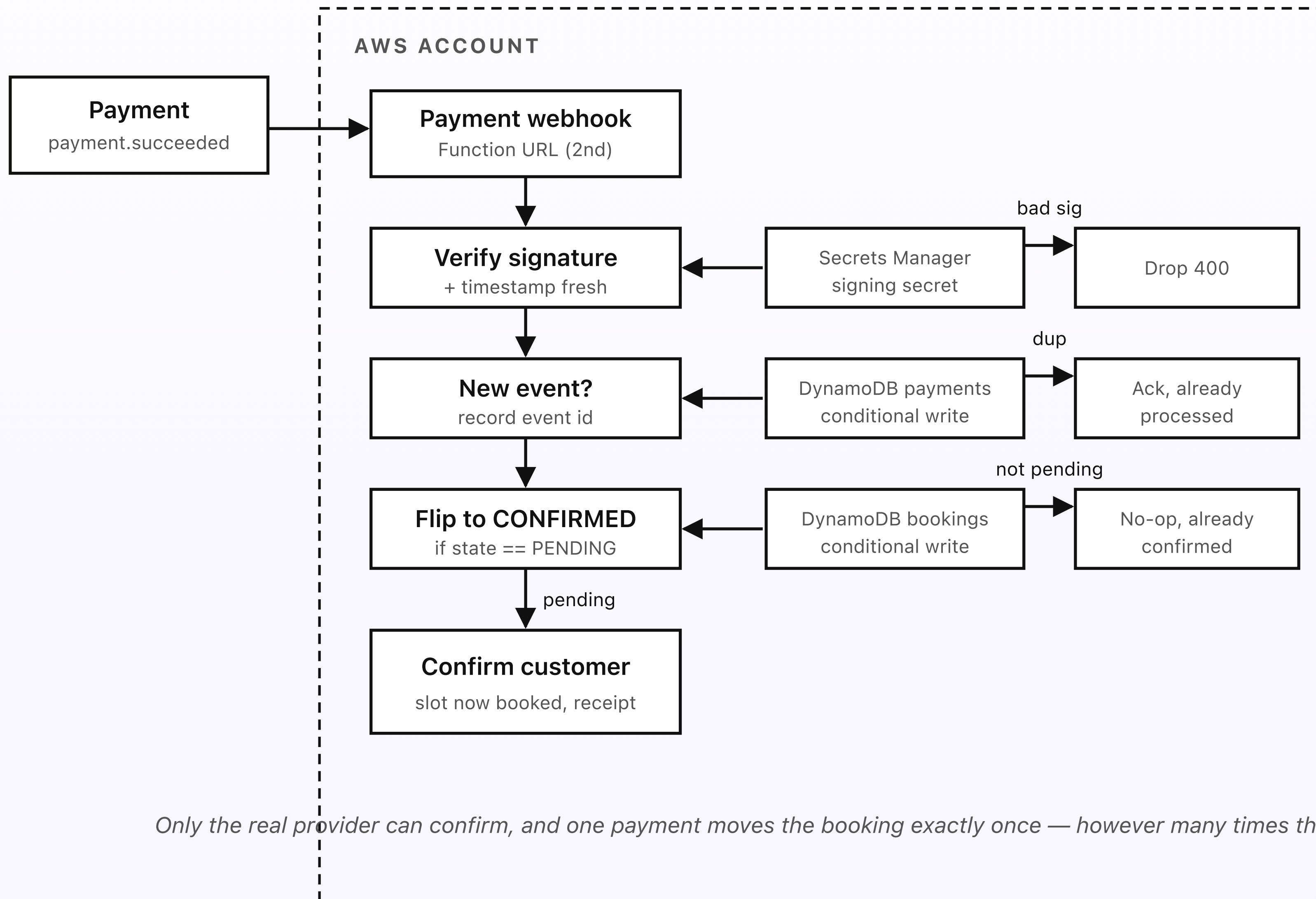
The confirmation comes in as a webhook. The moment the deposit clears, the payment provider posts a small signed message — `payment.succeeded`, with the payment id, the amount, and a reference tying it back to the booking — to a Lambda Function URL. This is the *second* Function URL in the system, deliberately separate from the booking intake in Part 2. Keeping the money surface apart from the booking surface means each has exactly the permissions it needs and no more: the webhook function can confirm bookings but can't create them, and the intake function can create bookings but can't confirm payment.

Prove the provider sent it

A Function URL is public, so the very first thing the webhook function does is verify the provider's signature. Payment providers sign every webhook with a secret shared only with you; the signature is a hash of the exact request body and a timestamp. The function recomputes that hash using the signing secret held in Secrets Manager and compares it — in constant time — against the one on the request. If they don't match, or the timestamp is stale (which blocks an attacker replaying an old captured event), the request is dropped with a `400` and nothing else happens. Without this check, anyone who found the URL could post a fake "deposit paid" and confirm bookings for free; with it, only messages genuinely from your provider get any further. This is why the money never comes from the customer's browser: a redirect back to your site after payment can be forged or simply lost, so the signed server-to-server webhook is the only thing the system trusts.

Exactly once, even when it arrives twice

Payment providers guarantee they'll deliver a webhook *at least* once, which means they will sometimes deliver it more than once — a network blip, a slow response, and the same `payment.succeeded` is sent again. The system has to treat that as a fact of life, not an error, and still confirm the booking exactly once. It does this in two layers. First, it records the payment event id with a conditional write to a payments table — the write only succeeds if that event id hasn't been seen before, so a re-delivery finds the record already there and stops, cleanly acknowledged. Second, and this is the real backstop, the confirming write to the booking is itself conditional: it moves the booking to CONFIRMED *only if it is currently PENDING*. If a duplicate slips through, or two deliveries race, the second one finds the booking already CONFIRMED, the condition fails, and it does nothing. One payment, one state transition, one confirmation email — guaranteed by the database, not by hoping the event only comes once.



Only the real provider can confirm, and one payment moves the booking exactly once — however many times the webhook

Fig 3. Confirming a deposit. The payment webhook is verified by signature against a Secrets Manager key, the event id is recorded with a conditional write so a re-delivery is ignored, and the booking is flipped PENDING → CONFIRMED only if it's still PENDING — so one payment confirms the booking exactly once and sends exactly one receipt.

What confirmation actually does

Once the conditional flip succeeds, the booking is genuinely confirmed and three small things follow, all downstream of that single state change. The PENDING hold becomes a firm booking — the slot is now sold, not just reserved. The customer gets one short confirmation, again phrased by a Bedrock call but with the facts (the slot, the deposit paid, that it comes off the bill) injected by code. And the release sweep from Part 4 will now leave this booking alone, because it only ever looks at bookings that are still PENDING past their deadline; a CONFIRMED booking is simply invisible to it. Because every one of these actions hangs off the exactly-once state transition, none of them can happen twice: no double confirmation, no doubled figure in the day's takings, no second text to a customer who paid once.

Edge cases are handled deterministically, not guessed at. A payment for an amount that doesn't match the expected deposit, a payment whose reference points at a booking that's already been released, or a rare `payment.succeeded` for a booking the system has no record of — each is recorded and routed to a person rather than auto-confirmed, because a mismatched payment is exactly the kind of thing a human should look at. The webhook is always acknowledged with a `200` once it's safely recorded, even in these cases, because a provider that

doesn't get an acknowledgement will keep retrying — and the whole point of the idempotency is that those retries are harmless.

DESIGN RULES THAT SHAPED CONFIRMATION

- A separate money surface. The payment webhook has its own Function URL and its own tight permissions, apart from the booking intake.
- Verify before you believe. The signature and a fresh timestamp are checked first — a forged or replayed webhook never confirms a booking.
- Trust the webhook, not the browser. The signed server-to-server event is the source of truth; the post-payment redirect is only a courtesy.
- Record the event once. A conditional write on the payment event id makes re-delivery a harmless no-op.
- Confirm on a condition. The booking moves to CONFIRMED only if it's still PENDING, so one payment transitions it exactly once.
- Mismatches go to a person. Wrong amounts or unknown references are recorded and escalated, never confirmed on a guess.

PART 4 OF 7

JULY 5, 2026 PART 4 OF 7 · BOOKING DEPOSIT COLLECTOR SERIES ~7 MIN READ

How an unpaid hold gets released

Most deposits get paid. Some don't — and an unpaid hold that never expires is a slot the business can't sell to anyone. This post is about the safety valve: the scheduled sweep that nudges once before the deadline and, when it passes with no payment, quietly releases the hold so the calendar keeps moving.

KEY TAKEAWAYS

- An unpaid hold that never expired would be a slot the business can't sell — so the system watches the deadline, not just the payment.
- An EventBridge Scheduler rule runs a sweep on a fixed cadence, looking for PENDING bookings near or past their deposit deadline.
- Before the deadline, it sends one reminder — a single nudge, never a stream of them — and marks the booking reminded so it's never nudged twice.
- At the deadline, an unpaid hold is released: the slot row is freed and the booking marked RELEASED, so the slot is back on sale within minutes.
- Release is a conditional write — it only frees a booking that's still PENDING — so a deposit that lands at the last second is never wrongly cancelled.

The holds that go quiet

Most deposits get paid, and Part 3 takes those from PENDING to CONFIRMED within seconds. But some don't. The customer books, means to pay, and then the meeting overruns or the phone goes in a pocket and the deposit never happens. From the calendar's point of view this is the dangerous case, because the slot still *looks* taken — there's a PENDING hold on it — even though no real commitment was ever made. Left alone, that hold sits on your busiest Saturday table forever, blocking customers who *would* have paid. The thing that needs acting on here

isn't an event that happened; it's an event that *didn't* — a payment that never came — and a system that only reacts to incoming webhooks would never see it.

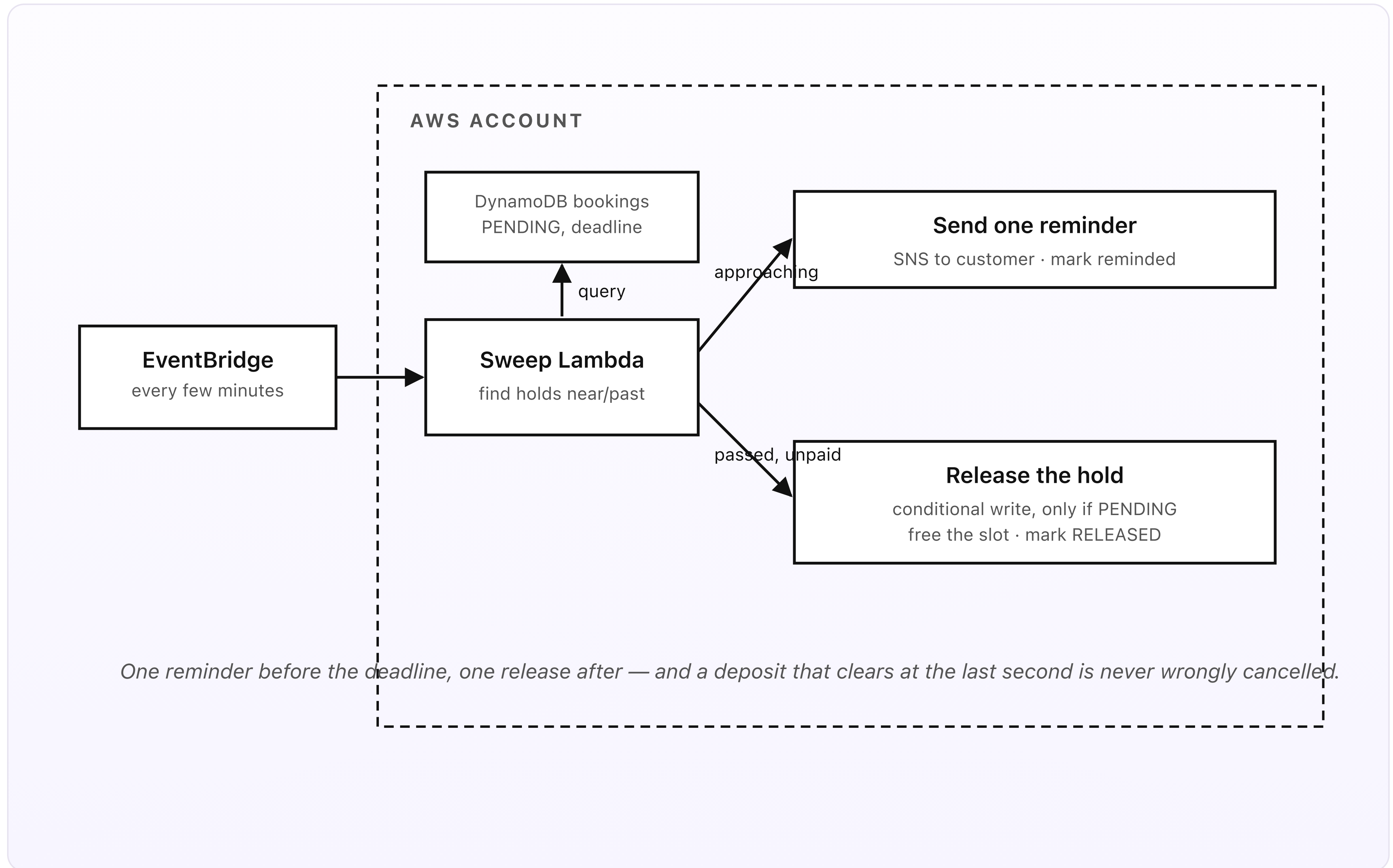
Catching an absence needs something that runs on a clock rather than on a trigger. That's the release sweep: a small scheduled job whose whole purpose is to find holds that are drifting toward their deadline, nudge them once, and free the ones that go past it.

A job on a clock

The sweep is driven by EventBridge Scheduler — a managed cron that invokes a Lambda on a fixed cadence, with no always-on compute and effectively no cost. A sensible cadence is every few minutes. Each run does one simple query: find the PENDING bookings whose deposit deadline is either approaching or already past. Those split into two lanes. A booking whose deadline is close but not yet reached, and which hasn't already been reminded, gets a single reminder — "your slot's held until 6pm today, here's the link" — and is marked reminded so the next run doesn't nudge it again. One reminder, not a drip-feed, because a customer chased every five minutes cancels out of irritation. A booking whose deadline has passed with the deposit still unpaid goes to the release lane.

Releasing a hold is where the care goes, because it's the moment a slot goes back on sale and a deposit that arrives a second later must not collide with it. So the release is a **conditional write**: the booking is moved to RELEASED and its slot freed *only if the booking is still PENDING*. If the deposit cleared in the gap between the sweep's query and its write — the customer paying at 5:59 while the 6pm sweep runs — the booking is already CONFIRMED, the condition fails, and

the release does nothing. The slot stays sold, the customer keeps their table, and the sweep quietly moves on. And just as in Part 3, the state field means the release is idempotent: a booking is released exactly once, and re-running the sweep never releases it again or frees the same slot twice.



One reminder before the deadline, one release after — and a deposit that clears at the last second is never wrongly cancelled.

Fig 4. The release sweep. EventBridge Scheduler runs the sweep on a fixed cadence; a hold nearing its deadline gets one reminder and is marked reminded, while a hold past its deadline is released with a conditional write that fires only if it's still PENDING — freeing the slot without ever cancelling a last-second payment.

Why a sweep, and not a timer per booking

You could imagine scheduling a one-off release for each individual hold — set a timer for the deadline, fire it, check if the deposit was paid. It works, but it means a scheduled entity per booking to create, track, and tidy up, and a pile of near-simultaneous invocations when a busy evening's worth of deadlines all fall at 6pm. A single periodic sweep is simpler and cheaper: one rule, one function, one query that scales to whatever the day brings, and idempotent by design because the state field means re-running it never double-releases. The few-minute granularity is plenty — whether a slot frees at 6:00 or 6:04 makes no difference to whether it can be re-sold that evening.

The sweep also carries the system's guardrails. Its reminder honours quiet hours and the STOP opt-out, so nobody is nudged about a deposit at 2am or after they've asked not to be texted. It reminds each booking exactly once and releases it exactly once, so a customer can't be pestered and a slot can't be freed twice. And it touches only its own state — it reads PENDING bookings and writes the reminded flag or the RELEASED transition, nothing more — which keeps it safe to run unattended every few minutes, forever. When a hold is released, the freed slot simply reappears in the booking source's availability, ready for the next customer who'll actually turn up; and if the business wants to know, a released hold can drop a quiet line into the handover inbox so the front desk can offer that Saturday table to the waiting list.

DESIGN RULES THAT SHAPED RELEASE

- Watch the deadline, not just the payment. An unpaid hold is only visible to a job that runs on a clock; the sweep is that clock.
- A clock, not a timer per booking. One EventBridge Scheduler rule and one query beat a scheduled entity per hold — simpler and cheaper.
- Nudge once. A single reminder before the deadline, marked so the next run never sends another — a chased customer cancels.
- Release on a condition. The hold is freed only if the booking is still PENDING, so a last-second payment is never wrongly cancelled.
- Release once. The state field makes the sweep idempotent — a booking is freed exactly once and a slot never twice.
- The same guardrails apply. Quiet hours and the opt-out list bind the reminder just as they bind every other message.

PART 5 OF 7

JULY 5, 2026 PART 5 OF 7 · BOOKING DEPOSIT COLLECTOR SERIES ~8 MIN READ

How a no-show gets handled

The deposit only earns its keep on the day. This post is about settlement: how a customer who turns up has their deposit applied to the bill, how a no-show forfeits it per the policy they agreed to, and why that decision is a plain rule in code — never a judgement call a model makes.

KEY TAKEAWAYS

- The deposit earns its keep on the day: a customer who turns up has it applied to their bill; a no-show forfeits it, per the policy they agreed to.
- Attendance is a fact the business records — a tap from front-of-house — not something a model or the system guesses.
- A settlement sweep after the appointment closes each confirmed booking exactly once: APPLIED for an arrival, FORFEITED for a no-show.
- The apply-or-forfeit decision is a plain rule from the settings doc — deterministic policy, never a judgement call by Bedrock.
- Every outcome is recorded with the amount and the reason, and anything contested is handed to a person with the full history.

The deposit only matters on the day

Everything so far — the hold, the link, the confirmation, the release — exists to get a real, committed booking onto the calendar. But the deposit itself only does its final job on the day of the appointment, and it does one of two things. If the customer turns up, the deposit is theirs: it comes off the final bill, exactly as the message promised, so it never feels like a charge, just money paid early. If the customer doesn't turn up, the deposit is forfeit — that's the whole point of taking it, and it's what makes the no-show cost the customer something instead of costing the business the slot. Settlement is the step that carries out whichever of those two the day actually delivers.

The important design choice here is that *attendance is a recorded fact, not an inference*. The system does not try to work out whether someone came — it has no way to, and guessing about money is exactly the wrong instinct. Instead, front-of-house tells it: a tap on the booking ("seated", "arrived", "started") posted back through the intake surface marks the booking attended. A booking that reaches the end of its appointment window with no such mark is, by definition, a no-show. Making attendance an explicit signal keeps the settlement honest and auditable: there is always a clear reason on the record for why a deposit was applied or forfeited.

A rule, not a judgement

How a deposit settles is set once, in the policy, and applied the same way every time. The settings doc holds the business's no-show terms — the ones the customer agreed to when they paid: apply the deposit to the bill on attendance,

forfeit it in full on a no-show, or a softer variant like a grace window or a partial forfeit for a late cancellation. Whatever the rule, it's **deterministic Python**. The settlement step reads the booking's outcome (attended or not, cancelled or not, and when), looks up the matching policy branch, and does exactly what it says. This is the sharpest example of the whole system's division of labour: Bedrock phrases the messages around all of this, but it is nowhere near the decision to keep or return someone's money. A model must never decide a forfeit, because a forfeit is a financial action a business has to be able to defend, and "the AI decided" is not a defence.

A settlement sweep — the same EventBridge Scheduler pattern as Part 4 — runs after appointment windows close and settles each confirmed booking once. For an arrival, it marks the booking APPLIED: the deposit is credited against the bill (or simply noted as already taken, depending on how the till works) and a short thank-you can go out. For a no-show, it marks the booking FORFEITED, records the amount and the reason, and — because a forfeited deposit is the one message most likely to prompt a reply — sends the customer the plain, policy-worded note they agreed to, and drops a line into the handover inbox so the business has a record and a chance to soften it for a good regular. The state field makes this exactly-once, just like every other transition: a booking settles a single time and can't be both applied and forfeited.

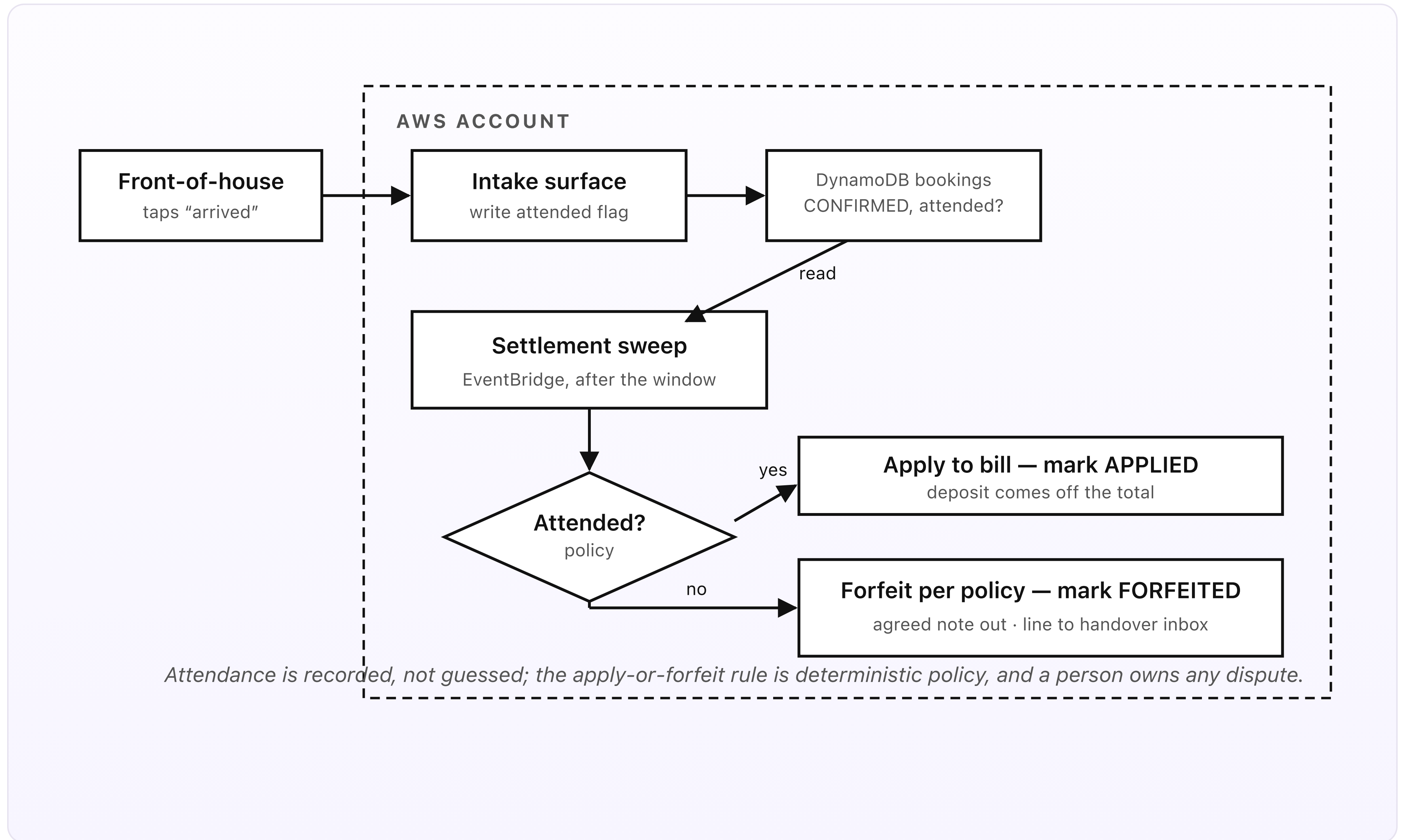


Fig 5. Settlement on the day. Front-of-house records the arrival through the intake surface; a settlement sweep reads confirmed bookings past their window and, by the policy rule alone, either applies the deposit to the bill (APPLIED) or forfeits it (FORFEITED)

— each booking settled exactly once, with any dispute handed to a person.

The human side of a forfeit

A forfeited deposit is the one outcome most likely to end in a conversation, so the system is built to hand that conversation to a person rather than have the last word itself. When a deposit is forfeited, the customer gets the plain note they agreed to — no arguing, no automated back-and-forth — and the business gets a line in the handover inbox with the whole history: when they booked, that the deposit was paid and confirmed, that the slot went unused, and the exact policy applied. If the customer replies "I was in hospital" or "I cancelled hours ago, check your messages", that reply goes straight to a human, who can waive or refund the forfeit with a click. The system enforces the policy consistently; a person retains the discretion to be kind. That balance — firm by default, human by exception — is what lets a small business take deposits without feeling like it's treating its regulars as suspects.

The same discretion covers the genuinely awkward cases. A double-booking that somehow slipped through (it shouldn't, given Part 2, but belt and braces), a customer who paid but whose slot was cancelled by the business itself, a partial-refund request — none of these are settled by rule, because none of them should be. They're recorded and escalated, with the full booking and payment history attached, for someone to resolve and, where needed, issue a refund through the payment provider by hand. The system's job ends at making the routine outcomes automatic, correct, and defensible; the exceptions are exactly where a human belongs.

DESIGN RULES THAT SHAPED SETTLEMENT

- Attendance is recorded, not inferred. Front-of-house marks the arrival; a booking with no mark past its window is a no-show by definition.
- Policy decides, code executes. Apply-or-forfeit is a deterministic rule from the settings doc — never a judgement a model makes.
- Settle exactly once. The state field means a booking is either APPLIED or FORFEITED, once, and never both.
- The deposit is the customer’s until it isn’t. On arrival it comes off the bill exactly as promised; forfeit is only the agreed no-show case.
- Firm by default, human by exception. The policy note is plain and consistent; a person can waive, refund, or soften any forfeit.
- Every outcome is defensible. Amount, reason, and policy are recorded, so a business can always explain why a deposit was kept or applied.

PART 6 OF 7

JULY 5, 2026 PART 6 OF 7 · BOOKING DEPOSIT COLLECTOR SERIES ~6 MIN READ

What the booking deposit collector costs

A system that costs more than the no-shows it prevents is a toy. This post is the cost breakdown: every AWS service this design touches, what each adds up to at around 120 bookings a month, why the total lands near $2.40 — and why the deposit processing fee lives on the payment provider's invoice, not this one.

KEY TAKEAWAYS

- About $2.40/month at roughly 120 bookings, and the fixed cost is almost nothing — nothing runs while the calendar is quiet.
- The two biggest lines are the deposit-request and reminder texts and the small Bedrock calls that phrase them. Everything else is cents.
- The only real fixed cost is Secrets Manager: two secrets — the payment signing key and the SMS key — at $0.40 each.
- The deposit processing fee isn't on this bill at all — it's a percentage the payment provider takes on their own invoice.
- At ten times the volume (around 1,200 bookings) the AWS bill lands near $15 — it scales with use, not with idle time.

Where the money goes

The system is serverless end to end, so there's no instance ticking over when the business is shut and no idle bill. You pay for a booking only when one happens. At a typical volume — call it 120 bookings a month, each getting a deposit request, most getting a reminder or a confirmation, a few settling as no-shows — here's the whole AWS bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two secrets — payment webhook signing key, SMS/provider API key ($0.40 each)	$0.80
SNS (SMS)	Deposit request, one reminder, and confirmation/settlement texts (~2 per booking)	$0.75
Bedrock (Claude Haiku 4.5)	One compose call per message — request, reminder, confirmation, forfeit note	$0.50
DynamoDB (on-demand)	Bookings, slots, payments, opt-out, audit — small conditional reads and writes	$0.12
CloudWatch Logs	Function logs, 7-day retention	$0.10
SES	Handover and dispute email to your team	$0.05
Lambda (Python 3.14, arm64)	Intake, payment webhook, composer, release sweep, settlement sweep	$0.05

AWS service	What it does here	Monthly
SQS + DLQ	Buffering between the webhooks and the slower model and SMS calls	$0.02
EventBridge Scheduler	The release sweep and the settlement sweep	$0.01
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~120 bookings/month**	**$2.40**

The shape of that bill is the point. The only line that costs money while the system sleeps is Secrets Manager — two secrets at $0.40 each, $0.80 a month no matter what, which is a third of the total at this volume. Everything else is genuinely usage-priced and rounds to zero at idle. The two lines that move with volume are the ones that actually reach the customer: the messages themselves and the small Haiku calls that phrase them. All the machinery doing the real work — holding slots, confirming payments, releasing unpaid holds, settling on the day — is plain Lambda and DynamoDB, and together it costs less than the SMS line alone.

The line that isn't on this bill

Two caveats belong on the SMS and payment lines. First, AWS prices outbound SMS per message, and the exact rate depends on the destination country and carrier — a UK mobile is a few pence, other countries differ, and some routes add

surcharges. The $0.75 here is a UK-leaning estimate; your real number will track your country and provider, and if your booking or telephony tool sends the texts instead of SNS, that cost moves onto their invoice and off this table. Second, and more importantly for this system: **the deposit processing fee is not an AWS cost at all**. When a customer pays a £30 deposit, the payment provider takes their cut — typically a small percentage plus a fixed few pence — on *their* statement, not here. That fee is the genuine cost of collecting a deposit, and it's the same handful of pence per transaction whether or not this system exists. What this system adds on top is the near-nothing you see above.

What ten times the volume costs

Push this to a busy business — 1,200 bookings a month, ten times the volume — and the AWS bill lands near $15, not $24. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules stay at a cent, and AWS Budgets stays free. What scales is the genuinely usage-priced work — roughly $7 of SMS for ten times the texts, about $5 of Bedrock, and a few dollars more spread across DynamoDB, logs, SES, and Lambda. Even then, the two things that reach the customer dominate, and all the machinery in between stays close to free. (The payment provider's fee scales too, of course — but it always sits on their invoice, and it's dwarfed by the deposits it's protecting.)

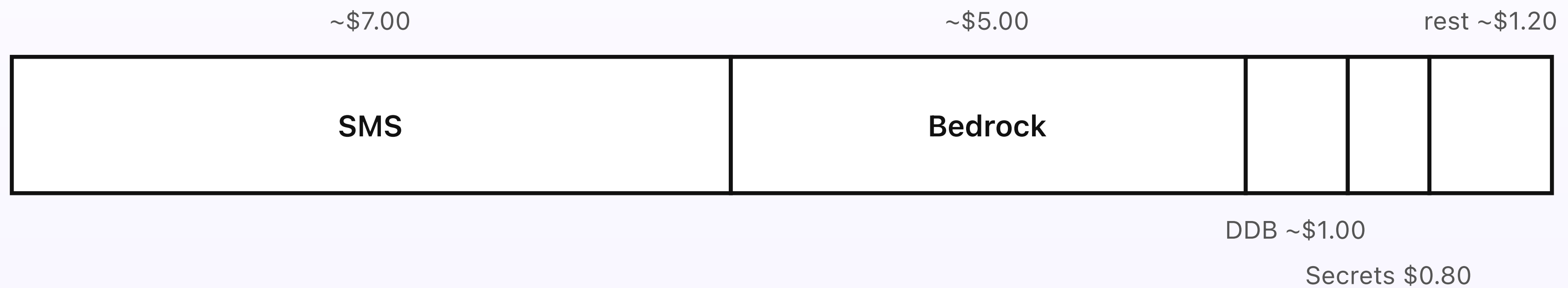


Fig 6. The monthly AWS bill at ten times the base volume, about 1,200 bookings. SMS and Bedrock are the bulk of it; the fixed lines stay put, so the total grows sub-linearly — near $15, not ten times $2.40 — and the payment processing fee lives on the provider's invoice, not here.

The honest way to read this: the AWS bill is rounding error against what a no-show is worth. A single empty table on a Saturday night, a missed two-hour tattoo sitting, a blocked grooming slot — any one of those is worth far more than $2.40, and this design protects them by the dozen. Even at $15 a month for a busy business, the system pays for itself the first time it turns a would-be no-show into

a paid, confirmed booking — or frees a dead slot early enough to sell it to someone from the waiting list.

DESIGN RULES THAT SHAPED THE COST

- Pay per booking, not per hour. No always-on compute means no idle bill.
- Spend the model sparingly. One Haiku call per message, and only to phrase — never to decide, confirm, or settle.
- Cheap work stays cheap. Holding, confirming, releasing, and settling are plain Lambda and DynamoDB, cents at this scale.
- Know your one fixed cost. Secrets Manager is the only line that bills while the system sleeps.
- The deposit fee isn't yours to host. Processing sits on the provider's invoice; the Budgets alarm watches the AWS side, with SMS the line to watch.

PART 7 OF 7

JULY 5, 2026 PART 7 OF 7 · BOOKING DEPOSIT COLLECTOR SERIES ~10 MIN READ

Engineering reference: the booking deposit collector architecture

This is the booking deposit collector with the friendly labels removed: the real resource names, the two public Function URLs, the table key schemas, the two conditional writes that keep money and the calendar honest, and the IAM scope. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Six Lambda functions, all Python 3.14 on arm64, wired through one SQS queue with a dead-letter queue.
- Two public surfaces: `bdc-intake` for bookings and `bdc-webhook` for the payment provider — each its own Lambda Function URL, no API Gateway.
- Five DynamoDB tables, all on-demand: bookings, slots, a payment-event ledger, an opt-out list, and an append-only audit log.
- Exactly-once rests on two conditional writes: `attribute_not_exists` on the slot to hold it, and a state condition to confirm, release, and settle a booking once each.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by the composer. Single region, `eu-west-2`.

The architecture, for engineers

This is the same system as Part 1 with the friendly labels removed and the real resources named. Everything is in one region, `eu-west-2` (London), in one account. There is no API Gateway, no NAT Gateway, and nothing always-on; the only inbound surfaces are two Lambda Function URLs — one for the booking source, one for the payment webhook — outbound messaging goes through SNS and SES, and work is buffered on a single SQS queue.

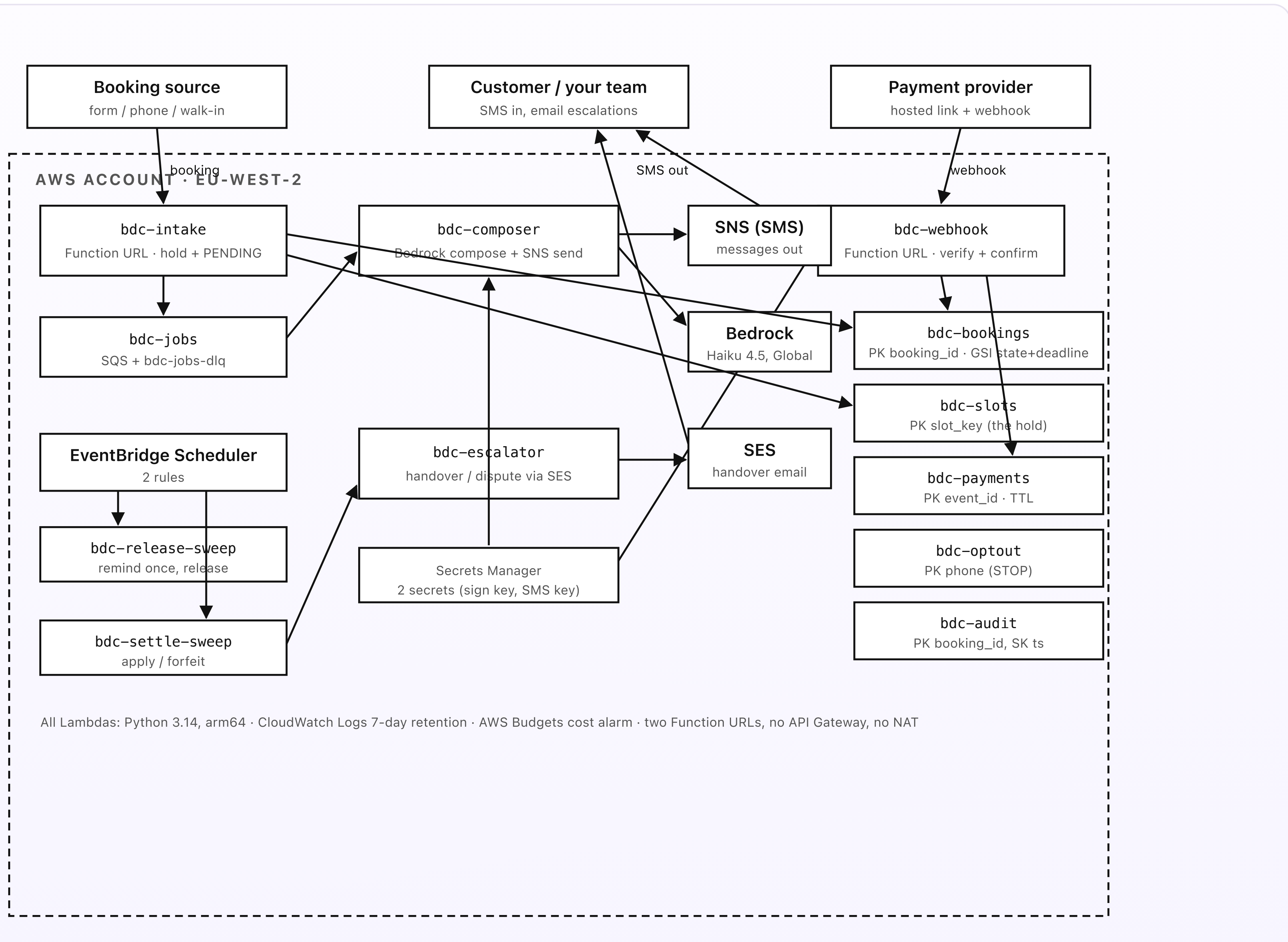

Booking source
form / phone / walk-in
Customer / your team
SMS in, email escalations
Payment provider
hosted link + webhook
AWS ACCOUNT · EU-WEST-2
booking
SMS out
webhook
bdc-intake
Function URL · hold + PENDING
bdc-composer
Bedrock compose + SNS send
SNS (SMS)
messages out
bdc-webhook
Function URL · verify + confirm
bdc-jobs
SQS + bdc-jobs-dlq
Bedrock
Haiku 4.5, Global
bdc-bookings
PK booking_id · GSI state+deadline
bdc-slots
PK slot_key (the hold)
EventBridge Scheduler
2 rules
bdc-escalator
handover / dispute via SES
SES
handover email
bdc-payments
PK event_id · TTL
bdc-release-sweep
remind once, release
Secrets Manager
2 secrets (sign key, SMS key)
bdc-optout
PK phone (STOP)
bdc-settle-sweep
apply / forfeit
bdc-audit
PK booking_id, SK ts
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm · two Function URLs, no API Gateway, no NAT

Fig 7. The booking deposit collector drawn for engineers: two Function URLs (`bdc-intake`, `bdc-webhook`), an SQS-buffered set of six Lambdas, five DynamoDB tables, Bedrock called only by the composer, SNS for SMS and SES for handover, and two scheduled sweeps. One region, one account, no API Gateway.

Lambda functions

Six functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. Each does one job and hands off; the SQS queue (`bdc-jobs`, with `bdc-jobs-dlq` as its dead-letter queue after five attempts) decouples the two public webhooks from the slower model and SMS calls.

- `bdc-intake` — the booking surface. Backs the first Lambda Function URL. Holds the slot in `bdc-slots` with a conditional write, writes the booking PENDING with its deposit deadline to `bdc-bookings`, and enqueues a compose job. It also takes the on-the-day attendance mark from front-of-house and writes the `attended` flag. Nothing slow happens here.
- `bdc-webhook` — the money surface. Backs the second Function URL. Verifies the provider's signature against the Secrets Manager signing key, records the event id in `bdc-payments` with a conditional write, and flips the booking PENDING → CONFIRMED with a conditional write — then enqueues a confirmation message. It cannot create bookings or hold slots.
- `bdc-composer` — SQS-triggered on message jobs. Makes the single Bedrock call, validates the draft, injects the payment or booking link, checks the opt-out list, sends via SNS, and writes `bdc-audit`. The only function that can call Bedrock or publish SMS.

- `bdc-release-sweep` — scheduled. Queries `bdc-bookings` by the state-and-deadline index for PENDING holds near or past their deadline; sends one reminder (marking the booking `reminded`) and, past the deadline, releases the hold with a conditional write on state `== PENDING`, freeing the `bdc-slots` item and marking the booking RELEASED.
- `bdc-settle-sweep` — scheduled. After the appointment window, reads CONFIRMED bookings and, by the deterministic policy, marks each APPLIED or FORFEITED once with a conditional write on state `== CONFIRMED`, handing disputes and mismatches to the escalator.
- `bdc-escalator` — builds the handover (booking + payment + outcome + reason), de-duplicates against open cases, and emails the team via SES for anything a person must own: a forfeit dispute, a payment mismatch, an unknown reference.

The data model

Five DynamoDB tables, all on-demand, and the schema is where the exactly-once guarantees live.

- `bdc-bookings` — PK `booking_id`. One item per booking with its `state` (`PENDING` / `CONFIRMED` / `RELEASED` / `APPLIED` / `FORFEITED`), the `slot_key` it holds, the deposit `amount`, the `deposit_deadline`, the `reminded` and `attended` flags, and the payment reference. A global secondary index on `state` + `deposit_deadline` is what lets both sweeps find exactly the bookings they care about — PENDING-past-deadline for release, CONFIRMED-past-window for settlement — without scanning the table.

- `bdc-slots` — PK `slot_key` (for example `resource#2026-07-11T20:00`). One item per *held* slot; this table is the double-booking guard. The hold is a `PutItem` with `attribute_not_exists(slot_key)`, so only the first booking for a slot can write it. Releasing a hold deletes the item, returning the slot to sale.
- `bdc-payments` — PK `event_id`. The idempotency ledger for the payment webhook: each incoming event is recorded with `attribute_not_exists(event_id)`, so a re-delivery finds it present and stops. A TTL expires old records, since a webhook is never re-sent after a day or two.
- `bdc-optout` — PK `phone` (E.164). The STOP suppression list, checked before every message the composer or a sweep sends.
- `bdc-audit` — PK `booking_id`, SK `ts`, append-only. Every message sent and every state transition, with the facts it was built from — the record that makes a forfeit or a confirmation defensible after the fact.

Exactly-once, and never twice

Two conditional writes carry the whole system's correctness, and everything else is downstream of them. The first is the **slot hold**: `attribute_not_exists(slot_key)` on `bdc-slots` means a slot can be held by exactly one booking, however many arrive at once — the database serialises the race, so there is no double-booking. The second is the **state transition** on `bdc-bookings`: every move is conditional on the current state, so `bdc-webhook` confirms only a booking that is still PENDING, `bdc-release-sweep` releases only one still PENDING, and `bdc-settle-sweep` settles only one still CONFIRMED. A duplicate payment webhook, an overlapping sweep, a retried SQS message — each finds the booking already past that state and does nothing. The payment

ledger's `attribute_not_exists(event_id)` is a belt-and-braces first line, but the state condition is the real backstop, because it makes every transition in the system idempotent by construction rather than by luck.

The SQS queue between the webhooks and the composer means a slow Bedrock or SNS call never blocks a webhook, and a transient failure is retried up to five times before the message lands in `bdc-jobs-dlq` for inspection. Because sends are keyed and audited, a redelivered message doesn't double-text: the composer records what it has sent, and the state conditions upstream mean there was only ever one job to begin with.

IAM scope, observability, and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `bdc-intake` can conditionally write `bdc-slots`, write `bdc-bookings`, read `bdc-optout`, and send to `bdc-jobs` — it cannot call Bedrock, SNS, or confirm a payment. `bdc-webhook` can read the signing secret, conditionally write `bdc-payments`, conditionally update `bdc-bookings`, and send to `bdc-jobs` — and nothing else; it is the only role that can move a booking to CONFIRMED, and it cannot create one. `bdc-composer` is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile, plus SNS publish and read access to the SMS secret; it cannot delete from any table. The two sweeps hold the narrow query-and-conditional-update permissions they need on `bdc-bookings` and `bdc-slots` and no inbound surface at all. `bdc-escalator` can send via SES and read its inputs only.

Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call for capacity and is not a data store. CloudWatch Logs hold each function's logs at 7-day retention, and an AWS Budgets alarm watches the monthly spend — with SMS the line most likely to move, it's the cheapest early warning that volume, or a loop, is running hot. The whole stack is one region, one account, and defined as infrastructure-as-code: the two Function URLs, the six functions, the five tables and their index, the queue and DLQ, the two schedules, the SNS and SES configuration, the secrets, and the Budgets alarm all come up from a single template, so the system is reproducible and reviewable rather than clicked together by hand.

That's the whole thing: a booking becomes a held slot and a deposit request, a signed webhook confirms it exactly once, an unpaid hold is reminded once and released, and on the day the deposit is applied or forfeited by policy — all on a few dollars a month of serverless AWS, with a model nowhere near the money and a human owning every exception.