7-PART SERIES · FREE COMPANION

Dunning recovery

A subscription payment fails, and most small businesses find out weeks later — once the customer has quietly churned and the revenue is gone. This is a small serverless system that catches the failed charge the moment the payment processor reports it, works through a smart retry schedule that backs off over several days and skips weekends, and sends a branded dunning email at each attempt. If every retry fails it pauses the customer's access, and it restores access the instant a retry succeeds or the customer updates their card. It never double-charges and it never silently cancels anyone. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Dunning recovery

PART 1 OF 7

JUNE 21, 2026 PART 1 OF 7 · DUNNING RECOVERY SERIES ~11 MIN READ

A dunning recovery system on AWS for a few dollars a month

A subscription business loses more revenue to failed payments than to customers who actually decide to leave. A card expires, a charge is declined, the renewal silently fails — and weeks later someone notices the account is still active but hasn't paid since March. Recovering those payments by hand is fiddly and easy to drop: you have to retry the charge at sensible intervals, email the customer without nagging, and pause access only once you've genuinely given up — never double-charging and never cutting someone off by mistake. This post walks through the design of a small system that catches every failed charge, runs a smart retry schedule, sends a branded dunning email at each step, and pauses or restores access on its own — while never moving money without a guard.

KEY TAKEAWAYS

- One trigger starts everything: a failed-charge webhook from your payment processor (Stripe-style).
- The retry schedule is plain Python — a backoff over several days that skips weekends, four retries in about ten days.
- A branded dunning email goes out at every attempt, each with a one-tap link to update the card.
- Access pauses only after the final failed retry, and restores the instant a retry succeeds or the card is updated.
- Designed on AWS for about $1.90/month at small-business volume. It never double-charges and never silently cancels.

The whole system on one page

Before any code, here's the shape of what we're designing. There are three things outside AWS the system talks to, and three pieces inside that do the work.

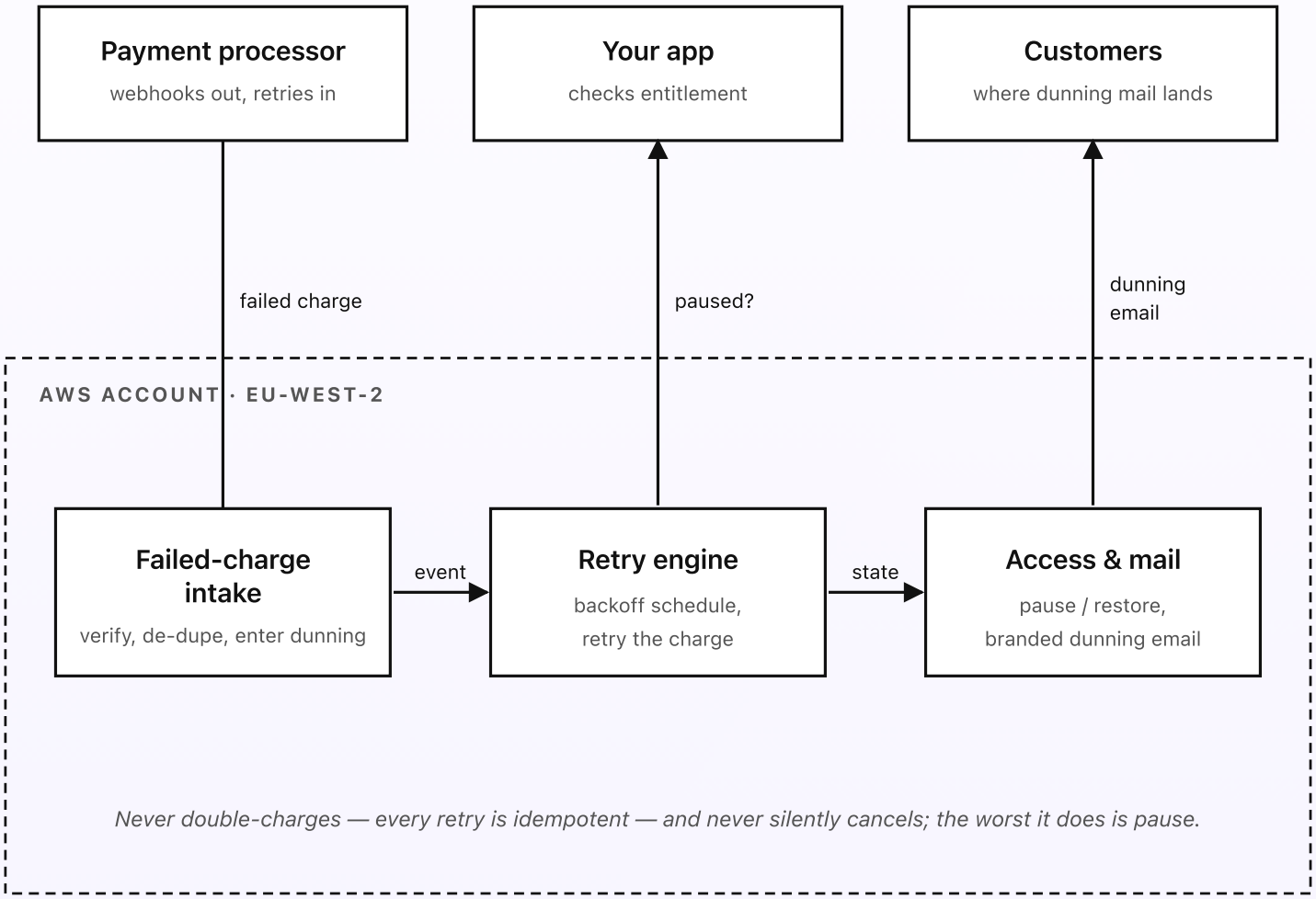

Payment processor
webhooks out, retries in
Your app
checks entitlement
Customers
where dunning mail lands
failed charge
paused?
dunning email
AWS ACCOUNT · EU-WEST-2
Failed-charge intake
verify, de-dupe, enter dunning
event
Retry engine
backoff schedule, retry the charge
state
Access & mail
pause / restore, branded dunning email
Never double-charges — every retry is idempotent — and never silently cancels; the worst it does is pause.

Fig 1. Three things outside, three pieces inside AWS. A failed-charge webhook flows in from the processor. The retry engine lays out a backoff schedule and retries the charge at each step. The access-and-mail piece sends a branded dunning email and pauses or restores access — on its own, but reversibly.

What you set up once (the outside)

- **The payment processor.** You already use a Stripe-style processor to take subscription payments. Two things connect it to this system. First, a *webhook endpoint* — you point the processor at a Lambda Function URL and subscribe to the events that matter: a charge failed, a charge succeeded, a payment method was updated, a subscription changed. Second, a *secret API key* (in Secrets Manager) the system uses to call back and retry a charge. You do not store card numbers anywhere; the processor holds the card, and the system only ever asks it to try the saved card again.
- **Your app's entitlement check.** Somewhere your product already decides "is this customer allowed in?" — on login, on an API call, on a page load. You change that one check to ask the gatekeeper (Part 5) instead of reading a flag directly. That single change is what lets the system pause and restore access without touching the rest of your app. While a subscription is healthy the answer is always *granted*; the moment dunning gives up it becomes *paused*, and the customer sees an update-card screen instead of the product.
- **The email brand.** One sender identity verified in SES (with DKIM and SPF), your logo, and a short HTML template in S3. The dunning emails go out from `billing@your-company.com` looking like you, not like a generic decline notice. You set the tone once — the copy at each attempt is written from that template — and you decide how many retries to run before access pauses.

What runs on every failed charge (the inside)

- **The failed-charge intake.** The processor fires a webhook the instant a renewal charge fails. `dunr-webhook`, behind a Lambda Function URL, verifies the signature against the webhook secret so it knows the event is genuine, throws the event away if it has seen its id before (processors retry their own webhooks, so duplicates are normal), and writes the subscription into the `dunr-dunning` table in a *retrying* state. It does not trust any amount the request claims; it records the subscription and invoice ids and lets the retry engine read the real figures back from the processor. Then it drops a message on an SQS queue and returns `200` fast, so the processor never sees a slow endpoint. Part 2 is all about this front door.
- **The retry engine.** A scheduler Lambda reads the queue and lays out the retry schedule in plain Python: the next business day, then roughly every three business days, four retries over about ten days, skipping weekends. Each step becomes a one-off EventBridge Scheduler rule that fires `dunr-retrier` at the exact minute. The retrier asks the processor to charge the saved card again — guarded by an idempotency key tied to the invoice and attempt number, so a duplicated trigger can never charge twice. A success cancels every remaining retry and marks the subscription *recovered*; a failure advances the attempt and, if it was the last one, hands off to the gatekeeper to pause access. Parts 3 covers the schedule.
- **Access and mail.** Two small pieces close the loop. `dunr-mailer` sends a branded dunning email at each attempt (Part 4) — one Bedrock Haiku call writes the copy in your voice, gentle on the first nudge and clear about the consequence by the last, each with a one-tap update-card link. `dunr-gatekeeper` is the entitlement check your app calls (Part 5); it pauses access only after the final failed retry and restores it the instant a retry succeeds or the customer updates their card. Every attempt, email, and state change is written to the `dunr-log` audit table.

The five states every subscription moves through

A subscription in this system is always in exactly one state, and the transitions are deterministic — no model decides them.

- **Active.** Paying normally. The system isn't doing anything; the gatekeeper answers *granted*.
- **Retrying.** A charge has failed and the backoff schedule is running. Access is still *granted* — you don't lock someone out the second a card blips. Each retry sends a dunning email.
- **Paused.** Every retry failed. Access is now *paused*; the customer sees the update-card screen. This is reversible, and it is the strongest action the system takes on its own.
- **Recovered.** A retry succeeded, or the customer updated their card and the next charge cleared. Remaining retries are cancelled, access is restored in the same second, and a short "you're all set" receipt goes out.
- **Cancelled.** Reached only by a human, in your billing tool. The system never cancels a subscription on its own; pausing is as far as it goes.

In plain words

A customer on your £29/month plan renews on a Tuesday and the charge declines — expired card. Within a second the processor fires a `charge.failed` webhook; `dunr-webhook` verifies it and puts the subscription into *retrying*. The scheduler lays out four retries: Wednesday, the following Monday, Thursday, and the Monday after — all weekdays. Wednesday's email goes out: "We couldn't process your payment — no need to worry, we'll try again, or you can update your card here." The Wednesday and Monday retries both fail; the customer is busy and ignores

the emails. Thursday's email is firmer: "Your access will pause on Monday if we can't take payment." On Saturday the customer finally taps the link and updates their card. The gatekeeper sees the new card, the next retry clears on the spot, the two remaining scheduled retries are cancelled, access stays on, and a receipt confirms it. The customer never lost access, you never lost the £29, and nobody on your team touched it.

The cost of running this is about $1.90 a month at small-business volume. The cost of *not* running it is the renewals that fail and never get chased — the quiet, compounding leak that subscription businesses call involuntary churn.

DESIGN RULES THAT SHAPED EVERY DECISION

- One trigger, verified. Everything starts from a signed failed-charge webhook; an unsigned or replayed request does nothing.
- The schedule is plain Python. A backoff over business days — no model decides when to retry.
- Every retry is idempotent. An idempotency key on the invoice-and-attempt means a duplicate trigger can never double-charge.
- Pause, never cancel. The strongest action the system takes on its own is a reversible pause; cancelling is always a human decision.
- Restore is instant. A successful retry or an updated card flips access back in the same second — no overnight batch.
- Everything is logged. Every attempt, email, and state change writes to an audit table you can read a year later.

Why this shape

Most small subscription businesses handle failed payments one of three ways: the processor's built-in dunning runs with default settings nobody tuned, a founder manually emails people when they notice, or nothing happens and the customer keeps their access for free until someone audits the books. The default-settings approach often retries on weekends and sends generic decline emails that look like phishing. The manual approach doesn't scale past a few dozen customers. And doing nothing is how a subscription business slowly gives its product away.

The setup above keeps the processor as the source of truth for whether a charge succeeded — the system never guesses — and adds a small layer that *reacts* to failures with a sensible, branded, fully logged recovery flow. Retries happen on weekdays at humane intervals. Emails look like you and escalate gently. Access pauses only when recovery has genuinely failed, and comes back the instant the customer fixes their card. And because the schedule and the pause are plain deterministic Python guarded by an idempotency key, the two things you most fear — charging someone twice and cutting someone off by mistake — simply can't happen.

The next four posts walk through each piece in turn: how a failed charge gets caught, how a retry schedule gets decided, how a dunning email gets sent, and how access gets paused and restored. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 21, 2026 PART 2 OF 7 · DUNNING RECOVERY SERIES ~8 MIN READ

How a failed charge gets caught

Everything downstream depends on hearing about the failed charge promptly and exactly once. This post is about the front door: a Stripe-style webhook lands on a Lambda Function URL, the system verifies it really came from the processor, throws away duplicates and replays, and records that this subscription has entered dunning — without trusting anything the request claims about money.

KEY TAKEAWAYS

- The processor fires a signed webhook the instant a renewal charge fails; a Lambda Function URL catches it.
- The signature is verified against the webhook secret before anything is trusted — an unsigned request does nothing.
- Events are de-duplicated by id, because processors retry their own webhooks and replays are normal.
- The handler trusts no amount in the request; it records ids and lets the retrier read real figures back from the processor.
- It returns `200` in milliseconds and hands the work to SQS, so the processor never sees a slow endpoint.

The front door is a webhook, not a poll

The system finds out about a failed charge the same way everything else in a Stripe-style processor finds out: a webhook. When a subscription's renewal charge is declined, the processor immediately fires an event — `invoice.payment_failed`, or your processor's equivalent — to an HTTPS endpoint you registered. The alternative, polling the processor every few minutes asking "anything failed yet?", would be slower, cost more, and miss the exact moment. So the front door is a single endpoint that listens.

That endpoint is a **Lambda Function URL** — a plain HTTPS address that invokes `dunr-webhook` directly. No API Gateway sits in front of it; for one webhook receiver, the Function URL does the whole job and costs nothing per request. The processor is configured to send the four events that matter: a charge failed, a charge succeeded, a payment method was attached or updated, and a subscription was cancelled in the billing tool. Everything the system does begins here.

Three checks before anything is trusted

A public URL that puts customers into a payment-recovery flow is exactly the kind of thing you don't want fired by a random request. So `dunr-webhook` does three things, in order, before it believes a word of the payload.

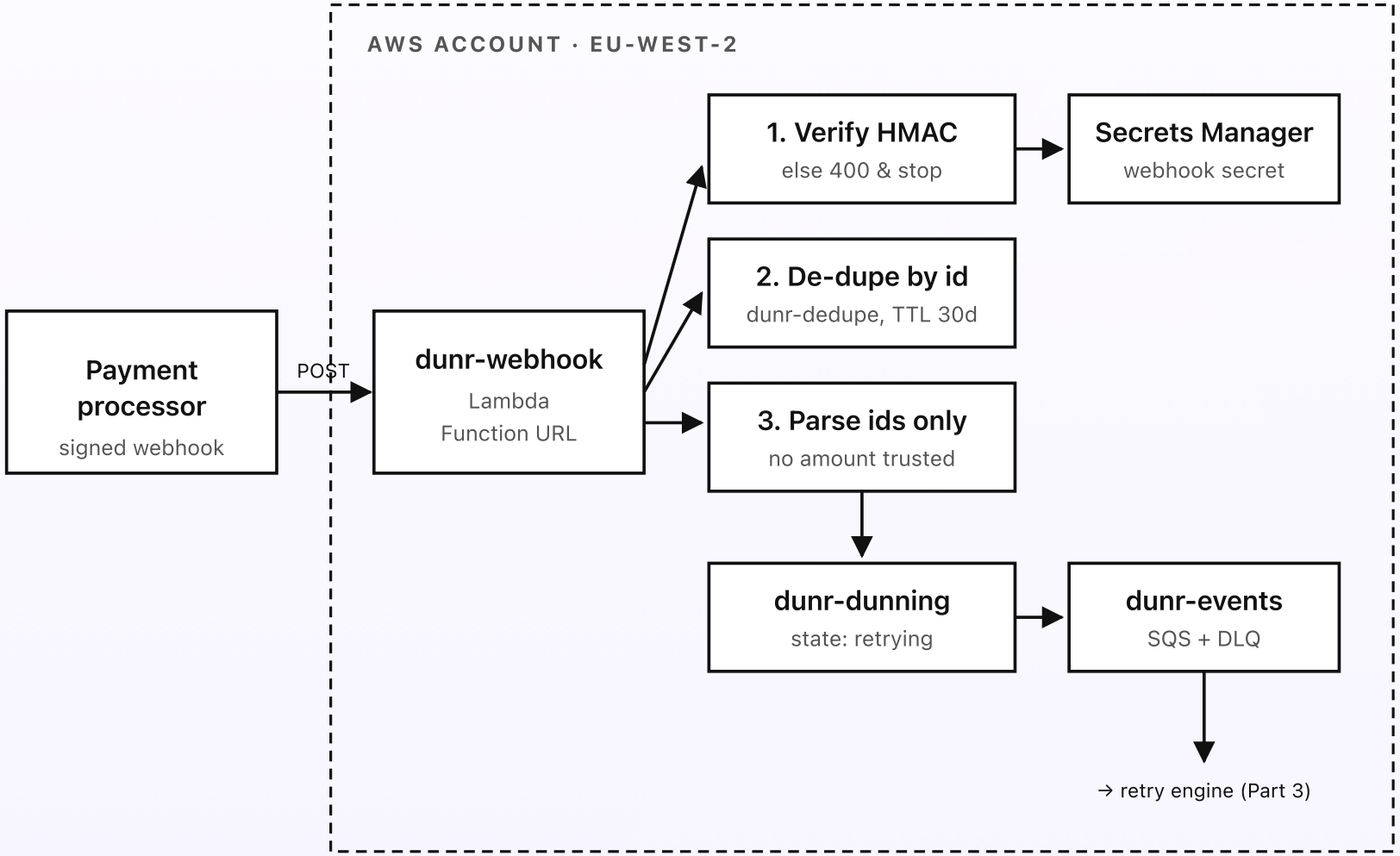
- **Verify the signature.** The processor signs every webhook with a shared secret and puts the signature in a header. The handler reads the webhook signing secret from Secrets Manager (`dunr/stripe/webhook-secret`), recomputes the HMAC over the raw request body and the timestamp, and compares. If it doesn't match — or the timestamp is older than five minutes, which blocks

replay of a captured request — the handler returns `400` and stops. An attacker who doesn't have the secret cannot make this system do anything.

- **De-duplicate by event id.** Processors guarantee *at-least-once* webhook delivery, which means they will happily send the same event twice if they don't get a fast `200`. Every event carries a unique id. The handler does a conditional write of that id to a small `dunr-dedupe` table (with a 30-day TTL); if the id is already there, this is a duplicate and the handler returns `200` immediately without doing anything again. This is the first line of defence against acting on the same failure twice.
- **Trust ids, not amounts.** The handler reads only the identifiers from the payload — the subscription id, the invoice id, the customer id, and the failure reason code. It deliberately does *not* trust any amount or currency in the request body as gospel; when the retrier later attempts the charge, it reads the live invoice from the processor using its API key, so the figure charged is always the processor's real number, not whatever arrived in a webhook.

Entering dunning, then handing off

Once an event is verified, fresh, and parsed, the handler does the smallest possible amount of work and gets out of the way.



Verify, de-duplicate, then hand off fast — the processor never waits on a slow endpoint.

Fig 2. The failed-charge front door. A signed webhook hits the Function URL; `dunr-webhook` verifies the signature, drops duplicates by id, parses only the identifiers, marks the subscription retrying, queues the work, and returns `200` in milliseconds.

It writes one row to the `dunr-dunning` table — keyed on the subscription id, marked *retrying*, recording the invoice id, the customer id, the failure code, and a timestamp — then drops a single message on the `dunr-events` SQS queue and returns `200`. The decision-making (how to retry, when, what to say) all happens downstream, off the back of the queue. The handler itself is deliberately dumb: catch, verify, record, hand off.

Returning `200` fast matters more than it looks. If the endpoint is slow, the processor treats the webhook as failed and retries it — which is why de-duplication exists — and a backlog of slow responses can get your endpoint disabled by the processor. By doing nothing heavy in the handler and pushing real work onto SQS, the front door responds in milliseconds however busy the retry engine is. The SQS queue also has a **dead-letter queue**: if a message fails to process twice, it lands there with an alarm, so a single malformed event never silently blocks the pipeline or gets lost.

The other three events

The same front door handles the good news too. A `charge.succeeded` for a subscription that's in dunning means a retry (or a fresh attempt after a card update) worked — the handler routes it to the recovery path that cancels remaining retries and restores access. A `payment_method.updated` means the customer fixed their card; the handler nudges the retrier to try again now rather than waiting for the next scheduled slot. And a `subscription.cancelled` — which only ever comes from a human acting in your billing tool — tells the system

to stand down: cancel the schedule, stop the emails, and mark the row closed. The system never originates a cancellation; it only reacts to one.

DESIGN RULES THAT SHAPED THE FRONT DOOR

- Verify before you trust. No signature, or a stale timestamp, means a `400` and nothing happens.
- Assume duplicates. At-least-once delivery is normal; de-duplicate by event id on a conditional write.
- Trust ids, not amounts. The real figure is read live from the processor at retry time, never from the webhook body.
- Answer fast, work later. Return `200` in milliseconds; push the real work onto SQS with a dead-letter queue.
- One door, four events. Failed, succeeded, card updated, cancelled — all arrive here and route from one place.

Next: how the retry engine takes that queued event and lays out a humane backoff schedule — the next business day, then every few business days, skipping weekends — and guarantees a charge is never attempted twice.

PART 3 OF 7

JUNE 21, 2026 PART 3 OF 7 · DUNNING RECOVERY SERIES ~7 MIN READ

How a retry schedule gets decided

A good dunning schedule is mostly restraint. Retry too fast and you burn attempts while the card is still frozen; retry too slow and the customer has moved on. This post is about how the system lays out a sensible backoff — the next business day, then every few business days, skipping weekends — turns each step into a one-off scheduled job, and makes absolutely sure a single charge is never attempted twice.

KEY TAKEAWAYS

- The schedule is plain Python: a first retry the next business day, then roughly every three business days — four retries over about ten days.
- Weekends are skipped, because a card frozen on Friday rarely clears on Saturday and a dunning email lands better midweek.
- Each step is a one-off EventBridge Scheduler rule that fires `dunr-retrier` at the exact minute, then deletes itself.
- An idempotency key on the invoice-and-attempt means a duplicated trigger can never charge a customer twice.
- A success cancels every remaining retry; the final failure hands off to pause access. No model decides any of it.

A good schedule is mostly restraint

The temptation with a failed payment is to hammer the card — retry every hour until it works. That's the wrong instinct. Most renewal failures are an expired card, a temporary hold, or an insufficient-funds decline that resolves on payday, not in the next sixty minutes. Retrying too fast burns your handful of attempts while the card is still frozen, and each failed attempt can ding your account's risk profile with the processor. Retrying too slowly is worse in the other direction — wait two weeks between attempts and the customer has forgotten they ever used you.

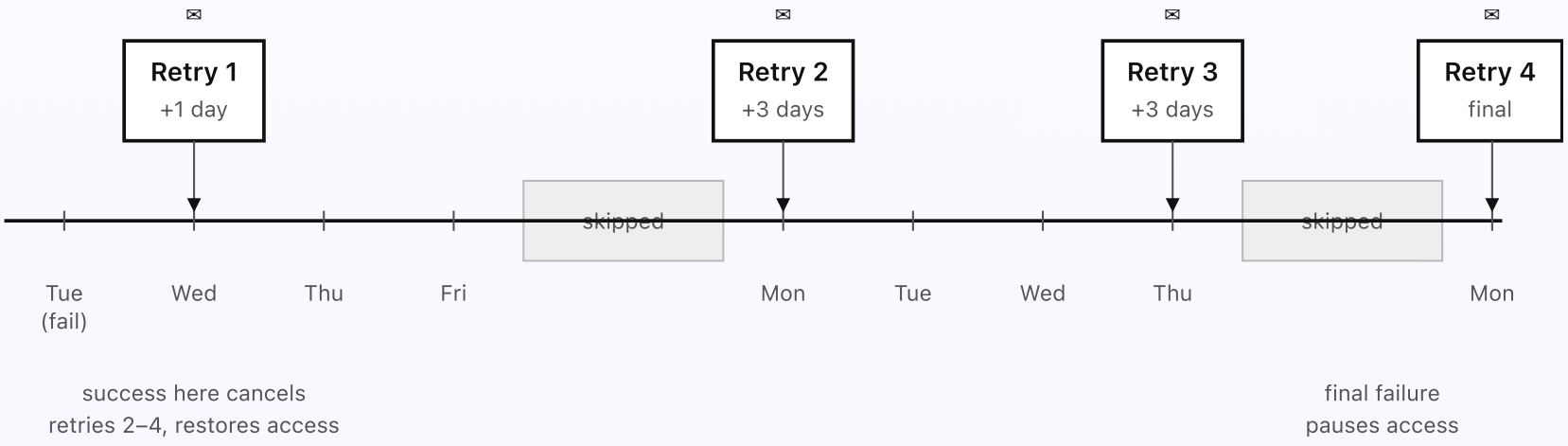
So the schedule is built around a simple, opinionated backoff that a person would recognise as reasonable: try again tomorrow, then give it a few days, then a few more, then one last try before pausing. Four retries spread over about ten days. Crucially, the schedule is plain Python doing date arithmetic — there is no model involved in deciding when to charge someone, because that is a decision that touches money and must be predictable.

The backoff, in business days

When `dunr-scheduler` reads a failed-charge message off the queue, it computes the retry dates from the failure date, skipping Saturdays and Sundays:

- **Retry 1 — next business day.** Gives a temporary hold or a same-day top-up time to clear, without waiting long.
- **Retry 2 — three business days later.** Catches the customer who needed a few days, and lands after the first dunning email has had time to be seen.
- **Retry 3 — three more business days.** The midpoint nudge, with a firmer email.
- **Retry 4 — three more business days (final).** The last automatic attempt. Its email warns that access will pause if this one fails.

Why skip weekends? Two reasons, both practical. A card that was declined for insufficient funds on a Friday is no more likely to clear on a Saturday — payday and bank processing run on weekdays. And a dunning email that arrives on a Sunday gets buried by Monday morning; a Tuesday email gets read. So a failure on a Tuesday produces retries on Wednesday, the following Monday, Thursday, and the Monday after — every one a weekday. The number of retries and the gaps are config, not code: you can run three retries over a week or five over three weeks by changing values, with no redeploy of logic.



Each retry is a one-off EventBridge Scheduler rule, fired at the exact minute and guarded by an idempotency key.

Fig 3. The backoff in business days. Four retries over about ten days, weekends skipped, an email at each. A success at any step cancels the rest; the final failure is what pauses access.

One job per retry, with EventBridge Scheduler

Each retry date becomes a **one-off EventBridge Scheduler** rule. The scheduler creates a rule with an `at(2026-06-24T09:00:00)` expression that targets `dunr-retrier` with a payload of the subscription id and the attempt number. EventBridge Scheduler fires it once, at that minute, and — with `--action-after-completion DELETE` — deletes the rule afterward, so there is no accumulating litter of stale schedules. This is the right tool here precisely because the work is sparse and far apart: there is no Lambda sitting awake waiting, no cron table to scan, nothing running between retries. Ten days can pass between a customer's first and last retry and the system uses no compute at all in between.

The schedule names encode what they are — `dunr-retry-{subscription}-{attempt}` — which makes the next part easy: when a retry succeeds, the retrier can cancel the remaining rules by name in one pass, so retries 2 through 4 simply never fire once retry 1 has recovered the payment.

The one guarantee that matters: never twice

EventBridge Scheduler, like most of AWS, is *at-least-once*: in rare cases it can fire the same schedule twice. A webhook can be replayed. A retry could overlap with a customer manually updating their card and triggering a fresh attempt. None of these may ever result in charging a customer twice. So every charge the retrier makes carries an **idempotency key** — a deterministic string built from the invoice id and the attempt number, for example `dunr-inv_8842-a3`. The processor treats two requests with the same idempotency key as the same request: the second one returns the result of the first without taking any money. The key is deterministic, not random, precisely so that a duplicate trigger reconstructs the *same* key and is harmlessly collapsed.

This is the load-bearing safety property of the whole system. Combined with the de-duplication at the front door (Part 2), it means there is no path — duplicate webhook, double-fired schedule, overlapping manual update — that charges a customer's card more than once for one attempt.

DESIGN RULES THAT SHAPED THE SCHEDULE

- Restraint over speed. Four retries over about ten days, not a barrage in the first hour.
- Business days only. Skip weekends — the card won't clear and the email won't get read.
- Plain Python, no model. A decision that moves money is deterministic and predictable.
- One job per retry. One-off EventBridge schedules that self-delete — no always-on compute between attempts.
- Idempotency, always. A deterministic key per invoice-and-attempt makes a double-charge impossible.
- Config, not code. The count and gaps are values you can change without redeploying logic.

Next: the message that goes out at each of those four attempts — a branded dunning email in your voice, with a one-tap link to update the card.

PART 4 OF 7

JUNE 21, 2026 PART 4 OF 7 · DUNNING RECOVERY SERIES ~7 MIN READ

How a dunning email gets sent

The retry does the recovering; the email does the asking. This post is about the message that goes out at each attempt — branded, in your voice, gentle on the first nudge and clear about the consequence by the last — with a one-tap link to update the card, sent through SES, and with bounces and complaints fed back so the system stops emailing a dead address.

KEY TAKEAWAYS

- A branded dunning email goes out at every retry attempt, from your verified SES sender — not a generic decline notice.
- One Bedrock Haiku 4.5 call writes the copy in your voice, escalating from a gentle nudge to a clear consequence.
- Every email carries a one-tap update-card link — a signed, single-use URL into the processor's hosted card page.
- Bounces and complaints are fed back through an SES configuration set, so the system stops emailing a dead address.
- The email only ever asks and informs; clicking nothing in it charges anything — the retry does the charging.

The retry charges; the email asks

It's worth being precise about the division of labour. The retry (Part 3) is what actually recovers the money — it asks the processor to charge the saved card again. The email does something different and just as important: it tells the customer what's happening and gives them a way to help. Most failed renewals recover not because the third automatic retry finally cleared, but because somewhere between retries the customer read a polite email and updated their card. The email is the human-facing half of the recovery.

That's why a generic, processor-default decline email is a wasted opportunity. It looks like spam, often lands in promotions, and gives no sense that a real business is behind it. The dunning email here goes out from your verified domain, with your logo, in your voice — so the customer recognises it as *you* and acts on it.

Copy that escalates gently

The four emails are not the same message sent four times. They escalate — carefully. The first is reassuring: cards fail all the time, no drama, here's the link. The middle two are a touch more direct. The last one is clear about the consequence without being a threat: if this final attempt doesn't go through, access will pause, and here's exactly how to avoid that. Getting that tone right four times, in your brand voice, is the one place a model earns its place in this system.

`dunr-mailer` reads a short HTML template and a tone brief from the `dunr-assets` S3 bucket, then makes a single **Bedrock Claude Haiku 4.5** call — a few hundred tokens — passing the attempt number, the plan name, the amount, and the retry date, and asking for subject and body copy at the right escalation level. The model writes the words; it does not decide anything. The amount, the dates, the link, and whether to send at all are all fixed by the deterministic code around it.

If the Bedrock call fails or times out, the mailer falls back to a plain templated version with the same facts — the customer always gets the email, even if it's the unstyled-copy version.

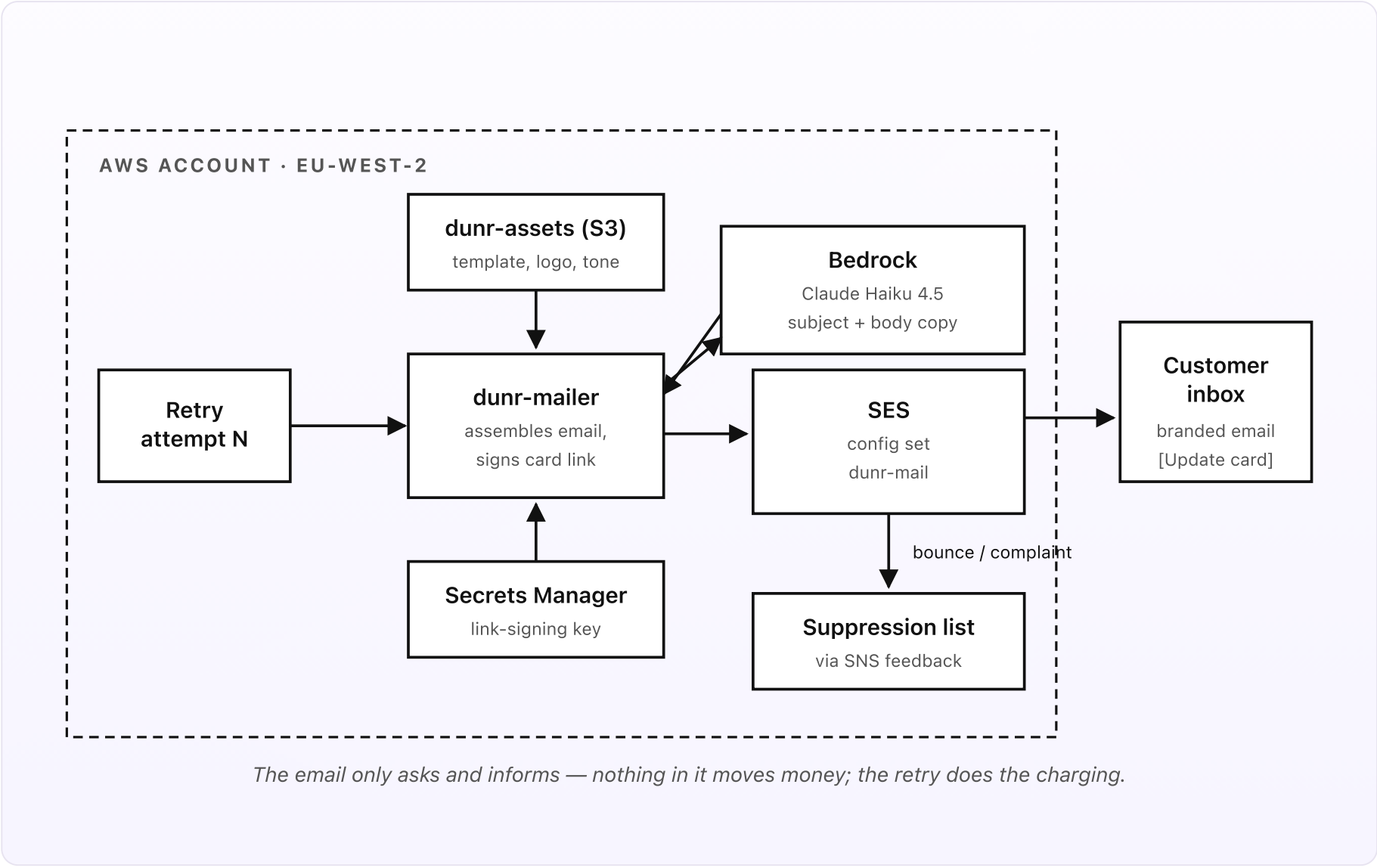


Fig 4. Assembling a dunning email. The mailer pulls the template from S3, gets brand-voice copy from one Bedrock Haiku call, signs a single-use update-card link, and sends through SES. Bounces and complaints feed a suppression list so the system stops mailing dead addresses.

The one-tap update-card link

The single most useful thing in the email is the button that lets the customer fix their card in two taps. The system never handles card details itself — that would drag it into PCI scope for no reason. Instead, the link is a **signed, single-use URL** that lands the customer on the processor's own hosted card-update page (a billing portal session or a setup link). `dunr-mailer` builds it with an HMAC over the customer id, the subscription id, and an expiry, keyed from a secret in Secrets Manager, so a forwarded or leaked link can't be reused after it's spent or expired. The customer taps it, updates their card on the processor's secure page, and the processor fires a `payment_method.updated` webhook — which (Part 2) tells the system to retry now rather than waiting for the next scheduled slot.

Not emailing a dead address forever

Sending mail responsibly means listening to what comes back. The SES **configuration set** `dunr-mail` routes bounce and complaint notifications to an SNS topic, and a small handler writes the affected address to a suppression list. If a customer's email hard-bounces (the mailbox doesn't exist) or they mark a dunning email as spam, the system stops sending to that address — the retries still run silently, but no more email goes to a place that doesn't want it. This protects your sender reputation, which is what keeps the *next* customer's dunning email out of their spam folder. It's also why the system is outbound-only on SES: there's no inbound mail to receive, unlike some other systems in this series.

DESIGN RULES THAT SHAPED THE EMAIL

- It looks like you. Verified domain, your logo, your voice — never a generic decline notice.
- It escalates gently. Reassuring first, clear about the consequence last; the model writes the words, not the facts.
- The model only writes copy. Amounts, dates, the link, and whether to send are all deterministic.
- No cards, ever. The update-card link is a signed, single-use URL to the processor's own hosted page.
- Listen to bounces. A configuration set feeds a suppression list, so the system never mails a dead address.
- Always falls back. If Bedrock is unavailable, a plain templated email with the same facts still goes out.

Next: the one thing the customer actually feels — how access pauses after the final failed retry, and how it's restored the instant a payment succeeds or a card is updated.

PART 5 OF 7

JUNE 21, 2026 PART 5 OF 7 · DUNNING RECOVERY SERIES ~7 MIN READ

How access gets paused and restored

Pausing access is the one thing this system does that the customer actually feels, so it has to be exactly right: never early, never silent, and instantly reversible. This post is about the gatekeeper — the small entitlement check your app calls on every request — how it flips a subscription to paused only after the last retry has genuinely failed, and how a successful retry or an updated card restores access in the same second.

KEY TAKEAWAYS

- Access is gated by one small entitlement check your app calls — `dunr-gatekeeper` behind a Function URL.
- Access stays granted all through the retry window; it pauses only after the *final* retry has genuinely failed.
- Pausing is reversible. A successful retry or an updated card restores access in the same second — no overnight batch.
- The gatekeeper reads state from `dunr-dunning` and answers in single-digit milliseconds, cached at the edge of your app.
- It never cancels. The strongest action it takes on its own is a pause; cancelling is always a human decision.

One question on every request

Everything in the previous four parts has been invisible to the customer. Pausing access is the one thing they actually feel — so it has to be exactly right: never early, never silent, and instantly reversible. All of that hangs on a single question your app already asks in some form on every request: *is this customer allowed in?*

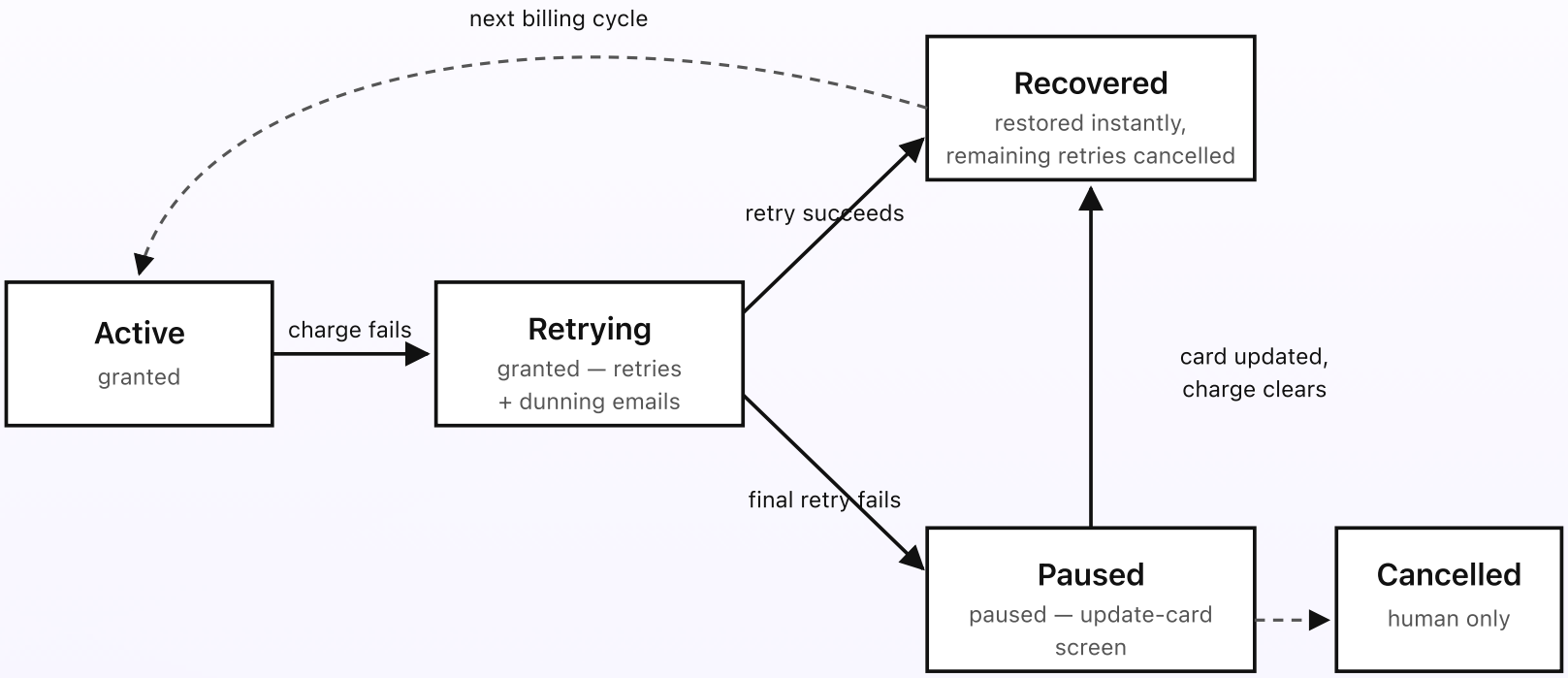
In this system that question is answered by `dunr-gatekeeper`, a small Lambda behind a Function URL. Your app calls it — on login, on a protected API call, on a page that costs you money to serve — with a subscription id, and gets back one of two answers: *granted* or *paused*. That's the entire contract. The gatekeeper reads the current state from the `dunr-dunning` table and translates it: *active*, *retrying*, and *recovered* all mean *granted*; only *paused* means *paused*. Because it's a single keyed read, it answers in single-digit milliseconds, and your app caches the answer for a minute or two at its own edge so the check adds no noticeable latency.

Access survives the whole retry window

A subscriber whose card just blipped should not be locked out while you're still trying to charge it. That would punish exactly the customers most likely to recover — the ones whose card expired or hit a temporary hold — and it would generate angry support tickets for people who are about to pay you anyway. So all through the *retrying* state, across all four attempts and the roughly ten days they span, the gatekeeper answers *granted*. The customer keeps full access while the retries and the dunning emails do their work quietly in the background.

Access flips to *paused* at exactly one moment: when the *final* retry fails. At that point `dunr-retrier` updates the row to *paused* and the next entitlement check returns *paused*. The customer now sees the update-card screen instead of the

product — not an error, not a dead end, but a clear “your payment didn’t go through; update your card to restore access” with the same one-tap link from the emails. Pausing is deliberately the strongest thing the system does on its own, and it’s a reversible state, not a deletion.



It never cancels on its own — pause is the strongest action, reversible the same second a payment clears.

Fig 5. The access state machine. Granted through the whole retry window, paused only after the final retry fails, and restored the instant a payment clears. Cancellation is reachable only by a human acting in your billing tool.

Restore is instant, not overnight

The flip side of pausing carefully is restoring fast. Nothing frustrates a customer more than fixing their card and then still being locked out "while the system catches up". There is no overnight batch here. Restoration happens on the same event-driven path as everything else:

- **A retry succeeds.** The processor fires `charge.succeeded`; the front door routes it; the row flips to *recovered*; the next entitlement check returns *granted*. If the subscription was paused, the customer is back in the moment they refresh.
- **A card is updated.** The customer taps the link and updates their card on the processor's page; the `payment_method.updated` webhook tells the retrier to attempt the charge *now* rather than waiting for the next scheduled slot. The moment that charge clears, access is restored — often within seconds of them entering the new card.

In both cases `dunr-gatekeeper` needs no special wake-up: it always reads the live row, so the instant the row says *recovered*, the answer is *granted*. The only lag is your app's own short cache on the entitlement answer, which is why that cache is kept to a minute or two.

Pause is not cancel

It's worth stating the boundary plainly, because it's the guardrail that defines this system. The strongest action dunning recovery takes on its own is to set a subscription to *paused* — a reversible flag that gates access and shows an update-card screen. It never deletes the subscription, never tells the processor to cancel it, never wipes the customer's data. Cancellation is a separate, deliberate

act that a human takes in your billing tool when they've decided this customer really has gone. When that happens the processor fires `subscription.cancelled`, and the system simply stands down — cancels any remaining retries, stops the emails, closes the row. The arrow into *cancelled* is the one arrow in the whole state machine the system never draws by itself.

DESIGN RULES THAT SHAPED ACCESS

- One check, two answers. The app asks `dunr-gatekeeper`; it replies *granted* or *paused*.
- Granted through retries. Nobody is locked out while you're still trying to charge their card.
- Paused only at the end. Access flips only after the final retry has genuinely failed.
- Restore is instant. A cleared charge or updated card restores access in the same second — no batch.
- Pause is reversible; cancel is human. The system never originates a cancellation.
- Every flip is logged. `dunr-log` records each pause and restore with a before-and-after snapshot.

That's the system end to end: caught, scheduled, emailed, paused, restored. The next post adds up what it all costs — and why the part that recovers the money is the cheapest part of the bill.

PART 6 OF 7

JUNE 21, 2026 PART 6 OF 7 · DUNNING RECOVERY SERIES ~6 MIN READ

What dunning recovery costs

This is one of the cheapest systems in the series, and the reason is worth stating plainly: the part that recovers the money — the retry schedule — is free, because it's plain Python doing date arithmetic. The whole bill is a few small variable lines on top of one fixed cost. This post breaks down exactly where the $1.90 a month goes, and what changes when you run ten times the volume.

KEY TAKEAWAYS

- About $1.90/month at ~150 active subscriptions and ~30 failed charges a month.
- The biggest single line is Secrets Manager — three secrets at $0.40 — because the actual work is so cheap.
- The part that recovers the money, the retry schedule, is free: it's plain Python doing date arithmetic.
- No always-on compute, no NAT Gateway, no API Gateway — the variable cost is pennies.
- At ten times the volume it lands around $4.90, because the fixed Secrets Manager cost doesn't grow.

The breakdown, line by line

The headline is almost funny: the single most expensive thing in a payment-recovery system is the place it keeps its passwords. At small-business volume — take ~150 active subscriptions, ~30 charges failing in a month, and around 75 dunning emails sent as those failures work through the retry schedule — here's where every cent of the ~$1.90 goes.

Service	What it does here	Monthly
Secrets Manager	3 secrets × $0.40 — processor API key, webhook secret, link-signing key	$1.20
CloudWatch Logs	All Lambdas, structured JSON, 7-day retention	$0.30
Bedrock (Haiku 4.5)	One small copy call per dunning email + the monthly report	$0.20
Lambda	Webhook, scheduler, retrier, mailer, gatekeeper, sweep — all arm64, short	$0.05
DynamoDB (on-demand)	Dunning state, audit log, dedupe — a few hundred small writes	$0.05
EventBridge Scheduler	One-off retry rules + the daily sweep (~100 invocations)	$0.03
SES (outbound)	~75 dunning emails at $0.10 per thousand	$0.02
S3	Email templates, logo, monthly report — a few MB	$0.02

Service	What it does here	Monthly
SQS (+ DLQ)	Buffers webhook events — a few hundred messages	$0.01
AWS Budgets	First two budgets are free	$0.00
Total		**~$1.90**

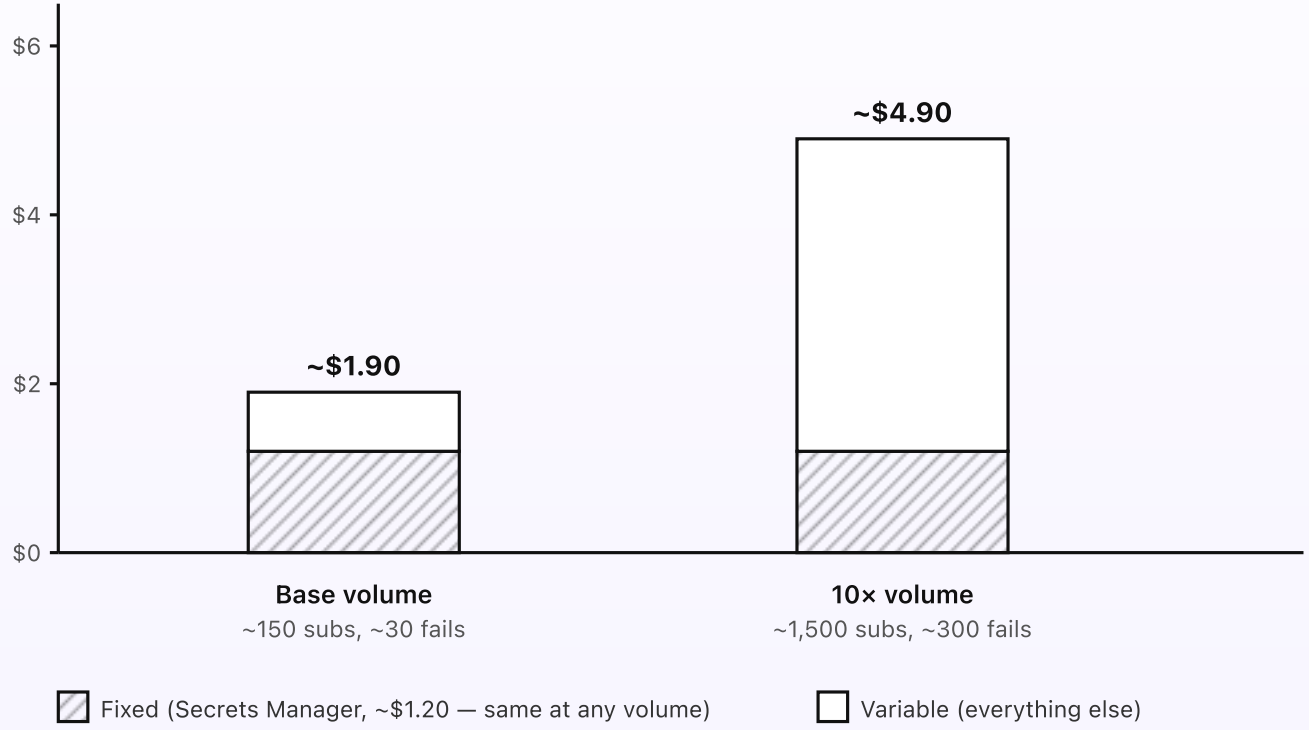
Where the dollars actually go

Secrets Manager (the bulk). Three secrets, each $0.40 a month, is $1.20 — almost two-thirds of the bill. That feels like a lot until you remember what they protect: the API key that can charge cards, the webhook secret that authenticates the processor, and the key that signs update-card links. Rotating them, scoping them per-Lambda, and keeping them out of environment variables is exactly where you *want* to spend a dollar. If you really wanted to trim it you could fold the link-signing key into Parameter Store (free for standard parameters), but the saving is $0.40 and the convenience of one secrets story isn't worth losing.

Bedrock (one small call per email). Each dunning email is a few hundred tokens in and a few hundred out on Haiku 4.5 — a fraction of a cent. Across ~75 emails it's well under twenty cents, and the monthly recovery report adds one slightly larger call. The retry schedule, the charge, and the pause all run with no model at all, so AI is a rounding error here, not a driver.

Lambda, DynamoDB, Scheduler, SES, SQS, S3 (pennies, together). This is the whole working machine, and it costs less than a dollar combined. The functions are short and arm64; the tables take a few hundred small writes; the one-off

schedules fire ~100 times; the emails are $0.10 per thousand. None of it is rounding up to anything meaningful at this volume.



The fixed cost is flat — ten times the work is far less than ten times the bill, and the retry schedule stays free.

Fig 6. Cost at base volume and ten times the volume. The Secrets Manager slice is the same height in both bars; only the variable slice grows. The retry schedule itself contributes nothing — it's plain Python.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the webhook receiver and the entitlement check.
- **NAT Gateway.** Nothing runs in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The system uses zero compute in the days between one customer's retries.
- **A scheduler that's always running.** No cron table to poll. Each retry is a one-off EventBridge Scheduler rule that self-deletes after it fires.
- **Models on the money path.** The schedule, the charge, and the pause are plain Python. Bedrock only writes email copy.

How the cost scales

At ten times the volume — around 1,500 subscriptions and 300 failed charges a month — the bill is about $4.90, not $19. The reason is the fixed/variable split in Fig 6: Secrets Manager stays at $1.20 no matter how many charges fail, while CloudWatch Logs, the per-email Bedrock calls, and the per-event Lambda and DynamoDB work scale with the number of failures. Logs tend to be the line that grows fastest, so structured-but-lean logging and the 7-day retention matter. Past a few thousand subscriptions you'd look at batching log delivery and at a provisioned-concurrency review — but those are tuning, not a redesign. The retry schedule never becomes the cost, because it never costs anything.

Set an AWS Budgets alarm at $10/month so anything unusual pages you before it matters. The recovered revenue dwarfs the bill at any volume: one saved £29 subscription pays for more than a year of running the whole system.

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

PART 7 OF 7

JUNE 21, 2026 PART 7 OF 7 · DUNNING RECOVERY SERIES ~10 MIN READ

Engineering reference: the dunning recovery architecture

Same system, drawn for engineers. Region, service names, resource identifiers, the Bedrock model id, the Lambda inventory, IAM scopes, the SES configuration set, the EventBridge Scheduler rules, the DynamoDB schemas, and the idempotency design that stops a double-charge. Read it alongside the previous six posts — this one is the build sheet.

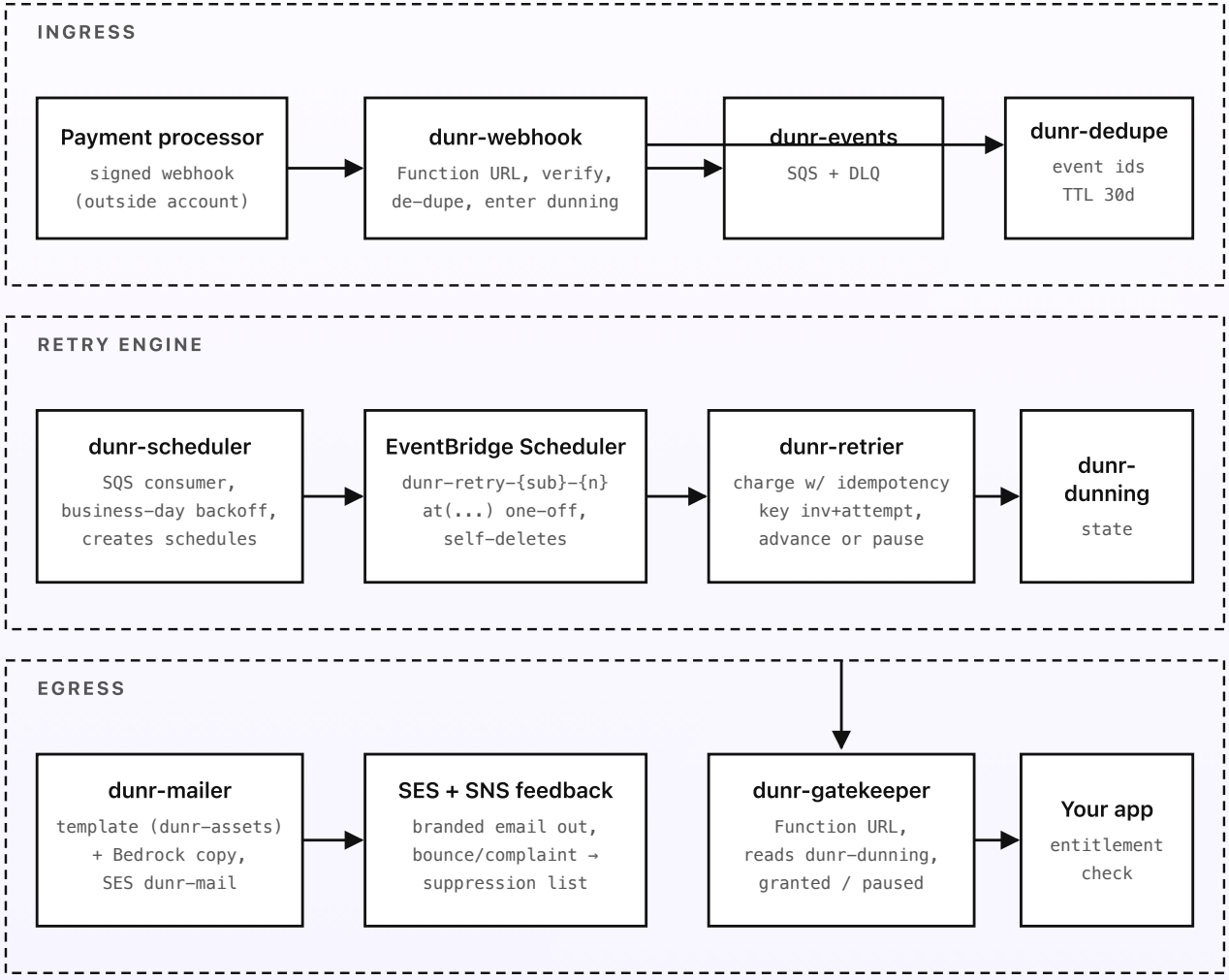
KEY TAKEAWAYS

- One region throughout: eu-west-2. One AWS account dedicated to the system keeps the IAM blast radius small.
- Seven Lambdas, three DynamoDB tables, one S3 bucket, one SQS queue with a DLQ, one SES configuration set.
- EventBridge Scheduler drives one-off retries (self-deleting) plus a daily sweep and a monthly report.
- The double-charge guard is a deterministic idempotency key on the invoice-and-attempt, passed to the processor.
- Bedrock Haiku 4.5 has exactly two callsites — the mailer and the monthly summary — never the money path.

Region and account shape

Default region: **eu-west-2** (London). SES outbound, Bedrock cross-Region inference, EventBridge Scheduler, and Lambda Function URLs are all in good shape there, and it keeps customer data in-region for a UK/EU subscriber base. A second region for resilience isn't worth the setup at this volume: the failure mode is a retry firing an hour late, not a regional outage, and because every charge is idempotent there is no in-flight payment to protect across a failover. One AWS account dedicated to dunning recovery (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every charge is idempotent and the system never cancels — the strongest state it sets on its own is paused.

Fig 7. AWS topology in three bands: ingress (the verified webhook into a queue), the retry engine (the scheduler creating one-off rules that fire the retrier against shared state), and egress (the mailer and the gatekeeper). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

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

- **`dunr-webhook`** — Lambda Function URL, public with `AuthType: NONE`; security comes from HMAC signature verification, not network auth. Verifies the processor signature against `dunr/stripe/webhook-secret`, rejects stale timestamps (replay window 5 min), de-duplicates by event id with a conditional `PutItem` on `dunr-dedupe`, writes `dunr-dunning`, and sends to `dunr-events`. Routes the four event types (failed / succeeded / payment-method-updated / subscription-cancelled). Returns `200` in milliseconds. Memory: 256 MB. Timeout: 10 s.
- **`dunr-scheduler`** — SQS trigger on `dunr-events`. Computes the business-day backoff in plain Python (next business day, then +3 business days ×3, weekends skipped; count and gaps from config), creates one-off EventBridge Scheduler rules `dunr-retry-{sub}-{n}` with `at(...)` expressions and `--action-after-completion DELETE`, and triggers the first dunning email. Memory: 256 MB. Timeout: 30 s. *No Bedrock.*
- **`dunr-retrier`** — EventBridge Scheduler target, one invocation per retry. Reads the live invoice from the processor (API key in `dunr/stripe/api-key`) and attempts the charge with a deterministic idempotency key `dunr-{invoice}-a{n}`. On success: marks `dunr-dunning` *recovered*, deletes

remaining `dunr-retry-{sub}-*` rules, restores access, fires a receipt email. On failure: advances the attempt or, if final, sets *paused*. Memory: 256 MB. Timeout: 30 s. *No Bedrock.*

- **`dunr-mailer`** — invoked per attempt by scheduler/retrier. Reads the template, logo, and tone brief from `s3://dunr-assets/`, makes one Bedrock Haiku 4.5 call for subject and body copy (falls back to a plain template on error), builds the signed single-use update-card link (HMAC from `dunr/links/secret`), and sends via SES with configuration set `dunr-mail`. Memory: 512 MB. Timeout: 30 s.
- **`dunr-gatekeeper`** — Lambda Function URL, public with `AuthType: NONE` but called with a short app-issued bearer token. Single keyed read of `dunr-dunning`; returns *granted* for active/retrying/recovered and *paused* for paused. Also exposes the internal pause/restore writes used by the retrier. Single-digit-ms responses; the app caches for 1–2 min. Memory: 256 MB. Timeout: 5 s.
- **`dunr-feedback`** — SNS trigger from the SES configuration set. Writes hard-bounce and complaint addresses to the suppression list so future sends skip them. Memory: 256 MB. Timeout: 10 s.
- **`dunr-sweep`** — EventBridge Scheduler target, daily at 9am local. Reconciles state: catches a retry whose schedule never fired (re-creates it), re-pings *paused* subscriptions that have a stale card, and closes rows past a configurable abandonment horizon. No Bedrock; a plain pass. Memory: 256 MB.
- **`dunr-summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month of `dunr-log`; one Bedrock Haiku 4.5 call writes a short narrative (charges recovered, value saved, recovery rate by failure code); emails it via SES and writes it to `s3://dunr-assets/reports/`. Memory: 512 MB.

Storage

- **DynamoDB · `dunr-dunning`** — one row per subscription in or recently out of dunning. PK `subscription_id`; attributes: `customer_id`, `status` (active/retrying/paused/recovered/cancelled), `attempt_no`, `max_attempts`, `next_retry_at`, `invoice_id`, `failure_code`, `access` (granted/paused), `schedule_names`, `entered_dunning_at`. On-demand.
- **DynamoDB · `dunr-log`** — one row per attempt or state change. PK `subscription_id`; SK `ts` (ISO-8601 + ULID suffix). Attributes: `event` (charge_failed/retry_attempted/retry_succeeded/email_sent/access_paused/access_restored/card_updated/cance `attempt_no`, `result`, `failure_code`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB · `dunr-dedupe`** — one row per processed webhook event id, for idempotency at the front door. PK `event_id`; attribute `seen_at`. TTL 30 days. On-demand.
- **S3 · `dunr-assets`** — email HTML template, logo, tone brief, and the monthly reports under `reports/`. Versioning enabled so a bad template edit rolls back in one click. No public access.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites only: `dunr-mailer` for the dunning-email copy, and `dunr-summary` for the monthly narrative. No model is ever on the schedule, charge, pause, or restore path.
- **Embeddings / Knowledge Base.** Not used. There is no corpus to retrieve over — the state is a handful of structured rows.

EventBridge Scheduler config

- **One-off retry rules** — created by `dunr-scheduler`, named `dunr-retry-{sub}-{n}`, `at(YYYY-MM-DDTHH:MM:SS)` expressions in the configured TZ, target `dunr-retrier`, with `--action-after-completion DELETE` so each self-cleans. Deleted en masse by name prefix when a retry succeeds.
- **`dunr-daily-sweep`** — `cron(0 9 * * ? *)` in TZ. Target: `dunr-sweep`.
- **`dunr-monthly-summary`** — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `dunr-summary`.

SES (outbound only)

- Verify a sender identity at `billing@your-company.com` with DKIM and SPF on the parent domain; out of sandbox by request. No inbound rule set — this system only sends.
- Configuration set `dunr-mail` with event publishing to an SNS topic for `Bounce` and `Complaint`; `dunr-feedback` consumes it and maintains the suppression list. Sends check the suppression list first.
- One template per escalation level lives in `s3://dunr-assets/`; the rendered subject/body come from Bedrock at send time, with the template as the deterministic fallback.

IAM (least privilege per Lambda)

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

- **dunr-webhook role:** `secretsmanager:GetSecretValue` on the webhook secret; `dynamodb:PutItem` on `dunr-dedupe` and `dunr-dunning`; `sqs:SendMessage` on `dunr-events`. No processor API key — it never charges.

- **dunr-scheduler role:** `sqs:ReceiveMessage` / `DeleteMessage` on `dunr-events`; `scheduler:CreateSchedule` / `DeleteSchedule` on the `dunr-retry-*` name prefix; `dynamodb:UpdateItem` on `dunr-dunning`; `lambda:InvokeFunction` on `dunr-mailer`. No `bedrock:*`.
- **dunr-retrier role:** `secretsmanager:GetSecretValue` on the processor API key; `dynamodb:UpdateItem` on `dunr-dunning` and `PutItem` on `dunr-log`; `scheduler:DeleteSchedule` on the `dunr-retry-*` prefix; `lambda:InvokeFunction` on `dunr-mailer`. The only role that can move money.
- **dunr-mailer role:** `s3:GetObject` on `dunr-assets`; `bedrock:InvokeModel` on the Haiku ARN; `ses:SendRawEmail` from the verified sender; `secretsmanager:GetSecretValue` on the link-signing secret.
- **dunr-gatekeeper role:** `dynamodb:GetItem` on `dunr-dunning`, and `UpdateItem` scoped to the access attribute. No secrets, no Bedrock, no charging.

The idempotency design (why it can't double-charge)

Two independent guards make a double-charge impossible. At the front door, `dunr-webhook` de-duplicates webhook events by id on a conditional write, so a replayed or re-delivered event is a no-op. At the charge, `dunr-retrier` passes the processor a deterministic idempotency key — `dunr-{invoice}-a{n}`, built from the invoice id and attempt number, never random. If EventBridge Scheduler fires the same retry twice, or a manual card update overlaps a scheduled retry, both requests reconstruct the *same* key, and the processor collapses the second into the first without taking money. The key is keyed on the attempt, not the clock, precisely so duplicates converge.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Metric filter on `"error"`, `"throttle"`, `"signature_fail"`.
- **SQS DLQ:** `dunr-events` has a dead-letter queue; a message that fails twice lands there with an alarm, so a malformed event never silently blocks the pipeline.
- **Alarms:** retrier failures > 0/hour; DLQ depth > 0; webhook signature failures > 5/hour (the secret may have rotated); SES complaint rate over threshold.
- **AWS Budgets:** $10/month threshold, alarm at 80% and 100% to an SNS topic `dunr-cost-alarm`.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) and AWS SAM. The opinionated bits: turn on S3 versioning for `dunr-assets` so a bad template edit rolls back in one click; give `dunr-scheduler` an SQS source with a DLQ so a single bad event never wedges the pipeline; and scope the processor API key to the *retrier* role alone, so it is the one and only function that can charge a card. Total deployable surface: seven Lambdas, three DynamoDB tables, one S3 bucket, one SQS queue with a DLQ, one SES configuration set, two recurring EventBridge schedules (plus the one-off retry rules created at runtime), and one Budgets alarm.

That's the full system. Six narrative posts and this engineering reference. If you want to talk about adapting it for your business, see Work with me.